# High-Performance Interconnect-Aware MPI communication for Deep Learning Workloads

Yıltan Hassan Temuçin

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) refers to using aggregate compute power of many small compute nodes to solve large complex problems which cannot be computed in a reasonable time on a single computer. In recent years these HPC clusters have moved towards using accelerators, such as Graphics Processing Units (GPUs), to offload computationally intensive portions of applications. Distributed Deep Learning workloads on these heterogeneous HPC systems has become increasingly important. These new workloads have been developed upon existing HPC libraries such as the Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA).

MPI communication is critical to distributed Deep Learning applications at scale as they place a large amount of pressure on the communication subsystem of HPC clusters. Improving the MPI communication run-time could benefit Deep Learning. For that, we first investigate the characteristics of Deep Learning applications to understand how we can propose and design communication mechanisms which solve some important communication challenges.

We focused on tackling the issues regarding large GPU messages, which we observed with Deep Learning applications. To begin our investigation, we studied NVLink usage within the context of point-to-point communication. Unified Communication X (UCX) framework, used within the Open MPI library, only utilises a small portion of the available NVLink bandwidth for intra-socket GPU-to-GPU communication. We propose a novel GPU-to-GPU data transfer mechanism that stripes the message across multiple intra-socket communication channels and multiple memory regions using multiple GPU streams to utilise all available NVLink paths. Our approach achieves 1.64x and 1.84x higher bandwidth for both UCX and Open MPI + UCX, respectively.

Then we propose a 3-stage hierarchical, pipelined `MPI_Allreduce` design that incorporates the new multi-path NVLink data transfer mechanism for intra-socket communication in the first and third stages of the collective, and PCIe and X-bus channels for inter-socket GPU-to-GPU communication in the second stage with minimal interference. For large messages, our proposed algorithm achieves a large speedup.

Finally, we evaluate our proposed `MPI_Allreduce` for Deep Learning applications such as Horovod + TensorFlow with a range of Deep Learning models. For Horovod + TensorFlow and VGG16, we observe up to 3.42x speedup over other MPI implementations.

# Statement of Collaboration

The work in Chapter 3 was completed collaboratively with Pedram Alizadeh. In Chapter 3, Pedram Alizadeh and I worked on the design of the profiler. Joint decisions were made on which metrics should be gathered and their importance. The PMPI profiler code was solely written by myself, as well as any data processing presented in this thesis.

In Chapter 4, AmirHossein Sojoodi and I collaboratively wrote code which showed that the multi-path copy was viable on the Mist compute node. This code was for a single process using multiple GPUs outside of the MPI library. The related results are not present in my thesis as that idea was extended to a multi-process environment and implemented inside UCX.

In Chapter 5, Pedram Alizadeh first mentioned the benefits of using a three stage algorithm and how that could potentially allow us to use the multi-path copy feature within the context of `MPI_Allreduce`. The design and implementation of the algorithm and the additional extensions, which are discussed in the thesis, was done by myself.

# Acknowledgements

I would like to thank my supervisor Professor Ahmad Afsahi for his guidance and support during this thesis. His passion and his pursuit for extending research in new directions has certainly helped me learn the complexity of research and what I am capable of. I would also like to acknowledge the Electrical and Computer Engineering (ECE) Department for their financial support through Teaching Assistantship, Queen's University for the Queen's Graduate Award (QGA), and The Natural Science and Engineering Research Council of Canada through Graduate Research Fellowships (GRF). Finally, I would like to thank Compute Canada for providing access to the Béluga, Cedar, and Mist compute clusters, as well as Fei Mao and Bruno C. Mundim at SciNet for their technical support on Mist.

Many thanks to my colleagues and friends at the Parallel Processing Research Laboratory; Dr. Mahdieh Ghazimirsaeeed, Pedram Alizadeh, Leila Habibpour Tanyani, AmirHossein Sojoodi, and Benjamin Kitor. Special thanks towards Pedram Alizadeh and AmirHossein Sojoodi for their collaboration during my thesis. I look forward to developing all these relationships both as friends and professionally as I start my PhD.

I appreciate my housemate Avi and his patience with me as we transitioned to working from home this past year. Lastly, I would like to thank my parents, Altan and Özlem, and my brother Ertan for their support during this period.

# Contents

# List of Algorithms

# List of Figures

# List of Tables

# List of Code Listings

# Glossary

| | |
|---|---|
| **ASIC** | An ASIC is an Application Specific Integrated Circuit. |
| **CPU** | The Central Processing Unit (CPU) is a processor of a computer system. |
| **CTS** | Clear To Send (CTS) is control signal where a receiver notifies the sender that they are ready to receive a data transfer. |
| **CUDA** | Compute Unified Device Architecture (CUDA) is an application programming model developed by Nvidia for their GPUs. |
| **D2D** | A term used to describe copying data from device memory to device memory. |
| **D2H** | A term used to describe copying data from device memory to host memory. |
| **DL** | Deep Learning is a subset of Machine Learning based on Artificial Neural Networks with representation learning. |
| **DNN** | A DNN is a Deep Neural Network with many layers between its inputs and outputs. |
| **FIFO** | First-In First-Out (FIFO) is a method of processing data in a data structure such as a buffer. The first data item placed in the buffer is the first to be removed and processed. |
| **FPGA** | An FPGA is a Field Programmable Gate Array. |

| | |
|---|---|
| **GPGPU** | General Purpose Graphics Processing Unit is a piece of computer hardware designed for Data-Level Parallelism. |
| **GPU** | See GPGPU. |
| **H2D** | A term used to describe copying data from host memory to device memory. |
| **HPC** | High-Performance Computing (HPC) is a large computer system used to solve complex problems. |
| **MP** | In this thesis we may refer to our multi-path design as MP. |
| **MPI** | The Message Passing Interface is a standardised and portable message passing standard for parallel programming. |
| **NCCL** | The NVIDIA Collective Communication Library (NCCL) is a multi-GPU communication library for Nvidia GPUs. |
| **NIC** | A Network Interface Card (NIC) is a piece of hardware that connects individual nodes to a network. |
| **NUMA** | Non-Uniform Memory Access is a computer memory design used in multiprocessors. |
| **NVCC** | Nvidia CUDA Compiler is a proprietary compiler by Nvidia which is used to compile CUDA programs. |
| **NVLink** | NVLink is a wire-based communications protocol for intra-node communication developed by Nvidia. |
| **NVVP** | The Nvidia Visual Profiler is a performance profiler for Nvidia CUDA code. |
| **PCIe** | PCIe (Peripheral Component Interconnect Express) is a high-speed serial bus standard for connecting high-speed components such as GPUs and network cards. |
| **RTS** | Request To Send (RTS) is control message where a sender asks to the receiver if they can transfer data. |

| | |
|---|---|
| **UCX** | Unified Communication X (UCX) is an open-source communication framework for high-performance applications. |

# Chapter 1

# Introduction

High-Performance Computing (HPC) often refers to using the aggregate compute power of many small compute nodes to solve large complex problems. This allows us to solve problems that would take months or years on commodity workstations. HPC systems are usually used for solving scientific and engineering problems within many domains. Recent improvements in HPC technology has resulted in advances in Artificial Intelligence (AI), Machine Learning, and Deep Learning (DL). Deep Neural Networks (DNNs) have been used to provide solutions to many problems within natural language processing, image recognition, autonomous vehicles, cancer detection, and medical imaging, to name a few. Training of these DNNs are usually limited by hardware resources which result in long training times and low productivity from researchers. As the size of data sets and the complexity of DNNs grow, the requirement for computing resources continues to increase. Scaling of these DL applications to massively parallel systems is required to continue solving problems in this domain.

Both HPC and Deep Learning applications place a large amount of pressure on the communication subsystem of HPC clusters. Their performance is significantly affected by the underlying hardware and software. Simultaneously, the prevalence of accelerators has developed a need for system software to adapt to these changes. General Purpose Graphics Processing Units (GPGPUs) are one of the most popular accelerators used today. We often see multi-GPU computing nodes with their own proprietary intra-node interconnects to accommodate communication between GPUs.

Developing applications on HPC systems relies on programming models designed for these systems. The Message Passing Interface (MPI) [1] is one of the most popular programming models used today with multiple implementations such as MPICH [2], MVAPICH2 [3], and Open MPI [4]. On GPU-based HPC clusters it is beneficial for MPI to be GPU-Aware and allow for optimised communication between GPU memory regions. MPI supports point-to-point, collective, and remote memory access (RMA) communication operations. MPI collectives, in particular, `MPI_Allreduce` and `MPI_Bcast`, which involve communications among a group of processes, play a crucial role in the performance of MPI applications, including Deep Learning workloads.

## 1.1 Motivation

MPI is the *de facto* standard for parallel programming within HPC environments. The MPI standard defines many different API calls to transfer data between processes. These data transfers place a lot of pressure on HPC systems as they are more costly than transferring data within a single process. MPI communication performance has been a large problem for HPC applications [5,6]. Therefore, Investigating MPI communication challenges is incredibly important as it affects many problem domains.

With the growth of Deep Learning, many frameworks have been developed to scale existing applications to HPC clusters using MPI. There are many frameworks available for distributed Deep Learning such as TensorFlow [7], PyTorch [8], Horovod [9], CNTK [10], and Deep500 [11]. Some frameworks, such as Horovod, implement distributed training using MPI but rely on other frameworks such as TensorFlow or PyTorch for the computation portion of their code. Other frameworks such as CNTK and Deep500 provide both distributed training tools and the implementation of Deep Learning models. As all these frameworks use MPI in some capacity, it is important to understand their usage. Some existing work shows that these applications use collective communication extensively [12–14] especially with large GPU messages. The exact usage of MPI collectives themselves is unclear and whether these applications use collective communication exclusively or a mixture of point-to-point and collectives. Gaining deep insight into the MPI communication characteristics such as the frequency of collectives or point-to-point, message sizes, or data volume, for a variety of DL frameworks, models, and optimisation algorithms will give us a clearer view of MPI-based Deep Learning.

Whether an application uses point-to-point or collective communication, improving point-to-point communication should improve most applications as collectives are often implemented upon point-to-point. The exact relationship between point-to-point communication and collectives differ between MPI implementations. Improving point-to-point communication for large messages can be achieved by fully utilising the available bandwidth. A higher bandwidth allows for data to be transferred in less time. As these Deep Learning applications are GPU-Centric it is important to investigate methods in which we can improve the available bandwidth for GPU-to-GPU data transfers.

## 1.2 Research Objectives

In this work we plan to investigate MPI based Deep Learning frameworks and GPU MPI communication. We aim to develop a better understanding of a few Deep Learning frameworks within the context of MPI and the communication characteristics that are present during their execution. Our goal is to study both CPU and GPU-based Deep Learning frameworks to identify their differences and similarities. As our main interest is in GPU-based Deep Learning, we also investigated GPU communication on Nvidia's intra-node interconnect NVLink [15]. Finally we try to draw a connection between the communication characteristics and GPU communication. In this work we aim to answer the following questions:

- Is MPI communication a major bottleneck to Deep Learning applications? Are there similar MPI communication problems across different applications or does each application have unique characteristics? Do we face communication challenges with data residing in GPU or CPU buffers? Does Deep Learning depend upon MPI point-to-point or collective communication, or both? Which MPI communication calls are called more frequently and which are the most important in regards to improving application run-time?
- Is intra-node communication important for Deep Learning applications? If so, can we improve GPU point-to-point communication within a single compute node? Could our improvements in MPI point-to-point communication help accelerate MPI collectives too?
- If MPI point-to-point or collective communication is problematic in distributed training of Deep Learning models, can we design communication mechanisms that have a direct impact on these applications?

## 1.3 Contributions

In this thesis we make a few observation with regards to Deep Learning frameworks and a few proposals for the MPI communication run-time. MPI communication is critical for distributed Deep Learning frameworks. Improving the MPI communication run-time should yield major performance improvements. We contribute by providing an improved point-to-point communication mechanism to better utilise bandwidth. We then use this mechanism to improve the `MPI_Allreduce` collective communication performance. Finally we see these MPI communication run-time improvements impact the application layer by accelerating Deep Learning workloads [16].

- We observe that `MPI_Allreduce` is the most prominent collective used by Horovod + TensorFlow, Horovod + PyTorch, and CNTK. For the CPU version Horovod + PyTorch, `MPI_Allreduce` takes up to 90% of application run-time at 512 processes. For both GPU configurations of Horovod, `MPI_Allreduce` takes up to 73% of application run-time at 64 GPUs. The message sizes used by both Horovod GPU configurations, resulted in mostly small CPU messages (less than 32B) and large GPU messages around 64MB. We observed that point-to-point communication was only used by CNTK and that point-to-point communication did not generate any message queue problems.
- We propose a novel multi-path GPU-to-GPU data transfer mechanism that partitions large point-to-point messages across device-to-device and device-to-host/host-to-device channels to utilise all available NVLink paths using a UCX one-sided put operation. Our approach achieves 1.69x and 1.84x higher bandwidth for UCX and Open MPI + UCX, respectively.
- We propose a 3-stage hierarchical, pipelined `MPI_Allreduce` collective design that utilises the new multi-path copy mechanism for intra-socket data transfers, while dynamically selecting NVLink and PCIe channels for different stages of the algorithm to minimise interference. Our experimental results show a speedup of up to 12.25x, 15.63x, 3.72x, 1.48x, and 1.38x against Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, MVAPICH2-GDR, and NCCL, respectively.
- We evaluate the impact of the proposed multi-path copy and `MPI_Allreduce` design at the application layer. For Horovod with TensorFlow training VGG16,

we observe up to 2.98x, 3.42x, 3.22x, 1.23x, and 3.24x speedup over Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, MVAPICH2-GDR, and NCCL, respectively. For ResNet50, we achieve 1.50x, 1.57x, 1.22x, 1.24x, 1.23x speedup over these MPI libraries, respectively.

## 1.4 Outline

The rest of this thesis is structured as follows: in Chapter 2, we introduce HPC clusters, MPI in greater detail, GPUs and Deep Learning. In Chapter 3 we analyse a few Deep Learning frameworks and observe their MPI communication characteristics. Then in Chapter 4, we propose a new multi-path UCX Put operation which better utilises NVLinks and we observe its impact on the MPI communication run-time. We then use this multi-path copy in `MPI_Allreduce` in Chapter 5, and evaluate it against Horovod + TensorFlow while studying a few different Deep Learning models. Finally we make our concluding remarks in Chapter 6.

# Chapter 2

# Background

Parallel computing is a form of computation where a problem is split into smaller parts which can be solved in parallel. Parallel computing is often applied to compute clusters as they are many small systems connected together with some form of network. Traditional compute clusters are homogeneous and are CPU based. Heterogeneous computing is growing evermore important in cluster computing as we move away from homogeneous systems towards those with accelerators. As of 2021, many forms of accelerators are being researched such as AI ASICs, FPGAs and GPGPUs. GPGPUs have been one of the most popular accelerators for HPC systems. From the list of the Top500 supercomputers, 6 out of the top 10 are GPU based systems [17]. Accelerators allow users to offload highly parallel and compute intensive portions of their applications using vectorised operations. Accelerator offloading usually outperform using the CPU alone. With these new GPU systems being available to researchers, applications are also moving in a direction to utilise accelerators. HPC applications have been modified to utilise these compute resources such as HOOMD-blue [18], LAMMPS [19], and many more. As discussed in [20] we saw the growth of GPU-Centric Deep Learning with [21, 22] as they were one of the first to start utilising their compute power in this domain.

In this section we provide background knowledge of HPC clusters, GPUs in this context and GPU-Aware MPI point-to-point and collective communication. Then we introduce parallel and distributed Deep Learning and their relation to the these HPC technologies.

## 2.1 Modern Heterogeneous HPC Clusters

A HPC cluster often contains a large number of nodes (individual computers) that are connected together to aggregate the compute power of the individual nodes. Having an aggregate of the compute power allows users to solve large complex problems which cannot be solved by a single machine in a reasonable time. In homogeneous systems, generally we have many CPU nodes of the same type. The individual CPU nodes can consist of one or more CPUs. Heterogeneous clusters are usually the same as homogeneous clusters but with the addition of hardware components such as accelerators. Hardware accelerators appear in many forms but their main purpose is to perform certain computations more efficiently than what can be computed on a CPU. Accelerators usually have certain compute operations implemented in hardware which often outperform general-purpose units. This usually decreases run-time and increases throughput of these operations. When working with systems with accelerators many challenges occur during software developments as we now need to account for communication time to transfer data to these accelerators.

Currently the most popular accelerator used in heterogeneous HPC systems are GPGPUs. GPUs were initially designed for hardware acceleration of computer graphics tasks. As GPUs are highly parallel, they eventually found their way into scientific computing. GPUs are connected to CPUs via various interconnects such as Peripheral Component Interconnect express (PCIe) or Nvidia's NVLink [15]. The intra-node interconnect topology can vary from system to system. Whether the system is homogeneous or heterogeneous, usually one or more Network Interface Cards

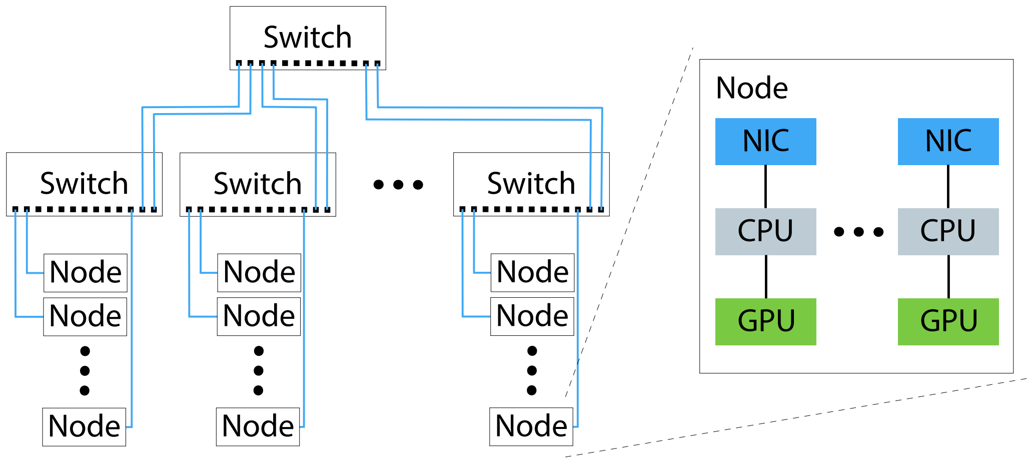


Figure 2.1: Schematic diagram of a modern heterogeneous HPC cluster

(NICs) are present to connect the nodes together using a switched network or fabric. Many network solutions exist such as Cray Aries Network [23], Intel Omni-Path Architecture [24], and InfiniBand [25].

In Figure 2.1 we can see a schematic diagram of a modern heterogeneous HPC cluster. The nodes are connected together in a fat tree topology where each node is connected to a switch. Then a second switch connects the first layer switches together. The second layer switches have higher bandwidth connections than the first. The nodes themselves contain a mixture of CPUs, GPUs, and NICs. The exact arrangement of the network switches and the intra-node topologies vary with each compute cluster.

## 2.2 Graphics Processing Units (GPUs)

GPUs are often found in the form of hardware accelerators in HPC systems. They are used to offload computation that would often be computationally intensive on CPUs. Within HPC, GPUs are generally used for floating-point operations computing matrices and vectors. There are many GPU vendors such as Nvidia, AMD, and Intel. The research conducted in this thesis is not bound to Nvidia GPUs but the work presented in this thesis will focus on Nvidia's V100 GPUs.

### 2.2.1 GPU Architecture

The V100 GPU is comprised of 84 Streaming Multiprocessors (SMs) [26]. SMs are the part of the GPU which execute CUDA kernels, but their design is significantly different than other hardware like CPUs. These GPUs also have a memory hierarchy; registers allocated to threads inside the SM, local memory to hold thread variables, shared memory for threads within a block to communicate, and finally global memory which is accessible by all SMs. The global memory in the V100 GPUs is High Bandwidth Memory 2 (HBM2). The send and receive buffers used in MPI will be allocated within global memory. We will discuss MPI further in Section 2.4.

### 2.2.2 Programming Nvidia GPUs

CUDA is a general-purpose parallel programming model that allows users to take advantage of Nvidia GPU's parallel compute engines. The CUDA environment allows users to program in C++ but CUDA can be interfaced with other languages such as C or FORTRAN. CUDA helps solve some of the challenges with transparently scaling applications in parallel environments. One of the main components of CUDA programming are kernels. These are similar to traditional functions in C/C++ but they can be executed in parallel using many CUDA threads. Each CUDA thread executes a kernel using its own thread ID.

#### Streams

CUDA applications concurrently transfer data and execute kernel via the concept of streams. A stream can be thought as a First-In First-Out (FIFO) queue of operations that will be executed in the order they are placed in the queue. Streams are in order with respect to themselves but work that is placed on multiple streams will not be in order relative to each other. Each stream will execute their commands concurrently. Streams can be synchronised individually or all streams on a single device can be synchronised.

#### CUDA Events

Nvidia devices can asynchronously record events. Recording of events are placed on streams. Once the stream reaches a record operation in the execution sequence, it will modify an event object. These event can also be queried to synchronise work between distinct streams.

#### CUDA Inter-Process Communication (IPC)

Device pointers are bound to the context that it was created in. Therefore, these pointers cannot be passed between processes via the usual means such as message passing or shared memory. To share device pointers and CUDA events between processes, the CUDA IPC API must be used [27]. Using the IPC API we can create a handle for a device pointer or an event and pass that handle to another process. That other process can open the handle to obtain a device pointer or event which can be used within that context.

## 2.3 Network Interconnects

GPU compute nodes usually have an interconnect to interface the multiple GPUs together within a single node. For the work presented in this thesis, we have worked with systems that use PCIe and/or NVLink. For HPC clusters, there is usually an interconnect to connect the individual nodes together. Many network solutions exist such as Cray Aries Network, Intel Omni Path Architecture, and InfiniBand. For the work in Chapter 3, we use an InfiniBand network.

### 2.3.1 PCIe

PCIe [28] is a high-speed interconnect used in compute systems to connect devices to CPUs. Many devices can be connected to the CPU with this interconnect such as GPUs, NICs, and SSDs. There are a few physical configurations for PCIe slots such as x1, x4, x8, and x16. Each configuration provides a different number of PCIe lanes to the attached device. An increase in lane count results in an increase in the available bandwidth to the device. GPUs attached via PCIe are often attached with x16 slots. For a PCIe 3.0 x16 slot, we would have a bi-directional bandwidth of 32GB/s and for PCIe 4.0 x16 slot we would have 64GB/s.

### 2.3.2 NVLink

NVLink [15] provides connectivity between Nvidia GPUs on multi-GPU systems. The main advantage of NVLink over PCIe for Nvidia GPUs is the increased bandwidth available for data transfers. The V100 GPUs discussed in this thesis has 6 attached NVLinks providing a total bi-directional bandwidth of 300GB/s (50GB/s per link). Depending on the system, the NVLinks are arranged in multiple different topologies. These topologies will be discussed in Chapter 4 when we introduce the compute systems that we will work with.

### 2.3.3 InfiniBand

InfiniBand (IB) [25] is a network specification maintained by the InfiniBand Trade Association. IB is used to connect high-performance compute node together which provides high throughput and low latency. The IB network used in the this thesis is Nvidia Mellanox EDR InfiniBand interconnect which provides 100Gb/s of bandwidth.

## 2.4 The Message Passing Interface (MPI)

The Message Passing Interface [1] is one of the most dominant programming models within HPC. MPI allows for the transfer of data between distinct processes. One of the main benefits of using MPI over other alternatives, such as the NVIDIA Collective Communication Library (NCCL) [29], is MPI's scalability, ability to use buffers from different memory regions, and portability. In Figure 2.2 we can see MPI's relation to other software libraries in a Deep Learning software stack.

MPI has three modes of communication; point-to-point, one-sided, and collectives. When using point-to-point communication, two processes are active with one sending data to another who is receiving. These send/recv pairs are matched. MPI guarantees that every message will arrive in order with no errors from the applications perspective. With one-sided communication, the data movement is decoupled from process synchronisation. One process exposes a memory region and another can read/write from that location. Only one process is active during the data trans-

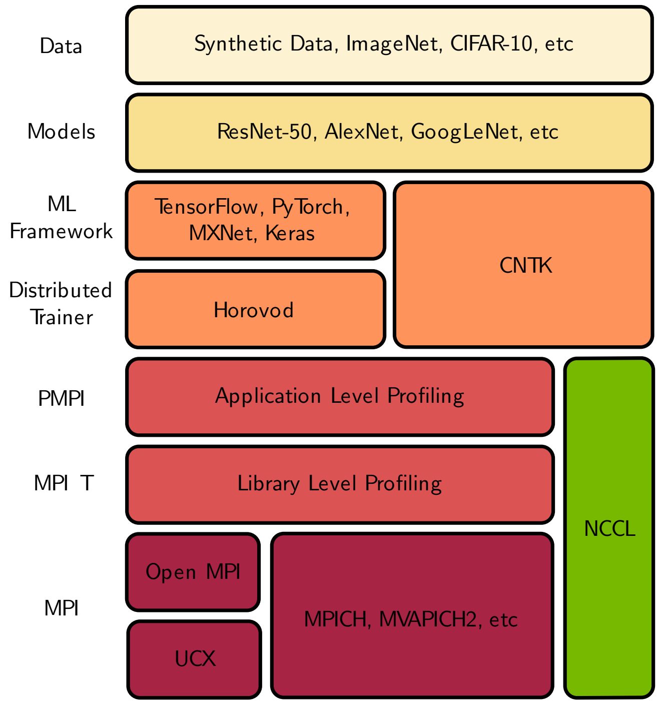

Data
Synthetic Data, ImageNet, CIFAR-10, etc
Models
ResNet-50, AlexNet, GoogLeNet, etc
ML Framework
TensorFlow, PyTorch, MXNet, Keras
CNTK
Distributed Trainer
Horovod
PMPI
Application Level Profiling
MPI T
Library Level Profiling
NCCL
MPI
Open MPI
UCX
MPICH, MVAPICH2, etc

Figure 2.2: Software Stack

fer. Collective communication allows for data to be transferred between multiple process in a predefined pattern between processes.

### 2.4.1 Point-To-Point Communication

Point-to-point communication involves two processes which directly transfer data between them. One process issues a send call and the other issues a receive call. The sent message contains two parts, the envelope and the data itself. The message envelope contains data about the message being sent such as the rank, tag and communicator. MPI provides both blocking (`MPI_Send` and `MPI_Recv`) and non-blocking (`MPI_Isend` and `MPI_Irecv`) routines for point-to-point communication.

Most MPI implementations have two protocols for sending messages, *Eager* and *Rendezvous*. Generally the Eager protocol is used for small messages and Rendezvous is used for large messages.

*Eager:* With the Eager protocol, the sender assumes that the destination process has sufficient allocated to store the incoming message. Therefore the sending process transfers all of the data to the receiver directly.

*Rendezvous*: With the Rendezvous protocol, the sender assumes that the destination process may not have allocated memory to store the incoming message. As we do not have the guarantee that there is allocated memory, the protocol executes a handshake to ensure that the data can be sent. The sender sends a Request To Send (RTS) control message to the receiver. Once the receiver has allocated memory to receive a message, it notifies the sender with a Clear To Send (CTS) control signal. Now the sender process transfers the data into the buffer of the receiver process.

#### MPI Message Queues

MPI implementations usually have two message queues to handle communication when two processes are out of sync. The two message queues are the Unexpected Message Queue (UMQ) and the Posted Receive Queue (PRQ). When `MPI_Recv` is called, the process first checks the UMQ to see if a message has arrived before `MPI_Recv` was called. If so, the message is removed from the UMQ. In the other scenario where `MPI_Recv` is called and the UMQ does not contain the message that the process is looking for, then the receive call is placed on the PRQ. Once the message arrives from the network, PRQ is checked for a matching receive call. If so, the receive call is removed from the PRQ and then the data transfer is complete. When searching for a message on the queue, a comparison is made with the message envelope that is created by the `MPI_Recv` call. Although this was described using blocking point-to-point calls the same is true for non-blocking calls.

### 2.4.2 Collective Communication

Collective communication occurs between a group of processes. This group is defined by an `MPI_Comm` object which is a communicator. The communicator `MPI_COMM_WORLD` refers to all processes that are present when a parallel job starts. Collective communication can be placed into four different categories; All-To-All, All-To-One, One-To-All, and Other [1,30]. Three of the categories can be seen in Figure 2.3. Within the All-To-All category all processes contribute to the result and all processes receive the result. This pattern can be seen in `MPI_Allreduce`, `MPI_Alltoall`, and

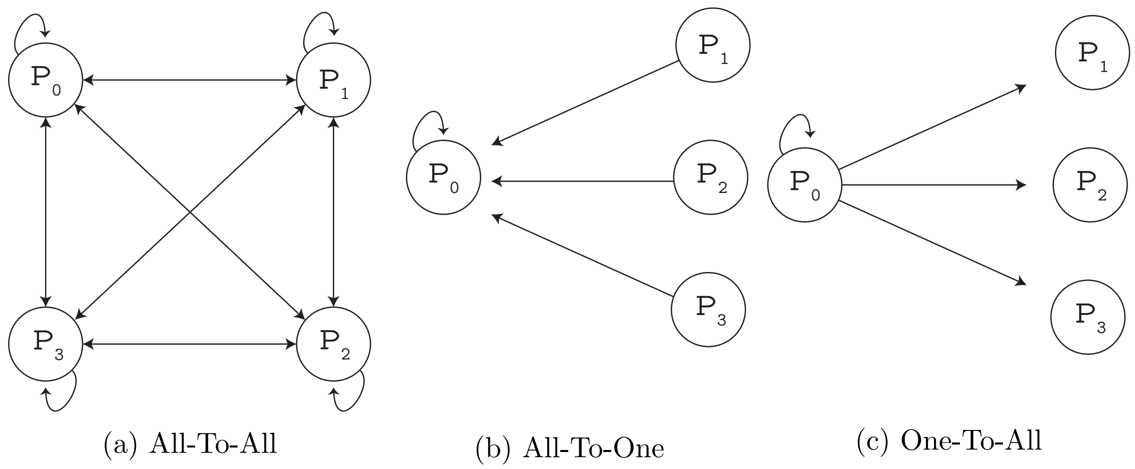


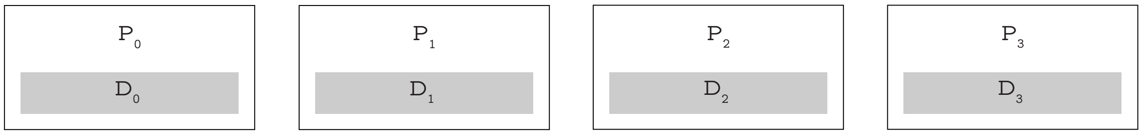
Figure 2.3: Different MPI Collective Communication Patterns

`MPI_Allgather`. All-To-One is where all processes contribute to the result and one process receives the result. `MPI_Reduce` and `MPI_Gather` are examples of this. In One-To-All communication, one process contributes to the result and all processes receive the result. `MPI_Bcast` and `MPI_Scatter` use this the pattern. MPI calls such as `MPI_Scan` and `MPI_Exscan` do not fit into the three previously discussed categories and are placed in the 'Other' category. For example, with `MPI_Scan`, $P_0$ receives it data from a single process but $P_n$ receives its data from all $n$ processes. Therefore, the communication pattern is different for each process in the communicator and does not fit in these three categories. For collectives such as `MPI_Allreduce`, `MPI_Reduce`, `MPI_Reduce_scatter`, and `MPI_Scan`, there is a computation that also occurs along the communication within the collective. This computation can be defined by the `MPI_Op` parameter.

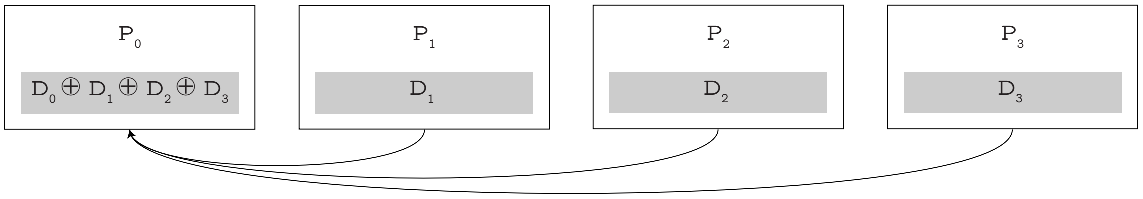
### MPI_Allreduce algorithms

This thesis will focus on the `MPI_Allreduce` collective as it is extensively used in Deep Learning and is a major performance issue for many frameworks. The performance issues will be discussed further in Chapter 3. Many different `MPI_Allreduce` algorithms exists, where they each optimise different metrics such as bandwidth or latency. Each algorithm performs differently based on process count, message sizes, and the system. We discuss the four algorithms implemented in Open MPI [4]: *Linear*, *Ring*, *Recursive Doubling*, and *Reduce-Scatter Allgather (RSA)*. The Ring and Segmented Ring algorithms will be discussed together as the difference between the two is small. Diagrams in this section are shown for four processes, as that is what we use in Chapter 5 but these algorithms work for any process count.

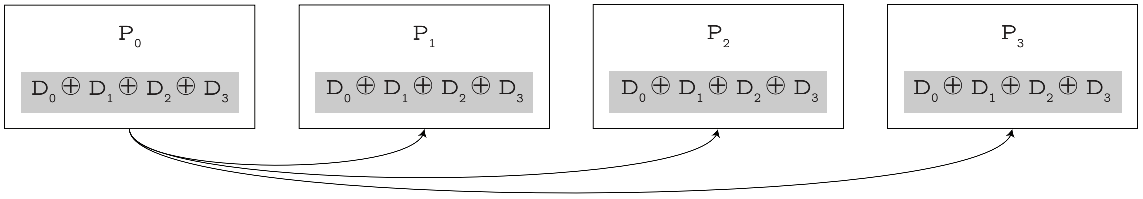
*Reduce-Broadcast (Linear):* This algorithm is one of the simplest implementations of `MPI_Allreduce`. It shows the fundamental behaviour of the algorithm in an easy manner. All processes directly send their data to the root process. In Figure



(a) Initial Data



(b) Step 1

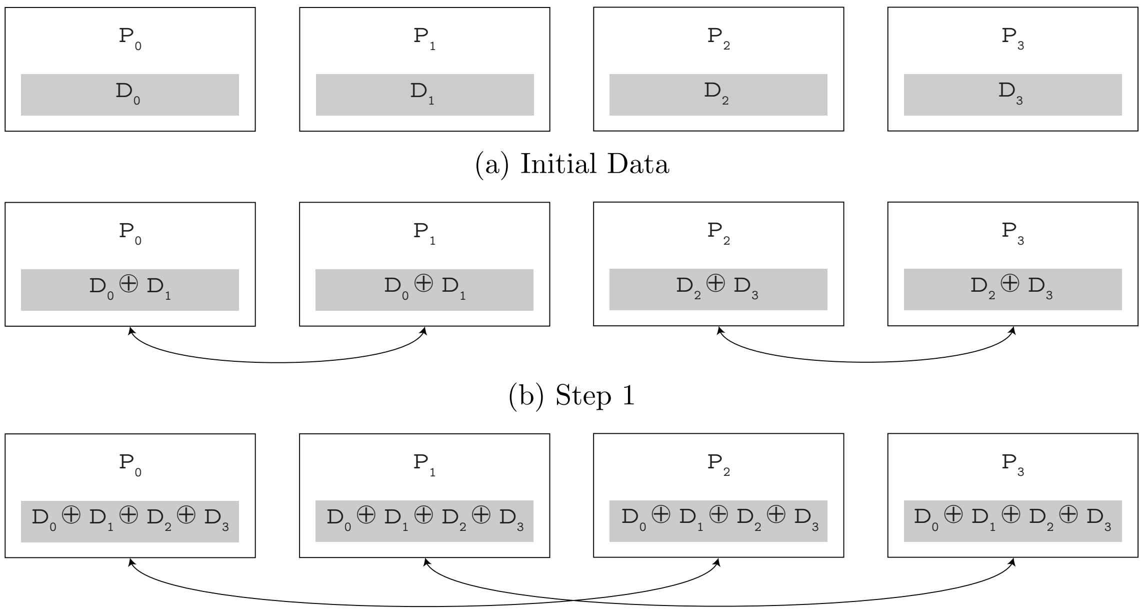


(c) Step 2

Figure 2.4: `MPI_Allreduce` Reduce-Broadcast algorithm for 4 processes

2.4 we show the root process as $P_0$. Once all the data has been received at the root, the process reduces the data. This reduction can be `MPI_SUM`, `MPI_MAX` or many other `MPI_Op` parameters. In the diagram the reduce operation is denoted as $\oplus$. Once the data has been reduced, the root process then sends a copy of the reduced data to all other processes in the communicator. The root process also copies the reduced data into its own receive buffer.

*Recursive Doubling (RD):* Figure 2.5 illustrates recursive doubling for four processes. In the first step processes exchange data with ranks that have a distance of one. Once each process has a copy of its partner processes data, it reduces the data. Then in Step 2, each process exchanges its data with ranks that have a distance of two. Again this received data is reduced. For four processes, this is the last step so data is then copied to the receive buffer. This algorithm can be generalised for any process count. Recursive doubling is often used for smaller message sizes.



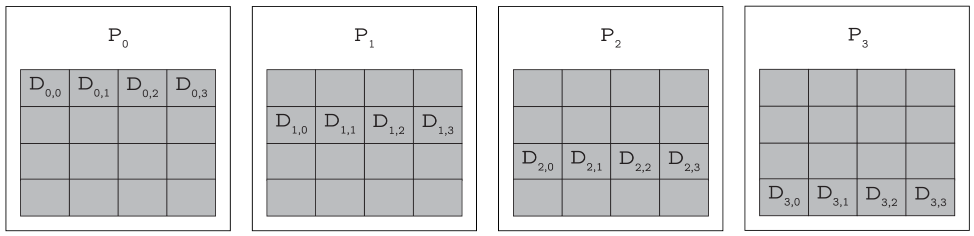
(a) Initial Data

(b) Step 1

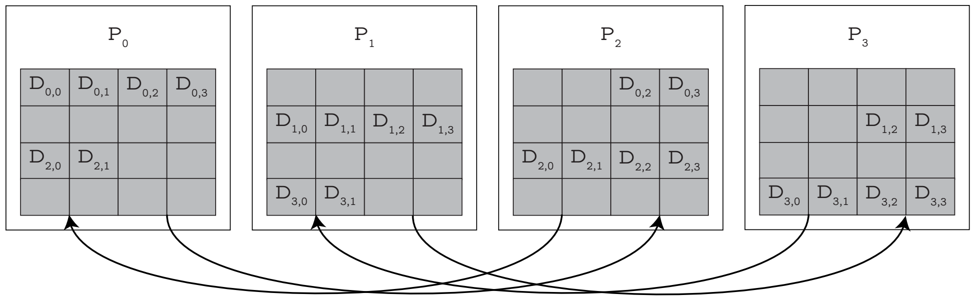
(c) Step 2

Figure 2.5: `MPI_Allreduce` Recursive Doubling algorithm for 4 processes

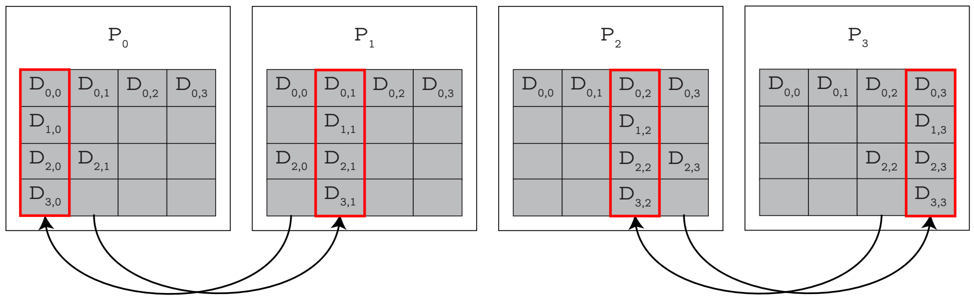
*Reduce-Scatter Allgather (RSA):* has two main parts as per its name. In Figure 2.6 we can see that Steps 1 and 2 act as Reduce-Scatter and Steps 3 and 4 act as Allgather. For the description of this algorithm we have not shown it with the $\oplus$ operator as this algorithm also divides the buffer during its execution. The red boxes denote that the data has been reduced. Here we denote the data $D_{rank,chunk}$ where each rank has four chunks. This algorithm can be generalised for $n$ ranks which will result in $n$ chunks. In Step 1 we send half of the buffer to processes with a distance of two. In Step 2 we send one quarter of the buffer to processes with a distance of one. Then we reduce the data. For each step the data size is halved. We can see from Figure 2.6 that the buffer chunks are chosen so that by the end of Step 2 we have executed a reduce-scatter. Now to complete the algorithm, we must execute Allgather to distribute the reduced data across the processes. In Step 3 we exchange the reduced data with processes with a distance of one. Then the reduced data is exchanged with processes with a distance of two. Now each process has a copy of the reduced data.



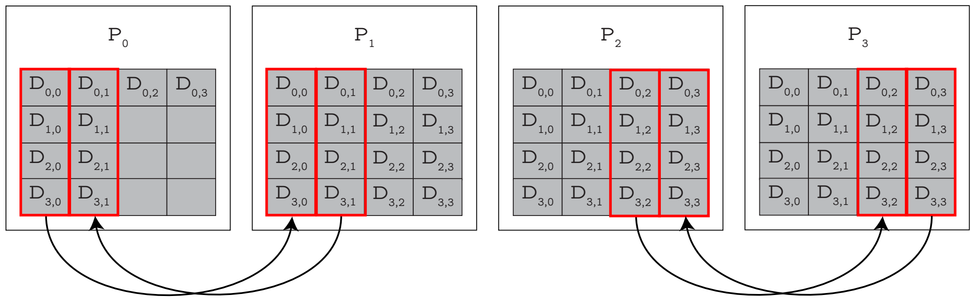
(a) Initial Data



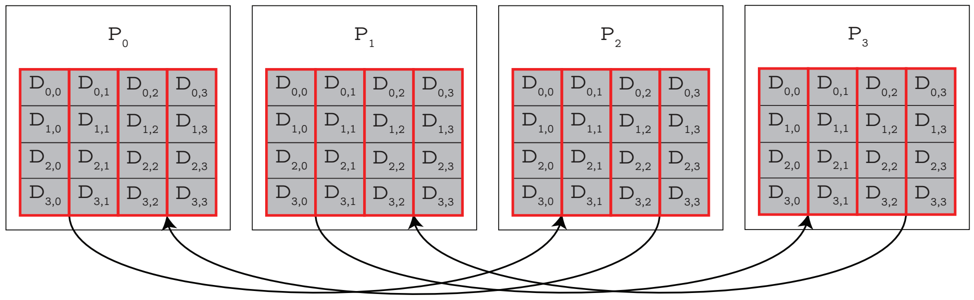
(b) Step 1



(c) Step 2

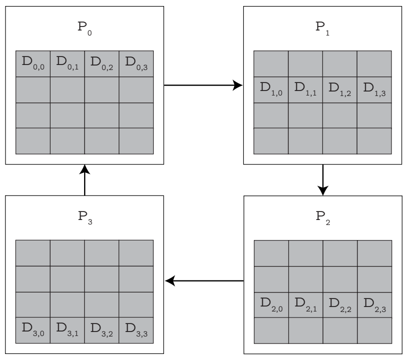


(d) Step 3

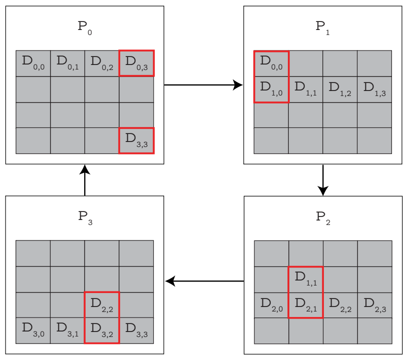


(e) Step 4

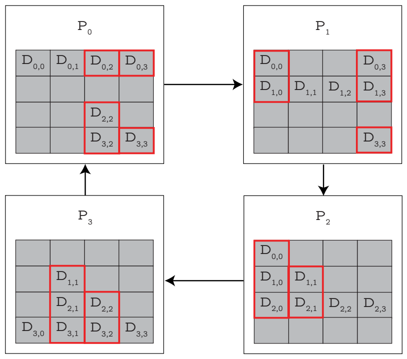
Figure 2.6: `MPI_Allreduce` RSA algorithm for 4 processes



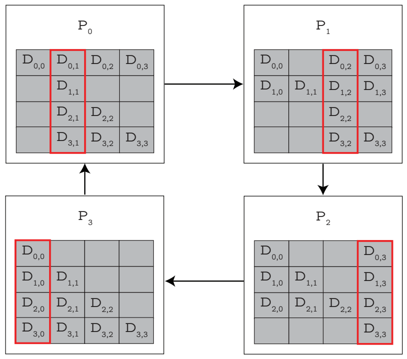
(a) Initial Data



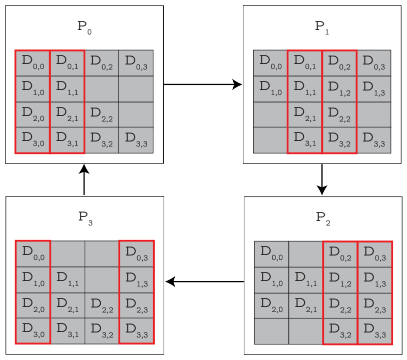
(b) Step 1



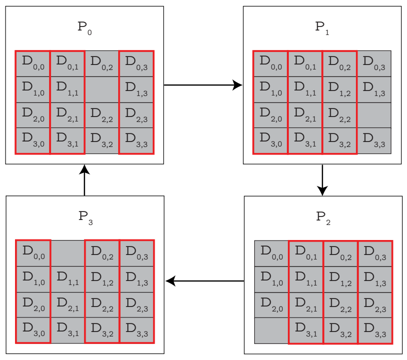
(c) Step 2



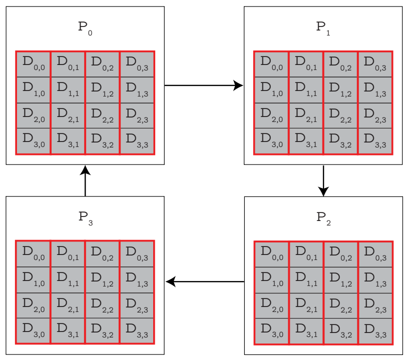
(d) Step 3



(e) Step 4



(f) Step 5



(g) Step 6

Figure 2.7: `MPI_Allreduce` Ring algorithm for 4 processes

*Ring:* Here we will discuss both the ring and segmented ring algorithm as they are fairly similar with small differences. In Figure 2.7 it can be seen that the processes are organised in a ring topology where each process sends to only one other process and receives from a single process. With the ring algorithm, the buffer is split into $N$ chunks. For the default ring implementation in Open MPI, $N = P$ where $P$ is the number of ranks. Steps 1-3 act as a ring-based reduce-scatter and Steps 4-6 act as an allgather. In Figure 2.7 we see the default ring implementation where $N = P$. So here the buffer is split into 4 chunks. At each algorithm step one chunk is sent to the next process. In Step 1 the first chunk is sent. In Step 2 the original data chunk is reduced with the chunk of data which was received in Step 1 and sent on to the next process. Step 3 is a repetition of Step 2 but as the rank now has all of the data the buffer is fully reduced for one chunk. This is shown with the red box. At the end of Step 3 the reduce-scatter is complete as each process has one quarter of the buffer that is fully reduced. In Step 4 the reduced data is sent to the adjacent rank. This is also repeated for Step 5 and Step 6 until every process has a fully reduced buffer.

In this example there are a total of 6 steps. This can be generalised for $P$ processes. The first reduce scatter step takes $(P - 1)$ steps, then the allgather step also takes $(P - 1)$ steps. Therefore the ring algorithm has a total of $2(P - 1)$ steps.

### 2.4.3 GPU-Aware MPI

When an MPI implementation is GPU-Aware it usually refers to being able to pass pointers into the send/recv buffer of GPU memory regions. The MPI implementation handles the explicit data transfer between GPU and the Host.

### 2.4.4 MPI Implementations

The Message Passing Interface is a standard which is approved by MPI Forum [1]. There are many implementations which follow this standard, some implementations are generic such as MPICH [2] and Open MPI [4] but others are are designed for specific hardware architectures such as Cray-MPICH [31] or Spectrum MPI [32]. In this thesis, the following implementations were used MVAPICH2, MVAPICH2-GDR [3], Spectrum MPI [32], Open MPI [4], and Open MPI + HPC-X [33].

### MVAPICH2

MVAPICH2 is an MPI-3.1 implementation based on MPICH. MVAPICH2 is an open source implementation provided by The Network Based Computing Laboratory (NBCL) at The Ohio State University (OSU). MVAPICH2-GDR is a closed source implementation which is highly optimised for GPU-based workloads. MVAPICH2 was used in Chapter 3 for the Deep Learning characterisation work. MVAPICH2-GDR was used in both Chapter 4 and Chapter 5 to compare results with the proposed work in that chapter.

### Open MPI + UCX

Open MPI is another open source implementation of MPI. It was used in both Chapter 4 and Chapter 5. Unified Communication X (UCX) was used for Open MPI's Point-to-point Management Layer (PML) component. Open MPI can use other point-to-point components in the PML, such as `ob1`, but better performance was observed when using UCX on the compute systems used in Chapter 4 and Chapter 5. Open MPI uses UCX's point-to-point features for its own point-to-point and collective implementation. When using point-to-point communication in MPI, we are directly using the point-to-point communication interface of UCX. For collective communication, UCX is abstracted through the PML layer, as shown in Figure 2.8. Figure 2.8 provides a slightly more detailed view of Open MPI + UCX than what was presented in Figure 2.2. Open MPI supports various flat algorithms for `MPI_Allreduce`, where the algorithm is chosen at run-time based on

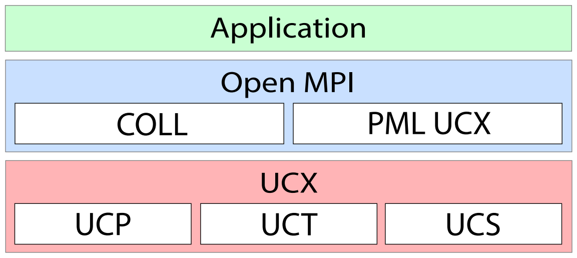


Figure 2.8: Simplified Software Stack of Open MPI + UCX

the number of processes and the message size. Flat algorithms are designed in a non-hierarchical approach and executed across all process in the cluster. In a flat algorithm, any pair of processes can communicate at the same cost. The following algorithms are implemented within Open MPI; ring, segmented ring, reduce-scatter-allgather, recursive doubling, nonoverlapping, and linear. Open MPI provides GPU support but for collectives with data residing on the GPUs it copies the data to the host and uses CPU based operations. Open MPI is not well optimised for GPU workloads.

Nvidia provides an additional component for Open MPI called HCOLL which is included in their HPC-X package. This component provides accelerated collective communication for Open MPI. For point-to-point communication, this component still depends on UCX. We have called this package Open MPI + HPC-X throughout this thesis.

*Unified Communication X (UCX):* UCX is an RDMA-based point-to-point communication library for modern low latency, high bandwidth interconnects [34]. It provides an abstract interface for communication that allows for network acceleration across many interconnects. In Figure 2.8, we see a simplified diagram of UCX relevant to this thesis.

*Unified Communication Protocol (UCP) Layer:* The UCP layer of UCX implements high level protocols that are used by other communication libraries such as MPI. UCP supports Remote Memory Access (RMA), active messages, and tag-matching operations, among others. The tag-matching interface is the most relevant to our work as it supports tag-matching for send-recv semantics of MPI. For this interface, UCP implements both the *Eager* and *Rendezvous* protocols. The UCP layer uses the UCT layer to implement these different protocols over a wide range of transports.

*Unified Communication Transport (UCT) Layer:* The UCT layer is a transport layer that abstracts the data movement across different memory regions. This layer uses low-level driver APIs such as InfiniBand Verbs, libfabrics, GDRCopy, and CUDA IPC to allow for efficient access to hardware with minimal overhead. This layer defines interfaces for small messages (short), buffered copy-and-send (bcopy), and zero-copy (zcopy) operations. In Chapter 3, it was shown that large GPU messages are important for Deep Learning workloads. Therefore, the zcopy operation for the CUDA IPC component of the UCT layer is important as this is where large GPU messages are transferred. This component handles GPU-to-GPU communication semantics via the usage of CUDA IPC. First, the receiver process places the CUDA IPC memory handle into shared memory. Then, the sender opens the handle and uses a Put operation to place the data into the remote process.

## 2.5 Distributed Deep Learning

Deep Learning is a subset of Machine Learning which often learns representations of data using Artificial Neural Networks (ANN). Deep Learning uses Deep Neural Networks (DNN) which is an ANN with many layers between its inputs and outputs. DNNs usually use convolution operations throughout a model architecture for many different models.

### 2.5.1 Distributed Training

There are two main methods of distributed training; data parallelism and model parallelism. At the time of writing this thesis, data parallelism seems to be the more popular method as its ecosystem is more mature. Model parallelism still appears to be in the research stage of development [35, 36].

### Data Parallelism

Training using the data parallel method involves having multiple processes, where each process has an instance of the Deep Learning model. Here the data set is split between the different processes and each process trains independently on that subset of data. This is useful for scenarios where the batch size cannot fit into GPU memory so that batch is split among processes. After a process trains its model, it must average the model parameters across all the processes to create a consistent global model. Usually a parameter server average the gradients across the different processes. If a single parameter server is used on rank 0, then all processes send their gradient to rank 0, and then average is calculated. Finally the average is broadcasted to all other processes. A parameter server can be centralised and have a single instance, as per the previous example, or it can be implemented in a decentralised manner where it is distributed across the processes. A decentralised parameter server is shown in Figure 2.9.

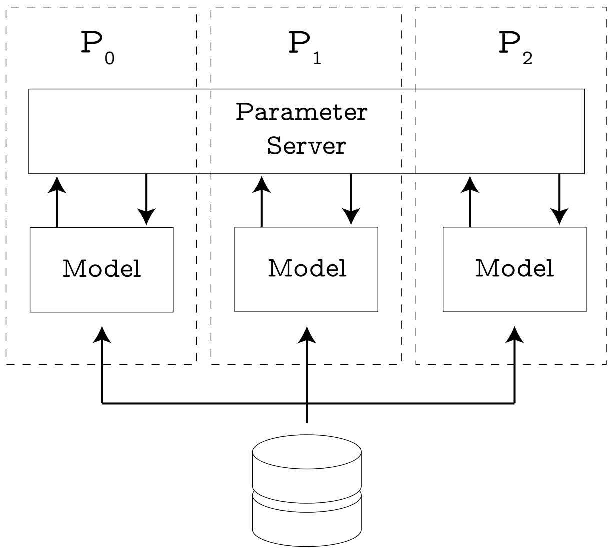


Figure 2.9: Schematic diagram of distributed training using data parallelism

**Model Parallelism**

Model Parallelism is where the Deep Learning model itself is parallelised. In Figure 2.10 we see that a model can be split among processes. Each process performs a part of the computation. This is usually used when a single model cannot fit into GPU memory or to create pipelining with computationally intensive layers. As Model Parallelism is not used in this thesis, this section is brief.

### 2.5.2 Deep Learning Frameworks

Programming Deep Learning models using language primitives is often a complex task. To simplify this process, many frameworks exist to provide high-level APIs containing the basic operations used in Deep Learning. In this thesis we will look at four popular frameworks: TensorFlow [7], PyTorch [8], Horovod [9], and CNTK [10]. These frameworks are built in mostly C/C++ and `Python` and they also provide their high-level APIs in those languages. CNTK also provides API in binding BrainScript.

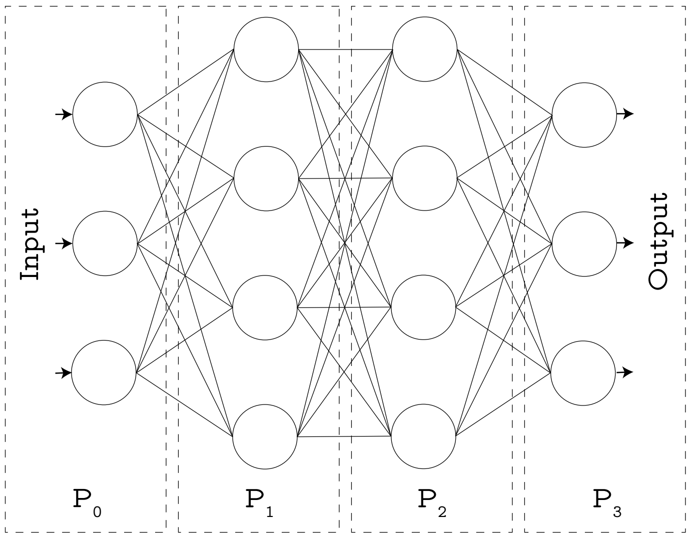


Figure 2.10: Schematic diagram displaying an example of model parallelism using 4 processes

### TensorFlow

TensorFlow is an open source Machine Learning framework, supported by Google, which allows for using symbolic math operations for data-flow and differentiable programming [7]. TensorFlow provides tools to deploy Machine Learning models in the web browser, mobile and IOT devices, and in production environments such as data centres. Although there was some work into using distributed TensorFlow using MPI [37], as of TensorFlow version 2.0, it no longer uses MPI for distributed training. For our work we are predominately interested in its implementation of GPU accelerated Deep Learning models, such as ResNet-50, which will be used with Horovod.

### PyTorch

PyTorch is a an open source Machine Learning framework which is built upon Caffe [38] and developed by FaceBook AI Research Lab (FAIR) [8]. PyTorch allows for multiple communication backends for distributed training such as: NCCL, MPI, and GLOO. It is also possible to train PyTorch models using a single process.

### Horovod

Models built using TensorFlow and PyTorch can be trained in a distributed fashion using Horovod [9]. Horovod does not have the tools to build the models themselves and relies on third party libraries. Horovod uses the *data parallel* approach to scaling DL models. The data set is split across multiple GPUs and are processed independently. Then an allreduce algorithm is used to average the gradient and distribute the results [9]. This allreduce can be from NCCL, MPI, or a mixture of both libraries.

Horovod has a feature called *Tensor Fusion* which enables the overlap of communication and computation by batching data for `MPI_Allreduce` operations. Tensor

Fusion first determines which tensors are ready to be reduced. Then it allocates the fusion buffer, whose size is determined by the run-time parameter `HOROVOD_FUSION_THREASHOLD`. The default size of the buffer is 64MB. Finally, the buffer is populated with the selected tensors and the reduction is executed. These steps are repeated until all tensors have been reduced.

### CNTK

The Microsoft Cognitive Toolkit (CNTK) is an open-source distributed Deep Learning toolkit [10]. This toolkit is commercial grade. CNTK allows users to create many different models such as DNNs, CNNs, etc. Alongside the models, CNTK also implements a distributed Stochastic Gradient Descent (SGD) which is an iterative method for optimising an objective functions used in distributed training. CNTK uses MPI for its distributed training. CNTK is flexible in that it allows users to program an application in multiple different languages: Python, C#, C++ or BrainScript. As of 2019, CNTK has been deprecated. It was included in this thesis as the work had been completed before the deprecation. That said, studying it still has its own merit from a research perspective as it is important to study many different Deep Learning applications.

# Chapter 3

# Communication Characterisation of Distributed Deep Learning Frameworks

In recent years there has been an increase in the popularity of Deep Learning applications, both to solve tradition problems in HPC and in new areas. However, the behaviour of these applications still have some degree of unknown characteristics from a system software perspective. In this chapter, we study the characteristics of distributed Deep Learning frameworks to find opportunities for optimisation.

As discussed in Chapter 2, distributed Deep Learning using MPI is an effective method to train these applications at scale. MPI usage can vary greatly from application to application. The behaviour of the MPI run-time is very dependent on which API calls are made. Depending on which portion of the library that an application uses, largely dictates which research areas could improve their performance. In this chapter, we characterise a few Deep Learning applications using the PMPI profiling interface of MPI. Application level profiling will allow for various metrics to be gathered such as frequency, time, and message sizes used by each API call.

Such metrics can guide research into the MPI run-time by determining potential performance problems. In this chapter we make the following contributions:

- We profiled two popular distributed Deep Learning frameworks, Horovod and CNTK. With Horovod we looked at using the framework with both TensorFlow and PyTorch for both CPU and GPU-based Deep Learning models. For CNTK, we investigated four different training algorithms.
- We measured the frequency of MPI point-to-point and collective communication calls made by both frameworks to determine which MPI API calls are of merit. From this, we measure the impact of MPI on application run-time for these calls. For the most used MPI calls, we collected and analysed the most frequently used message sizes.
- We also investigated a configuration of Horovod which uses a mixture of NCCL and MPI at run-time to see how NCCL can be used to offload MPI communication.
- For CNTK, we measured how long is spent in MPI message queues and its impact on application run-time.

## 3.1 Related Work

MPI communication characterisation has previously been studied in great depth for traditional HPC applications. A recent survey on US exascale computing projects showed that 73% of projects use MPI [39]. Application developers expect 89% of point-to-point communication and 93% collective communication to be performance critical for the exascale version of their applications. They also expect for applications to use multiple threads per process in 86% of their projects. A study on MPI usage on a production supercomputer showed that currently 30% of MPI jobs use multiple threads per process [40]. Although there is a large discrepancy in these values, it still shows that multi-threaded MPI is becoming more prominent.

MPI often contributes to large application performance bottlenecks as 15% of jobs spend over 80% of application run-time in MPI. Collective communication accounts for around 60% of total MPI time [40]. Five or fewer MPI calls account for 90% or more of the total communication calls for a particular application [6]. `MPI_Allreduce` and `MPI_Bcast` are the most frequently used collectives. `MPI_Allreduce`, `MPI_Alltoall`, and `MPI_Bcast` transfer the largest volume of data. Applications which use collectives often send messages smaller than 2KB while those which use point-to-point often send larger messages [6]. Chunduri et al. found that 40% MPI jobs used messages less than 256B for reduction operations. In [41], most of the applications which they studied showed that smaller message sizes (<1KB) are more prominent. Each study shows slightly different results which suggests that this type of work is tightly coupled to the applications under test.

The communication pattern of point-to-point messages used by applications gives some insight into the network utilisation. Kamil et al. measured the topological degree of connectivity and found most applications have a low degree of connectivity and do not take advantage of the fully connected network that exists in many HPC systems [6]. Similar results were shown by Zamani and Afashi, and they also noted that some applications have a substantially higher frequency of sends/recvs to process 0 [41]. Frequent communication to a single process often results in applications developing long message queues [42].

None of the above works study Deep Learning workloads; as previously noted these results are application dependent. We were only able to find a two papers which characterised MPI communication for Deep Learning applications [13, 43]. Only a few metrics were gathered and they were specific to Horovod's `MPI_Allreduce`. They measured the latency of each message size used by the application. They also investigated Horovod's fusion buffer by presenting results on how message sizes change with enabling and disabling it. This paper did not look into the other collectives used by this framework and how MPI impacts the application's total run-time.

The few papers which look into collective optimisation for Deep Learning applications do give some insight into what would be the expected behaviour of MPI communication. The performance of CNTK was improved by up to 47% by focusing on `MPI_Bcast` for large message sizes. Also, a modified CUDA-Aware CNTK with an optimised `MPI_Bcast` used message sizes up to 250MB [12, 44, 45]. They discussed different Deep Learning models but did not state which SGD algorithm they used for their evaluation, therefore it is somewhat unclear where these large message sizes occur. Other work focused on the improvement of `MPI_Allreduce` for large messages [46] but this was mostly a performance evaluation of TensorFlow.

Although it is clear that "large" messages are used for Deep Learning workloads, to our knowledge no work exists that directly quantifies the fine-grained details of MPI communication such as frequency of collectives or point-to-point, message sizes, or data volume for a variety of Deep Learning frameworks, models, and optimisation algorithms.

## 3.2 Motivation

Due to the popularity of Deep Learning, many frameworks exist for scientist to develop Deep Learning models. Each framework has a unique set of features, but to HPC researchers their MPI functionality is important. Without gaining a better understanding of an application's behaviour with respect to the MPI run-time, it is difficult to determine what could be potential performance problems.

Existing work indicates that the latency of `MPI_Bcast` and `MPI_Allreduce` are both problematic in Deep Learning workloads, as described in the related work section of this chapter. Although some existing work exists in this area, it is not at the level of granularity that is required for our study and of interest to the research community. Gathering metrics related to these applications at run-time using the PMPI profiling interface will allow us to observe the API entry points to the MPI library and speculate what can be done to improve Deep Learning workloads. Alongside the MPI calls that are made, we would like to know their frequency and their impact on application run-time. Also, understanding which messages sizes are used and from which memory regions could direct the design and development of proposed methodologies to improve communication performance. These metrics could help focus which MPI calls to target and and what performance issues could yield the best outcome if they are resolved.

## 3.3 Dynamic Analysis of Deep Learning Frameworks

### 3.3.1 Experimental Setup

#### Béluga GPU Cluster

Béluga is a GPU cluster located at École de Technologie Supérieure in Montréal, Canada. This cluster is one of Compute Canada's super-computing research facilities. Béluga has four Nvidia V100SMX2 (16G HMB2 memory) GPUs per node. The GPUs are all connected to each other via NVLink; each GPU is one hop to every other GPU. It is populated with two 20-Core Intel Xeon Gold 6148 (Skylake) at 2.4 GHz where all four GPUs are connected to a single socket. Each node has 186GB of RAM. The nodes are connected using Mellanox EDR (100Gb/s) InfiniBand interconnect. Béluga uses the GNU/Linux distribution CentOS 7.3. This cluster has a total of 172 GPU nodes.

### Niagara CPU Cluster

Niagara is a homogeneous CPU Compute Canada cluster hosted by the University of Toronto. It is populated with two Intel Xeon Gold 5115 (Skylake) at 2.4GHz for a total of 40 cores per node. Hyper-threading is enabled on this cluster with two threads per CPU core. Each node has 188GB of RAM. The nodes are connected via Mellanox EDR (100Gb/s) with a Dragonfly+ topology. Niagara uses the GNU/Linux distribution CentOS 7.4. This cluster has a total of 2024 compute nodes.

### Software Platform

For both clusters and all frameworks, we are using MVAPICH2-2.3 and everything has been compiled using gcc 7.3.0. On Béluga, our MPI implementation is CUDA-Aware using CUDA 10.0.130. For our NCCL tests, we are using version 2.5.6-1. Since GPU Direct RDMA is not available on Béluga, we have not conducted any tests using MVAPICH2-GDR. We have used CNTK 2.6 compiled from source as we found some compatibility issues with the default pip packages on CentOS. Horovod (0.18.0) has been compiled to use TensorFlow (1.13.0) and PyTorch (1.2.0). The CPU and GPU versions of TensorFlow and PyTorch are used for the appropriate platforms.

For CNTK, we used ResNet-20 from the example folder. Slight modifications were made to use the four of the distributed training methods offered by CNTK. We ran the models for one epoch of training using CIFAR-10 to approximate a workload for this framework. For CNTK, we used one process per GPU with the CPU cores distributed evenly between processes.

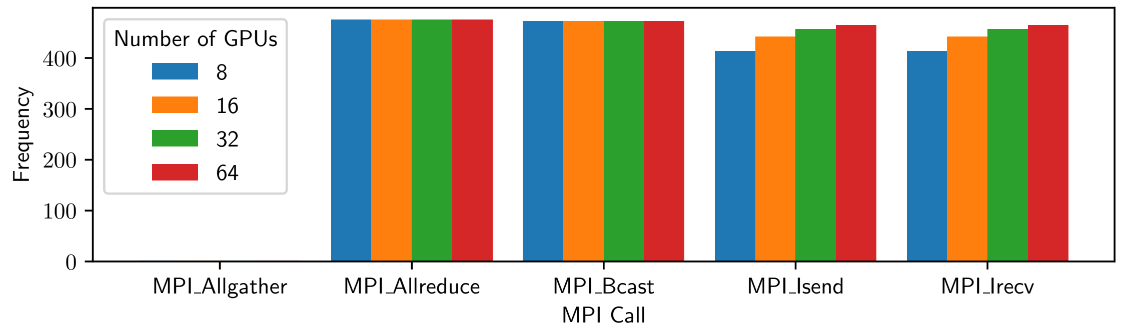
The Horovod synthetic benchmarks measure the image classification throughput (images/sec). This benchmark runs ten batches for ten iterations per process to roughly emulate training of 1 epoch. The default configuration with a batch size of

32 is used for both CPU and GPU versions. Although throughput measurements would benefit from an increased batch size [46] we chose to keep the batch size as a constant variable in our tests. This decision was made so that the comparison between CPU and GPU workloads could be consistent. The usage of synthetic data removes I/O as a variable from our results which is important as we are focusing on MPI communication time. For the GPU version we have assigned the default one process per GPU and for the CPU version we have used the four process per node configuration outlined in [47]. For both platforms, we have distributed the available number of CPU cores evenly between processes when working with Horovod.

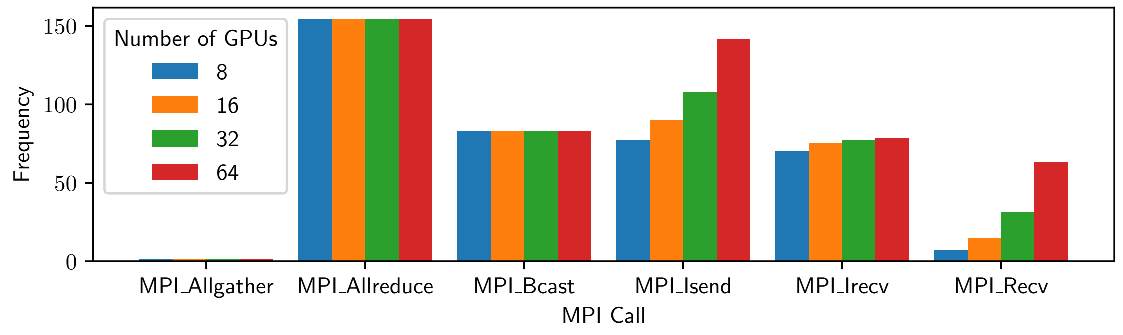
A custom MPI profiler is used to obtain the results. The profiler design uses the `PMPI` and `MPI_T` interfaces [1]. Using `PMPI` we gather data from the application layer such as message size, count, type, frequency, and the time spent in each MPI call. The `MPI_T` interface was used to profile message queues from inside MVAPICH2. Our profiler has an overhead of approximately 1% at 64 GPUs. The complete software stack was introduced in Chapter 2 and can be seen in Figure 2.2. The profiler stores all measured values into memory and prints the traces into files in `MPI_Finalize`. All characteristics results are obtained from analysing the traces.

### 3.3.2 Frequency of MPI API calls

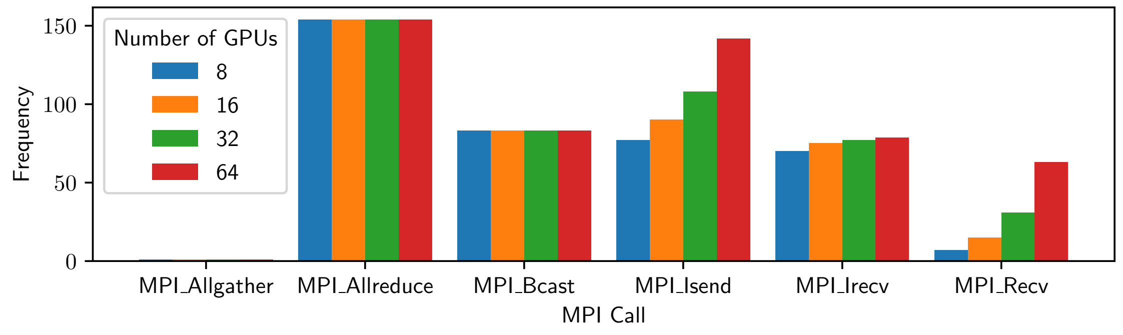
Figures 3.1-3.4 show the frequency of various MPI calls made per process at the application layer for CNTK, TensorFlow, and PyTorch, respectively. When profiling CNTK we looked at four distributed training methods: `DataParallelSGD`, `BlockMomentumSGD`, `ModelAveragingSGD`, and `DataParallelASGD`. For both Horovod and CNTK we noticed that there was only a single communicator and that `MPI_Allgather` is called exactly once per process, suggesting that it is used for initialisation. For CNTK, `MPI_Allreduce` and `MPI_Bcast` are the most frequently called collectives in terms of frequency across all our tests.



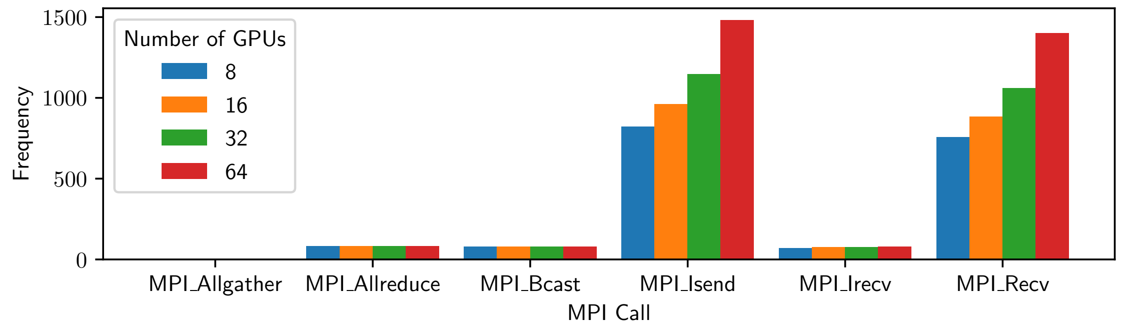
(a) `DataParallelSGD`



(b) `BlockMomentumSGD`



(c) `ModelAveragingSGD`



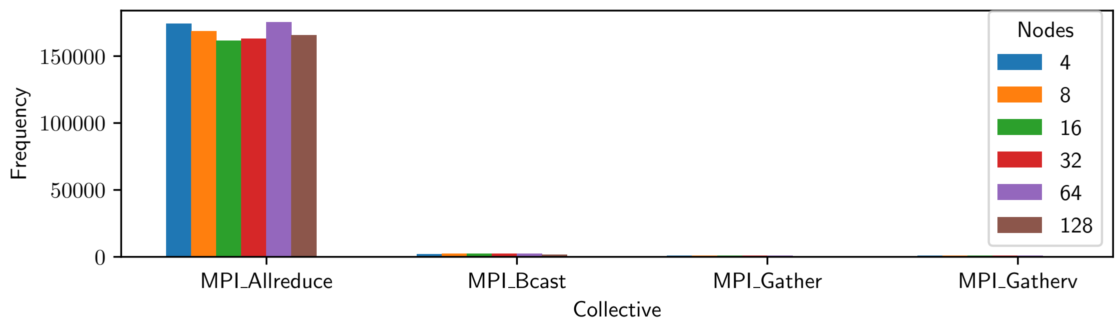
(d) `DataParallelASGD`

Figure 3.1: Frequency of MPI calls used in different configurations of CNTK for ResNet-20

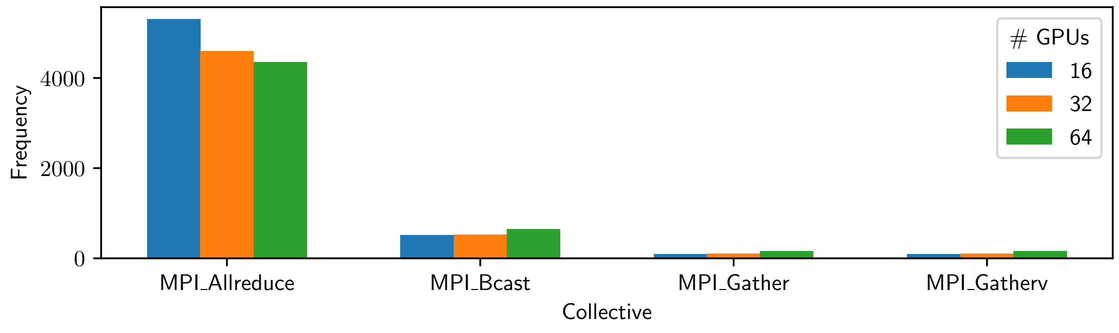
We noticed that `DataParallelASGD` seldomly uses collectives and relies heavily on point-to-point communication; almost 10x more than other methods used by CNTK. All of CNTK's training algorithms use non-blocking sends and a mixture of blocking and non-blocking receives apart from `DataParallelSGD` which only uses non-blocking. As expected, increasing the number of GPUs results in an increase in the frequency of point-to-point communication. The frequency of `MPI_Irecv` seems to be constant but the increase in frequency of point-to-points stem from an increase in `MPI_Recv`. The frequency of `MPI_Recv` is highest in `DataParallelASGD` tests.

Unlike CNTK, Horovod does not use point-to-point communication at the application layer and relies heavily on collectives. We looked at both the CPU and GPU versions of Horovod in Figures 3.2 and 3.3. When looking at all configurations of Horovod we saw similar collective usage between CPU and GPU versions of the framework, only the NCCL configurations differ. Regardless of scale and platform, `MPI_Allreduce` has the highest frequency, with `MPI_Bcast` in second place. A total of five MPI collectives were present but `MPI_Allgather` has been omitted from the figure. Like CNTK, Horovod called `MPI_Allgather` exactly once per process for all configurations. Again, this suggests that it is used for initialisation purposes. Our profiling shows that `MPI_Bcast` is used at start-up but `MPI_Allreduce` is used for the majority of training. This is expected as `MPI_Bcast` is used to initialise the global model state and that Horovod uses the All-Reduce method of averaging gradients. `MPI_Allreduce` is used significantly more in the CPU version than the GPU although the workload per processes is fixed. We suspect that this is due to the packing of the fusion buffer and that there is a 10x longer run-time for the benchmark on CPUs.

For our NCCL profiling we did not repeat the tests for PyTorch as we saw minimal difference in frequency between PyTorch and TensorFlow. In these two configurations the tensor reductions are offloaded to NCCL.

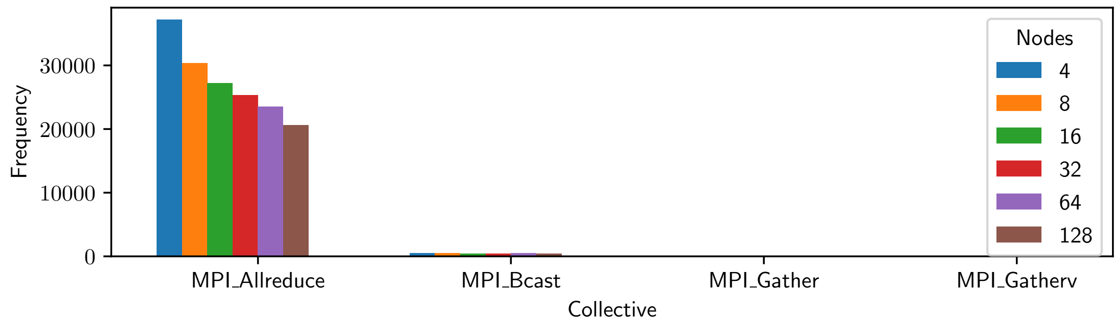


(a) TensorFlow CPU

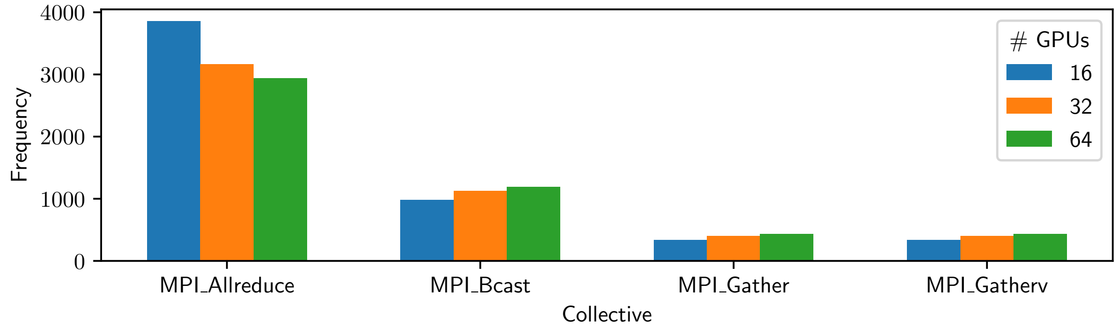


(b) TensorFlow GPU

Figure 3.2: Frequency of MPI collectives used in CPU and GPU configurations of Horovod + TensorFlow for ResNet-50

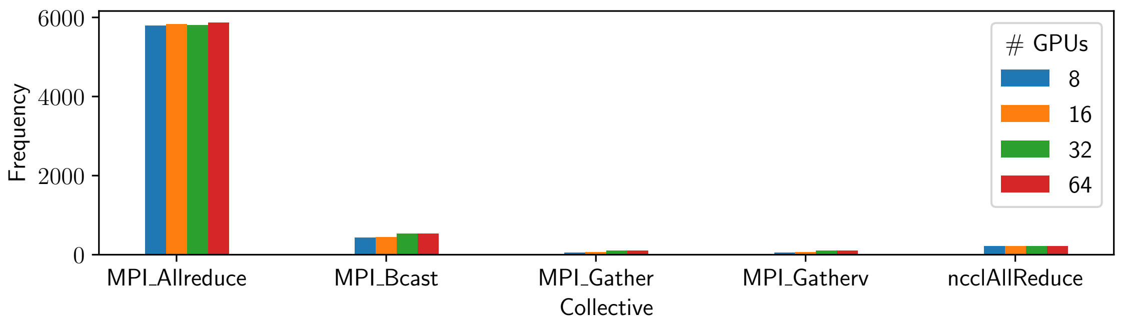


(a) PyTorch CPU

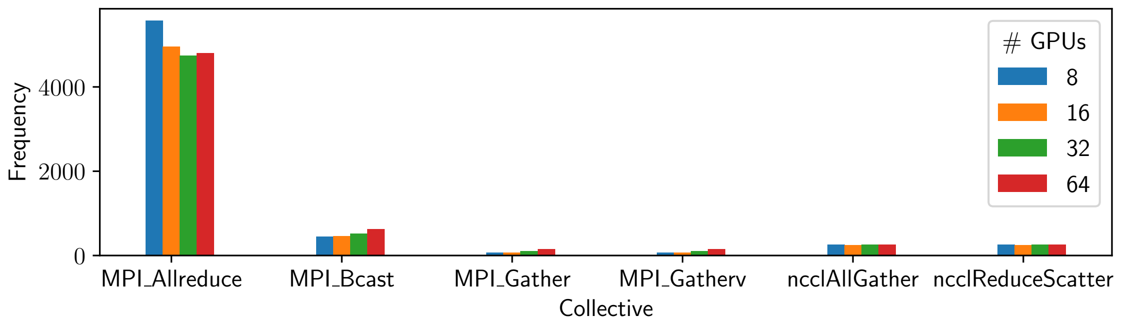


(b) PyTorch GPU

Figure 3.3: Frequency of MPI collectives used in the CPU and GPU configurations of Horovod + PyTorch for ResNet-50



(a) TensorFlow GPU (NCCL)



(b) TensorFlow GPU (NCCL Hierarchical)

Figure 3.4: Frequency of MPI and NCCL collectives used in the NCCL configurations of Horovod + TensorFlow for ResNet-50

When looking at the NCCL configurations we can enable and disable the environment variable `HOROVOD_HIERARCHICAL_ALLREDUCE`. For the flat allreduce (when the environment variable is set to 0) we see that only `ncclAllReduce` is used directly. In this scenario, all tensor reductions are computed using `ncclAllReduce`. For the hierarchical NCCL configuration, we see `ncclAllGather` and `ncclReduceScatter`. Here, Horovod initially uses an intra-node `ncclReduceScatter`, then an inter-node `MPI_Allreduce`, and finally an intra-node `ncclAllGather`.

When looking at Figure 3.4a we see that even though tensor reductions are isolated to NCCL we still have very high frequency of `MPI_Allreduce`. This could be the `MPI_Allreduce` bottleneck noted in [48].

### 3.3.3 Impact of MPI Communication on Application Time

Figure 3.5 shows the percentage of run-time spent in different collectives and point-to-point at 64 GPUs for CNTK. The time spent in the non-blocking point-to-point calls and `MPI_Wait` were measured. Less than 1% of application run-time was spent in two types of calls for all training methods. From our profiling method it was difficult to match an `MPI_Isend` to its corresponding `MPI_Wait` as multiple `MPI_Isend` may be made before `MPI_Wait` is called. Therefore, in Figure 3.5 we only see how much time was spent in a specific function call from the application's perspective. We can see that for all training methods around 8-21% of run-time is spent inside `MPI_Bcast`. `DataParallelSGD` differs from the other algorithms as it does not rely heavily on `MPI_Bcast` (≈8%) while it spends around 72% of run-time in `MPI_Allreduce`.

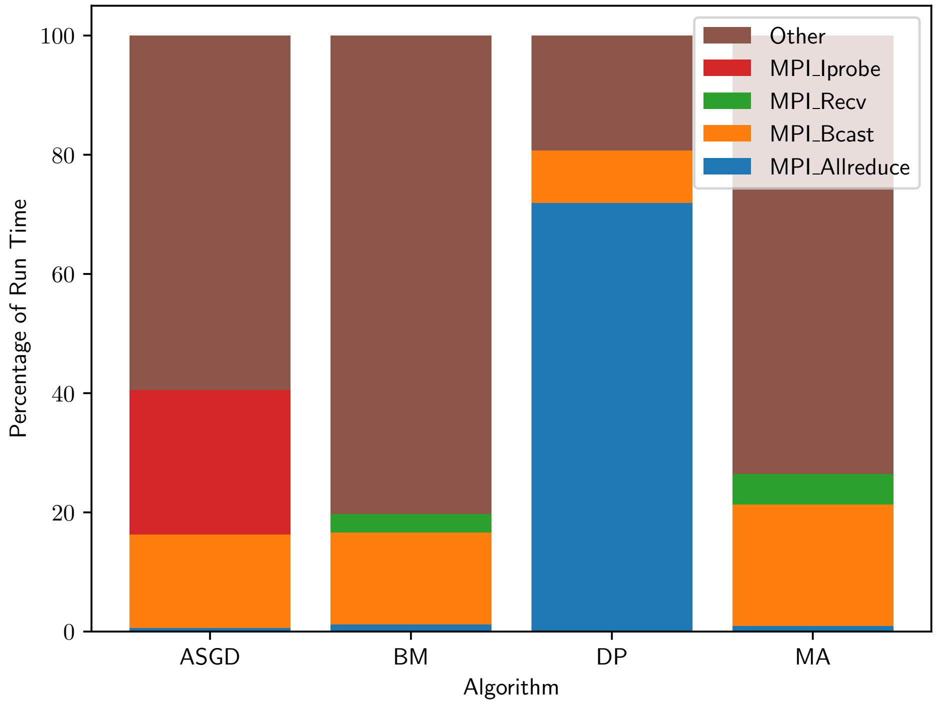
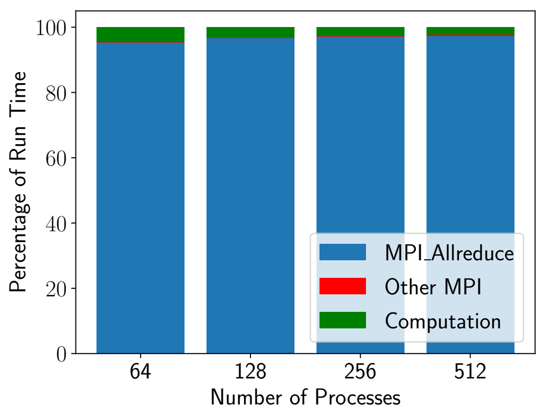
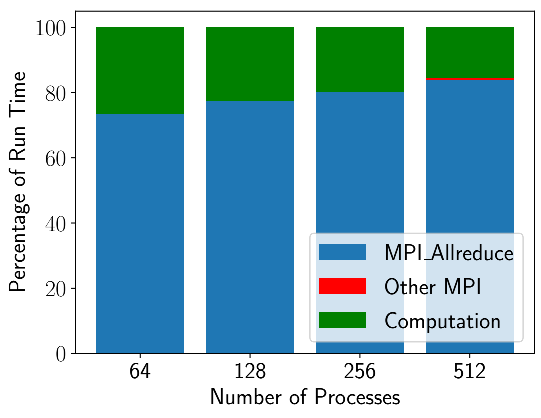


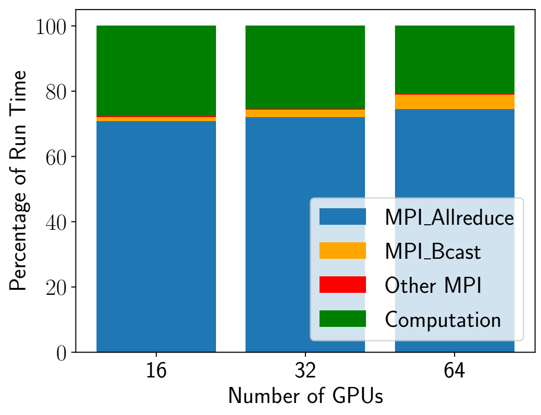
Figure 3.5: Percentage of Run-Time Spent in MPI for CNTK's Training Methods



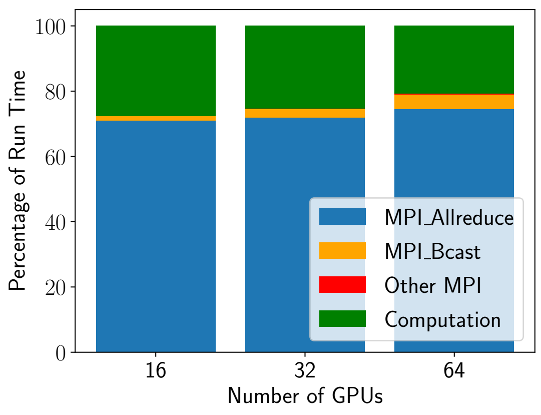
(a) PyTorch CPU



(b) TensorFlow CPU



(c) PyTorch GPU



(d) TensorFlow GPU

Figure 3.6: Impact of MPI Communication on Application Run-Time for Different Process Counts of Horovod using ResNet-50

The impact of MPI on run-time for the CPU and GPU versions of Horovod can be seen in Figure 3.6. For the most part, the time spent in these collectives correlate with their frequency. As the number of GPUs or CPU processes increase, so does the percentage of MPI communication. Generally, TensorFlow spends slightly more time in communication than PyTorch on GPUs but the inverse is true for CPUs. In terms of frequency `MPI_Gather` and `MPI_Gatherv` contribute a similar amount to `MPI_Bcast` but its impact on run-time is significantly less. On the CPU version of Horovod, less than 1% of run-time is spent in `MPI_Bcast`. The impact of collectives such as `MPI_Allgather`, `MPI_Gather`, and `MPI_Gatherv` are negligible. Regardless of scale or framework `MPI_Allreduce` has the largest impact on run-time. Up to

97% and 73% of runtime for CPU and GPU versions of these frameworks is spent in this one collective, respectively. For the GPU frameworks up to 6% is spent in `MPI_Bcast`.

As the default configuration of Horovod uses `MPI_THREAD_MULTIPLE`, it is possible for multiple threads to concurrently make MPI calls within a single process. At 64 GPUs a maximum of 2 MPI calls overlap and approximately 0.15% of all MPI calls overlap in their execution time. This suggests that Horovod uses at least 2 threads for communication. It is important to note that when looking at Figure 3.6, although blocking MPI collectives can take up to 97% of an application's run-time, it seems that Horovod provides good computation/communication overlap within a single process. Therefore, some computation could also occur in this time period.

In Table 3.1 we can see the percentage of run-time spent in ResNet50 for the GPU configurations of TensorFlow. Table 3.2 shows the associated throughput measurements. Entries have been left blank if the collective was not used. 0.00 denotes that a collective has been used but its impact on run-time was not measurable at two decimal points. The usage of `MPI_Allgather`, `MPI_Bcast`, `MPI_Gather`, and `MPI_Gatherv` are fairly consistent between the different configurations as these calls are not offloaded to NCCL. The largest difference is with `MPI_Allreduce` and the NCCL collectives. As expected, the NCCL versions spend less time in `MPI_Allreduce` than the MPI-only implementation. We observe in the flat NCCL implementation that the decrease in `MPI_Allreduce` communication results in the increased usage of `ncclAllReduce`. It appears that Horovod has simply offloaded the reductions using GPU buffers to NCCL. With the hierarchical NCCL we see that some of that `MPI_Allreduce` time is offloaded to `ncclAllGather` and `ncclReduceScatter`. We know the hierarchical NCCL approach also uses `MPI_Allreduce` for inter-node communication.

From these results we can see the reduction in `MPI_Allreduce` needs to be further

Table 3.1: Percentage of Run-time Spent in Communication for ResNet50 on 64 GPUs using Horovod + TensorFlow

| **Collective** | **Percentage of Run time** | | |
|---|---|---|---|
| | **MPI** | **NCCL** | **NCCL (Hierarchical)** |
| MPI_Allgather | 0.00 | 0.00 | 0.01 |
| MPI_Allreduce | 87.10 | 80.94 | 75.64 |
| MPI_Bcast | 4.36 | 5.59 | 5.38 |
| MPI_Gather | 0.20 | 0.28 | 0.23 |
| MPI_Gatherv | 0.04 | 0.04 | 0.05 |
| ncclAllReduce | - | 2.35 | - |
| ncclAllGather | - | - | 0.01 |
| ncclReduceScatter | - | - | 0.01 |

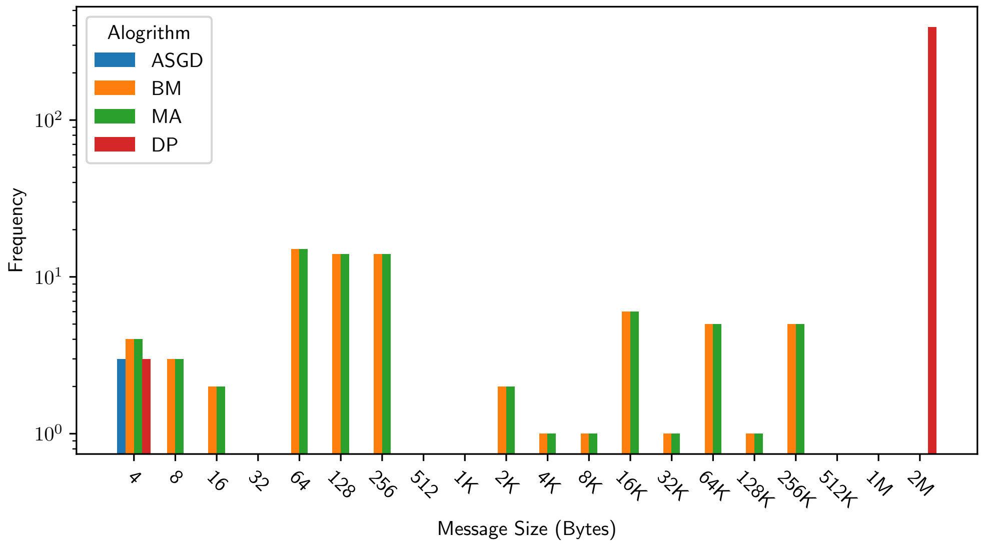
Table 3.2: Throughput for ResNet50 on 64 GPUs using Horovod + TensorFlow

| **Library for Tensor Reductions** | **Throughput (Images/Sec)** |
|---|---|
| MPI | 7024.2 |
| NCCL | 12439.3 |
| NCCL (Hierarchical) | 12704.9 |

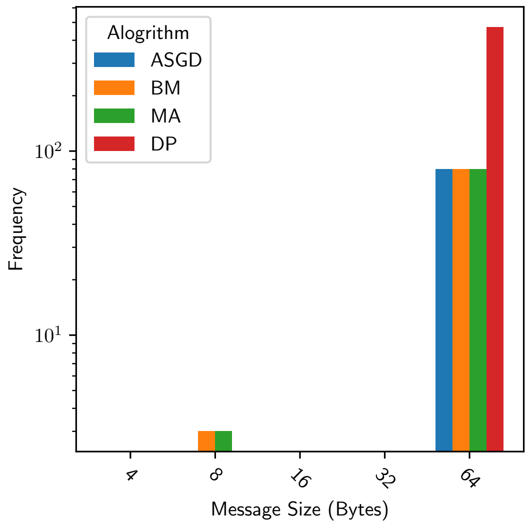
investigated to see if there are any changes in regards to message size when using NCCL. That said, we do see a correlation between the reduction in time spent in `MPI_Allreduce` with an increase in throughput. Our results shows that hierarchical NCCL performs better than the default NCCL at 64 GPUs, however it has been shown in [48] that the NCCL-only version performs significantly better at scale (27000 GPUs).

### 3.3.4 Message Sizes used by Collectives

Figures 3.7 to 3.10 show the message sizes used in CNTK and Horovod. Message sizes for a particular collective were binned on intervals of $(2^{n-1}, 2^n]$ bytes where $2^n$ is the message size displayed on the x-axis. When profiling CNTK, we were unable to find 'large' `MPI_Bcast` messages which were present in other works [12, 44, 45]. Our profiling shows that CNTK only uses messages in the 8-64B range for all of the distributed training methods we looked at. We think that this could have occurred



(a) `MPI_Allreduce`



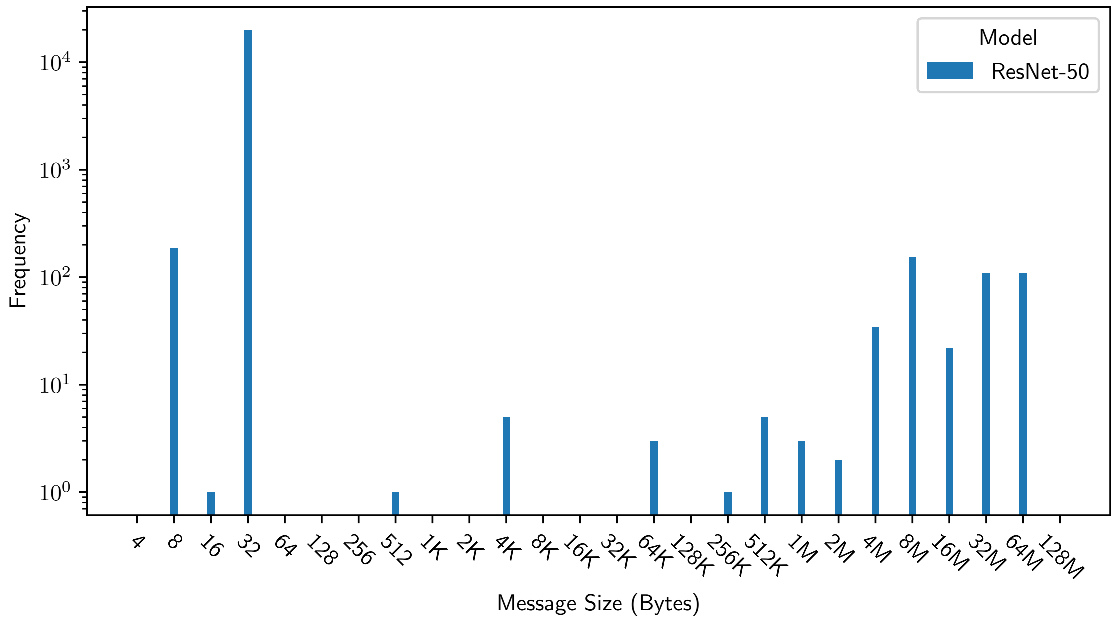
(b) `MPI_Bcast`

Figure 3.7: CNTK's Message sizes used in Different Distributed Training Algorithms for ResNet-20 with CIFAR-10

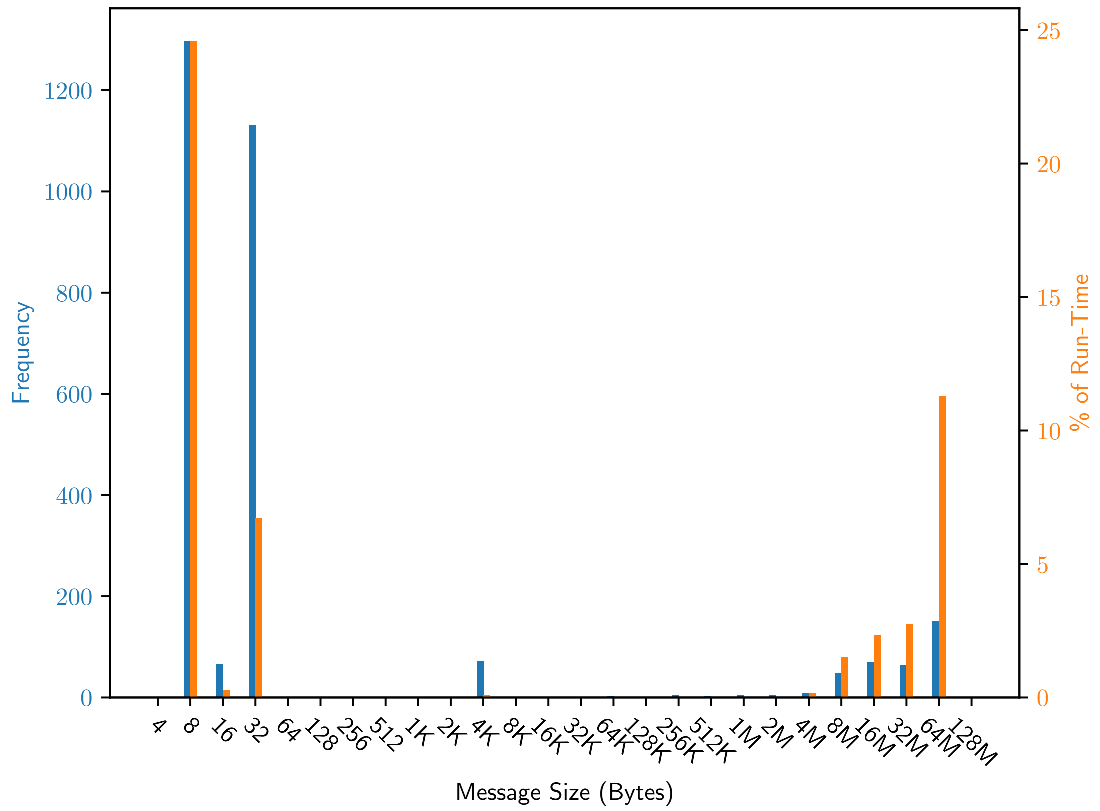
for a few reasons: 1) they have used a modified CNTK framework; 2) and their work was before CNTK v2.0 was released whereas we have used v2.6; and 3) they could have been using different distributed training method that we have not studied in this thesis. We do not think our results are incorrect as the `DataParallelSGD` method was verified with the artefacts of [14] which were available on GitHub. CNTK uses a wide variety of message sizes for `MPI_Allreduce` in the 4B to 256KB range for three of the algorithms we studied. The `DataParallelSGD` differs as there are many in the (1MB, 2MB] range and at 4B, but not elsewhere. When looking at the impact that these training methods have on run-time, in Figure 3.5, we see that `DataParallelSGD` has spent significantly more time in `MPI_Allreduce`. This is reflected in the nearly 100x increase of the 2MB messages compared to other sizes.

For brevity we have not presented plots of the message sizes used by `MPI_Bcast` in Horovod, but we found that the most frequently used message size is 4B. We observed a maximum of 6% of run-time is spent inside `MPI_Bcast` as seen in Figure 3.6c and 3.6d. `MPI_Bcast` is mostly used to initialise global variable inside the training script. This includes the model and optimiser state. It is likely that improving `MPI_Bcast` would have a smaller impact on performance than `MPI_Allreduce`.

For both platforms and configurations of Horovod, there exists a peak in the 8-32B message size range for `MPI_Allreduce`, as seen in Figure 3.9 and Figure 3.8. Messages in this range nearly always used `MPI_BAND` for the `MPI_Op` parameter. The main difference between messages in this range is the count parameter. We found these small messages to be allocated on host memory. We believe the small messages here are Horovod's synchronisation mechanism involving the `MPI_BAND` operation [48]. The environment variable `HOROVOD_CYCLE_TIME` controls the frequency of these messages. Keeping the size of the fusion buffer constant and increasing the cycle time resulted in a decrease in the frequency of the 8B messages. We were able to reduce the frequency of this message 30x on Béluga but but we saw minimal improvement in application performance. It was also shown in [47] that increasing the cycle time often results in a degradation in performance. Since this message is a control message, to synchronise workers, we speculate that it is sent in a separate thread to the main computation. Thus, it may not dramatically impact performance at this scale.

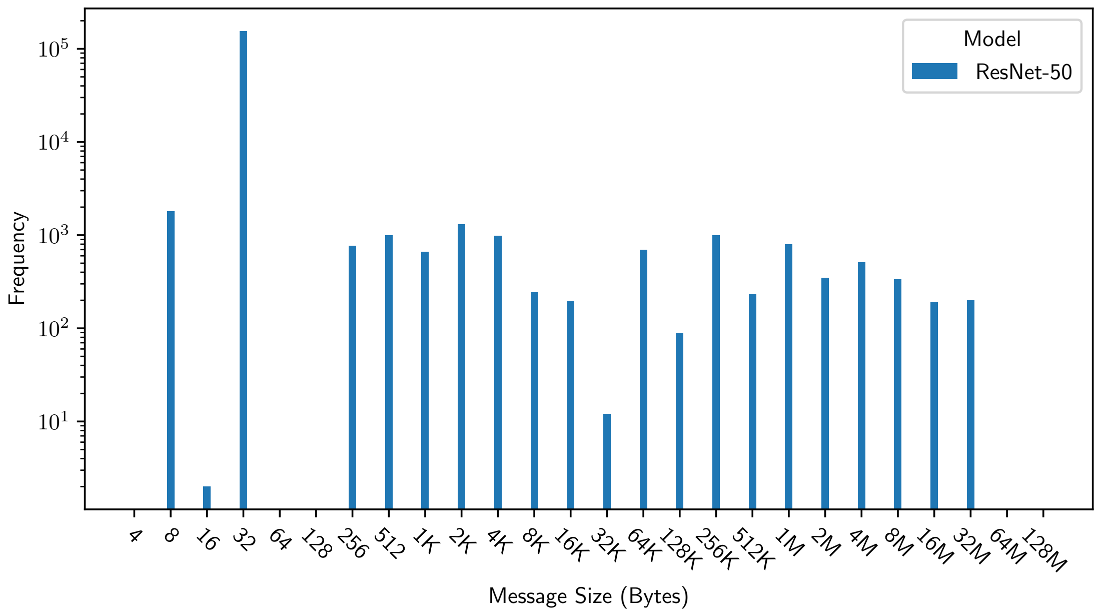


(a) `MPI_Allreduce` (CPU)

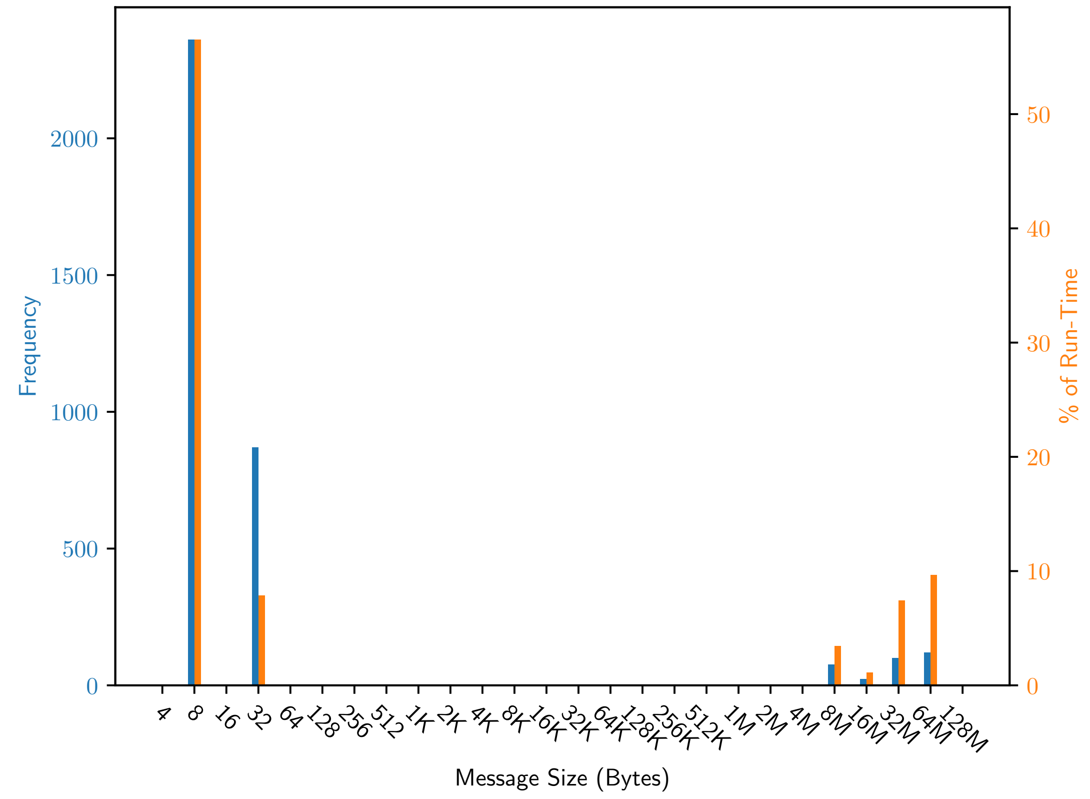


(b) `MPI_Allreduce` (GPU)

Figure 3.8: Message sizes used by different collectives in PyTorch for CPU and GPU configurations



(a) `MPI_Allreduce` (CPU)



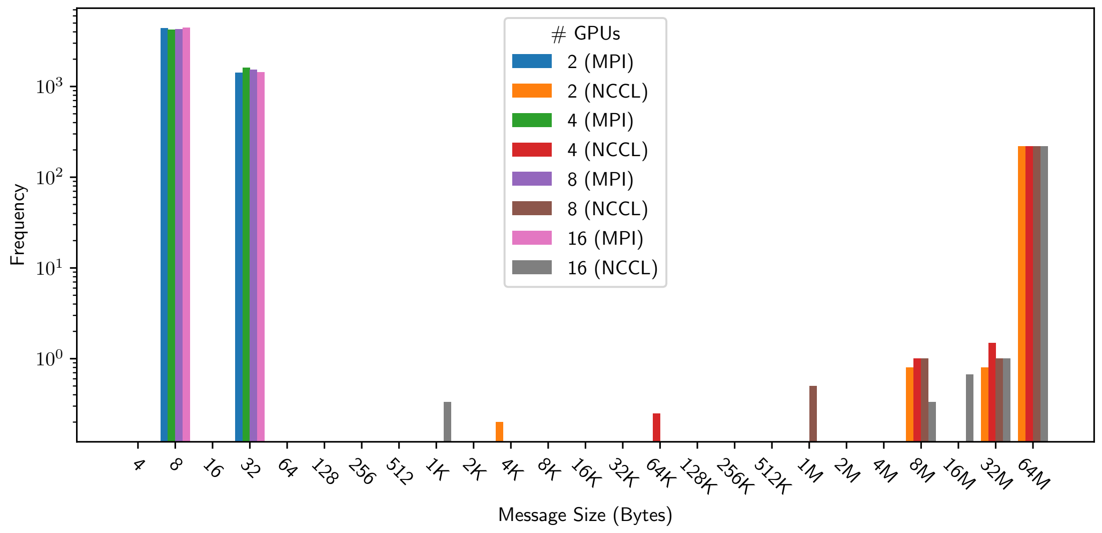
(b) `MPI_Allreduce` (GPU)

Figure 3.9: Message sizes used by different collectives in TensorFlow for CPU and GPU configurations

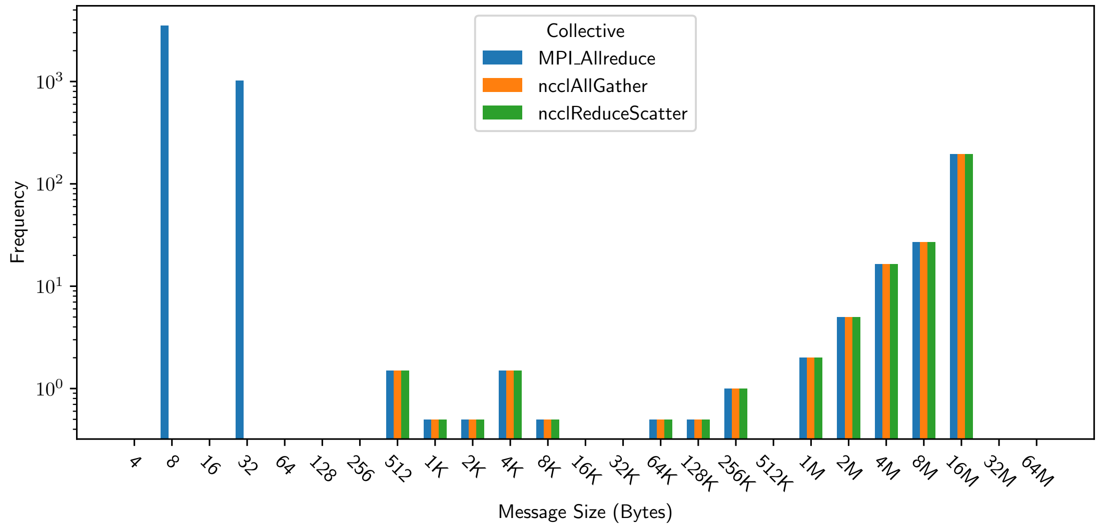
The maximum message size we see for `MPI_Allreduce` on both frameworks corresponds to the default value of the `HOROVOD_FUSION_THRESHOLD`. Although the fusion buffer can be disabled [13] and varied in size to improve performance, we chose to keep it constant for our tests. We see that the CPU version has a more uniform distribution of message sizes for `MPI_Allreduce` compared to the GPU versions. The GPU frameworks cluster towards the upper limit of 64MB. When measuring the end-to-end performance of these benchmarks CPUs have a lower throughput than GPUs. We think that the fusion buffer is less packed before the reduction operation as the cycle time (3.5ms) is fixed between tests. This results in many smaller messages than a few large messages closer to the fusion buffer size.

Although more time is spent in the 8B message for `MPI_Allreduce`, we were able to dramatically decrease the total communication frequency. However, we were not able to improve overall application performance. Since Horovod spends approximately 20-30% of run-time in the 8MB-64MB range for `MPI_Allreduce`, improving latency at this range could yield some application performance improvement. We see in Table 3.1 and Table 3.2 that offloading this workload to NCCL does improve throughput.

In Figure 3.10a we present the message sizes used by the configuration which uses only NCCL for tensor reductions. As the process count is increased very little changes. We see a distinct split between message sizes and how they are sent between MPI and NCCL. For messages less than 32B, `MPI_Allreduce` is used. For large messages in the 1KB to 64MB range, `ncclAllReduce` is used. In Figure 3.10b we see the message sizes used by the hierarchical NCCL implementation of Horovod. Only the frequency of training with 64 GPUs have been shown here for clarity of the figure. We see consistent frequency of `MPI_Allreduce`, `ncclAllGather`, and `ncclReduceScatter` for large messages. This shows the hierarchical algorithm in action. Messages do not reach the maximum buffer size.



(a) NCCL



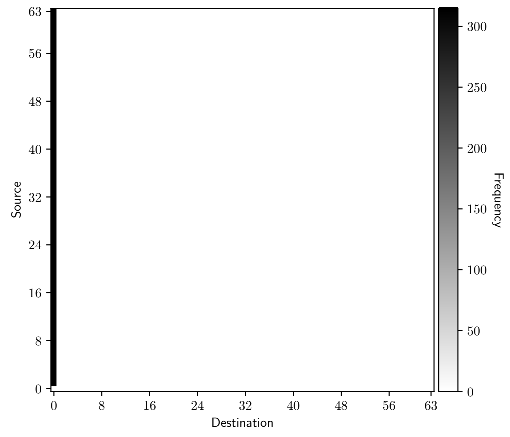
(b) Hierarchical NCCL

Figure 3.10: Message sizes used by AllReduce for different NCCL configuration for TensorFlow training ResNet50

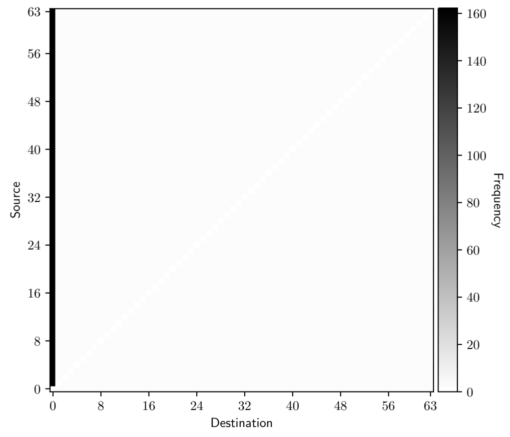
Changing the `HOROVOD_FUSION_THRESHOLD` would allow messages to reach 64MB but we have not modified the parameter to keep things consistent between different communication back-ends. Again, there is a peak for `MPI_Allreduce` in the 8-32B range. The peak occurs regardless of the configuration of Horovod we use. This reinforces the claim that these 8B messages are not for tensor computation. Also we see that improving the latency of large messages (i.e., offloading them to NCCL) results in a significant performance improvement. Therefore, large messages on GPUs are important.

### 3.3.5 Point-To-Point Communication Pattern

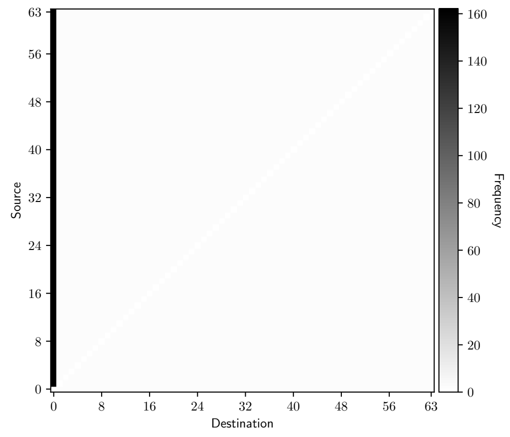
Figure 3.11 shows the source/destination pairs for all point-to-point communication at the application layer for the different configurations of CNTK that we studied. All methods apart from `DataParallelSGD` communicate with each other at least once. This is not very visible in Figure 3.11b and Figure 3.11c as only around 2-4 messages per pair of processes are sent to non-zero ranks. In Figure 3.11d the



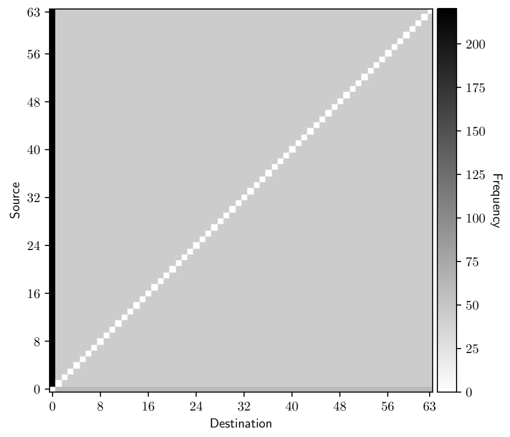
(a) `DataParallelSGD`



(b) `BlockMomentumSGD`



(c) `ModelAveragingSGD`



(d) `DataParallelASGD`

Figure 3.11: Communication Partners of Point-To-Point Calls in different configurations of CNTK for ResNet-20

fully connected communication is more visible. This is mainly due to the increased point-to-point communication we see with `DataParallelASGD` in Figure 3.1d.

Regardless of which algorithm we look at, the most communication occurs by all processes communicating with process one. Traditional HPC applications such as the Fire Dynamics Simulator (FDS) exhibit long message queue traversals due to multiple processes communicating to a single process [42]. This is often a bottleneck in MPI applications designed in this manner. Therefore we studied this further by measuring the message queues.

### 3.3.6 MPI Message Queues

Using the `MPI_T` interface we measured the performance variables (pvars) labelled as `time_matching_unexpectedq` and `time_failed_matching_postedq` which measure the total time and the number of searches in the UMQ and PRQ respectively. Very little time was spent in both of the message queues for all of the algorithms apart from `DataParallelASGD` configuration. Here a significant amount of time is spent

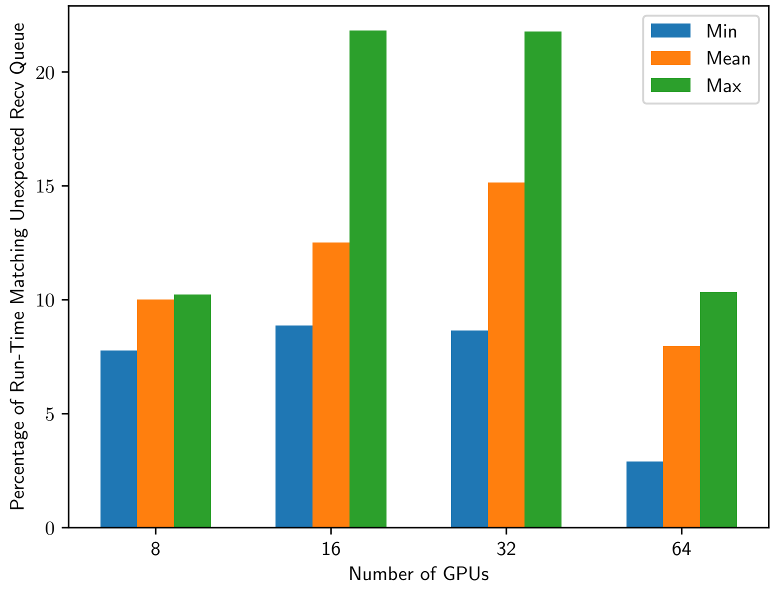


Figure 3.12: Percentage of Run-Time Spent Matching Unexpected Recv Queue for ResNet-20 using `DataParallelASGD`

in the `time_matching_unexpectedq` pvar. The time spent in this pvar is plotted in Figure 3.12. Measuring this pvar varied greatly between runs, therefore minimum, mean, and maximum are shown. We see that up to 22% run-time is spent in this pvar. To determine if message matching is an issue with CNTK we directly profiled the message queue functions to determine the issue. Our results showed that this large amount of time spent in the message queue was due to `MPI_Iprobe`. We can see the impact of `MPI_Iprobe` on run-time in Figure 3.5. `MPI_Iprobe` was frequently called but usually the message queue was empty. This suggested that traditional HPC message matching techniques could not be applied here and this would be better resolved in the application layer.

## 3.4 Summary

Similar to traditional HPC applications, Deep Learning applications use a mixture of point-to-point and collectives. `MPI_Allreduce` and `MPI_Bcast` continue to be the dominant collectives for MPI applications at scale. The majority of application run-time is spent inside `MPI_Allreduce`.

The introduction of NCCL to MPI applications poses new challenges as communication is split between these two libraries. We saw that communication can be offloaded from `MPI_Allreduce` to `ncclAllReduce` with some performance gain. When investigating how performance improvements can be made to these Deep Learning frameworks, it is important to note how messages are split between these libraries at run-time, whether it is by message size or by splitting between intra and inter-node communication.

For both MPI and NCCL the message sizes we have seen are much larger than what we have seen with traditional HPC workloads. This reflects an increased bandwidth we see with new interconnects such as NVLink. We saw two message ranges which should be focused on for `MPI_Allreduce`; 8-32B and 2MB-64MB.

Although we measured that some time is spent in polling the message queues of CNTK, the application was checking an empty message queue rather than traversing a long message queue. As long message queues are not present, we will not investigate message matching for Deep Learning. We will continue working with these Deep Learning applications but most of our focus will be on `MPI_Allreduce` for large message sizes as we think this is the best direction to move in to improve these workloads.

# Chapter 4

# Multi-Path Point-to-Point GPU Communication

GPU communication is in the critical path of many HPC and Deep Learning applications. In Chapter 3, we saw that GPU-based MPI communication is a major bottleneck for Horovod and CNTK. These are not the only applications which exhibit the usage of GPU communication but these are examples of applications which are frequently used by parallel and distributed Deep Learning researchers. Horovod and CNTK use a mixture of point-to-point and collective communication. For this chapter we will be focusing on GPU versions of the frameworks profiled in Chapter 3. Not every communication call uses GPU buffers but a large portion do.

MPI's collective communication is implemented directly upon point-to-point communication. Therefore, improving point-to-point communication *should* also benefit collectives. Tackling GPU-based point-to-point communication will allow us to have largest impact on the MPI library. GPU point-to-point communication transfers data residing on GPU global memory between two processes. The two data regions can be on the same or different GPUs. The data transfer can be direct between the two regions or they can be staged in host or NIC memory.

In this chapter we plan to investigate data transfers between two GPUs while using multiple different copy mechanisms. Although some work in this area exists for collective communication [49], to our knowledge, none apply this to point-to-point communication for a single data transfer. In this chapter we make the following contributions [16]:

- We propose a multi-path copy mechanism for UCX Put operations using simultaneous data transfers via host memory and CUDA IPC. We study the optimal number of CUDA streams and the ratio of data sent via each path to see how they impact performance.
- The design was tested with micro-benchmarks at the UCX Put layer and at the MPI point-to-point layer, and significant performance improvement was observed at each layer.
- The design was evaluated on two platforms, one with PCIe to the host and NVLinks connecting the GPUs, and the other with NVLink for both paths.

## 4.1 Related Work

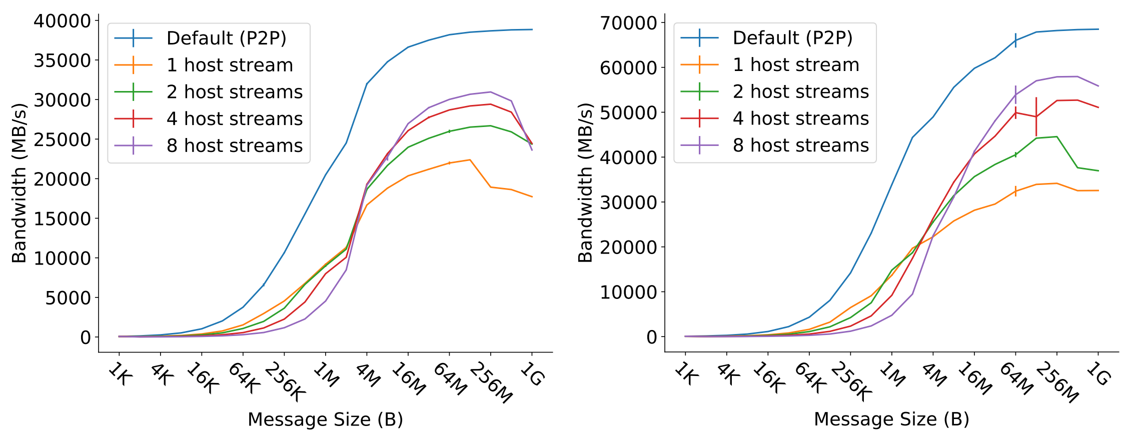
With the emergence of modern high-bandwidth interconnects such as NVLink, research has focused on their impact on communication performance. In [50], Pearson et al. evaluated the characteristics of CUDA communication primitives on high-bandwidth interconnects to understand memory transfer behaviour across different memory regions. Tallent et al. presented the impact of NVLink and PCIe interconnects on Deep Learning workloads [51]. In [52], Li et al. evaluated such interconnects with a multi-GPU benchmark suite.

Proposals to integrate compute accelerators within the context of MPI has existed for a long time [53]. In recent years, workloads have began to rely much more on accelerators such as GPUs. Nvidia introduced Inter-Process Communication (IPC) with CUDA 4.1. This is a fairly old feature of the CUDA Runtime library but today it is still one of the prominent methods of transferring data between GPU buffers allocated in different address spaces within the same compute node. In [27], Potluri et al. explored using CUDA IPC features within MVAPICH2-GDR. They saw significant performance improvement using CUDA IPC when compared to transferring data between GPUs via host memory. They presented up to 74% improvement in latency for 4MB messages. Today many MPI libraries provide CUDA-Aware MPI communication and they often use these CUDA IPC features to allow for good performance on GPU workloads.

CUDA IPC is not the only communication mechanism in which data can be transferred between GPUs, host memory can also be utilised. Faraji and Afsahi explored using a combination of CUDA IPC copies with host stage copies in [49]. They were able to show that using multiple communication channels allow for the acceleration of MPI collectives. In [54], Chu et al. presented that using a combination of host memory with GPU global memory to accelerate `MPI_Allreduce` and Deep Learning workloads. As such, using a combination of host and GPU memory has mostly been focused on improving GPU collectives. We would like to understand if such mechanisms can be effectively designed to point-to-point communication performance. To the best of our knowledge, studying point-to-point communication within this context has yet to be investigated. In this chapter, we explore weather using a mixture of CUDA IPC copies and host-staged copies can accelerate GPU point-to-point communication.

## 4.2 Motivation

To start our study we first investigated the potential of using a mixture of host memory based data transfers with peer to peer copies using CUDA IPC. This would allow us to see what the maximum performance we could gain when using these



(a) Inter-Socket (b) Intra-Socket

Figure 4.1: UCX Put bandwidth measurement taken using `ucx_perftest`. Using peer copies is compared to using host staged copies with a different number of streams. Results from Mist are shown.

two communication channels simultaneously.

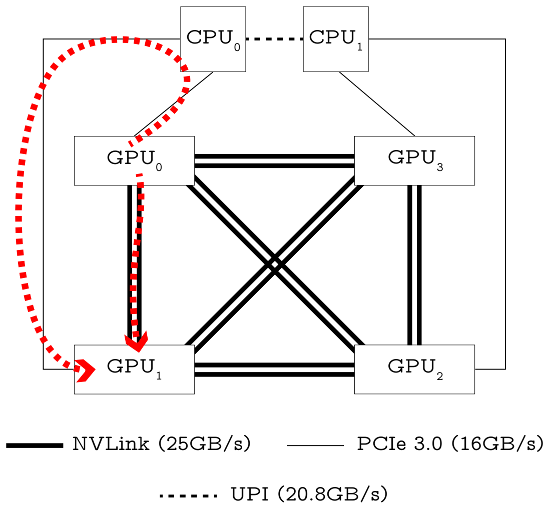
We first investigated the bandwidth we could achieve when transferring the data through the host alone. In these results, when copying data from $GPU_0$ to $GPU_1$, we copied data from $GPU_0$ to pinned host memory and then from host memory to $GPU_1$. We split the data into chunks when executing the device-to-host (D2H) and host-to-device (H2D) copies. This was based off of the number of streams which we used. For example, when using four streams we partitioned the buffer into 4 chunks and assigned one stream per chunk. In Figure 4.1 we can see that varying the number of streams modifies observable bandwidth. Generally, larger messages benefit from more streams and small messages perform better with a single stream.

Regardless of the number of streams that we use, we are not able to reach the bandwidth achieved by a device-to-device (D2D) copy. On Mist, shown in Figure 4.2, we have six NVLinks to the CPU and in theory we should be able to achieve 75GB/s for intra-socket copies. We speculate that the reason as to why we do not achieve this is due to the delay induced by storing data in host memory before passing it into the destination GPU. Using multiple streams allowed us to minimise this delay for large messages. This is because we can overlap the D2H and H2D copies between streams.

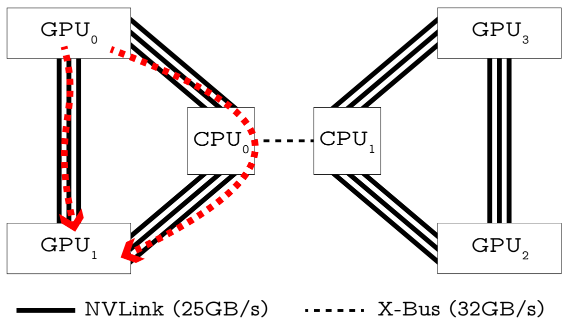
From this study it is clear that we cannot use host-stage copies alone for optimal bandwidth usage. We will need to combine this with the existing D2D copy mechanism used by UCX. Therefore, from these results we expect to achieve a peak bandwidth of 130GB/s once fully implemented on Mist.

## 4.3 Design and Implementation

The goal of our design is to send messages through multiple communication paths which may be underutilised. We plan to split the send buffer and transfer the data via two independent data transfer paths. In Figure 4.2, we can see the paths in which we plan to send messages on Cedar and Mist. For a point-to-point communication when sending data from $GPU_0$ to $GPU_1$ data would usually be transferred directly between the two GPUs via the NVLink. On Cedar we will send one part of the data via PCIe and another via NVLink. On Mist we will only use NVLink channels as we have NVLink directly connected to the CPUs.



(a) Cedar



(b) Mist

Figure 4.2: Single node physical typologies of Compute Canada systems. Uni-directional bandwidth is labelled for each system. The path of the multi-path copies are shown in red.

### 4.3.1 Host Staged Data Transfers

As we intend to send a portion of the data via the links connecting to the CPU, we must stage the data in host memory. To our knowledge there is no mechanism to transfer data between GPUs via the links connected to the CPU without staging in host memory. Therefore, we must implement this ourselves. We used the CUDA Driver API call `cuMemAllocHost()` to allocate a region of host memory that is page-locked and accessible to the GPU. Using this API, instead of the standard `malloc()` which we have in the C standard library, allows the Nvidia GPU driver to track the virtual memory ranges allocated and automatically accelerates `cuMemcpy()` operations. Using pined memory allows for read and writes with a much higher bandwidth as the memory will never be paged by the operating system.

When sending data via the host we plan to further split the portion of data that is going to be sent via this channel. By splitting the data it will allow for pipelining between D2H and H2D copies. We will place each `cuMemcpyAsync()` for each chunk of data on its own CUDA stream to implement this overlap. The number of chunks which we further subdivide the data is different for each message size. During the implementation we tested 1-8 chunks. The impact of the different number of subdivisions can be seen in Figure 4.1. Then we chose the optimal number of chunks to use for each message size in the form of a static tuning table.

Although this chunking idea is simple we had to add a few extra lines of code for the data transfers to behave as expected. We use the CUDA Runtime API call `cudaSetDevice()` before creating the streams to correctly associate stream with the desired device. We also call `cudafree(0)` as the CUDA Runtime API uses lazy initialisation to create the GPU context. Without this call the streams would not be created correctly. `cudafree(0)` was chosen as we required a function call which does not modify anything in our environment. Finally when creating the stream we used the `CU_STREAM_NON_BLOCKING` flag to ensure that these streams would not

```
cudaSetDevice(remote_GPU);
result = cuIpcOpenMemHandle(pointer, memory_handle
                                   CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS);
cudaSetDevice(local_GPU);
```

Listing 4.1: Opening CUDA IPC Memory Handle Pseudo-code

synchronise with the default CUDA stream.

The CUDA IPC code was already implemented within UCX but we also had to make a small change in how we open the IPC handle. We would have to set the device context to the destination of our PUT operation, open the IPC handle as normal, then finally set the device context back to original context. This is not required when transferring data directly between GPUs but we noticed that this was required to get our host staged copy to correctly pipeline. The pseudo-code can be seen in Listing 4.1.

### 4.3.2 Using Multiple Paths - Host Staged and Device To Device Copies

The final stage of our design is to combine the host staged data transfer described in Section 4.3.1 with the existing device to device copy implemented by UCX. Algorithm 4.1 shows how we can use the host staged copy in conjunction with the original device to device copy. Initially, we must calculate the percentage of data which we send via the host and directly to the other GPU, this is shown in Lines 1 to 3. The percentages sent over each path are predetermined from experiments where we varied how the message size was split across the two channels. We found the maximum aggregate bandwidth for each message size and stored it in another static tuning table.

Once we determine the size of data which we are sending to the host we then further subdivided across the streams. Then in parallel we copy the data from our source GPU to our destination GPU, from our source GPU to host memory,

**Algorithm 4.1** Multi-Path Copy Algorithm

```
Input: sbuf, host_buf, data_size, host_share, n_host_streams
Output: dbuf
1 host_dsize = data_size * host_share;  host_chunk_dsize = host_dsize /
    n_host_streams;  d2d_dsize = data_size - host_dsize;  do in parallel
2     Copy d2d_dsize bytes from sbuf to dbuf;
3     for $i \leftarrow 0$ to n_host_streams by 1 do in parallel
4         Copy host_chunk_size bytes from sbuf to host_buf[i];  Wait for data in
            host_buf[i];  Copy host_chunk_size bytes from host_buf[i] to dbuf;
5     end
6 end
```

and from host memory to our destination GPU as seen in Lines 5 to 9. When copying from host memory to our destination GPU we wait for the data to be stored in memory before executing the data transfer. Although not discussed in the algorithm, we offset the source, host, and destination pointers by the appropriate amount to ensure that the data is numerically consistent.

## 4.4 Performance Evaluation and Analysis

We evaluated our design on two HPC nodes, Cedar and Mist. Results were collected for UCX and Open MPI point-to-point communication using micro-benchmarks. micro-benchmarks are controlled environments and they do not accurately represent what would happen if the same algorithm was applied to an application. With micro-benchmarks all processes synchronise, messages are perfect power of 2 sizes, and it allows for caches to have the optimal values as for each iteration of the benchmark as we repeatedly use the same data values. Although these limitation of micro-benchmarks are true, they provide an upper bound on performance and scalability.

### 4.4.1 Experimental Setup

The Mist compute cluster is located at the SciNet HPC Consortium. Mist is an IBM POWER9 AC922 machine with two sockets for a total of 32 cores and 382GB of memory. Each node has four Nvidia V100 (32GB) GPUs per node, with three NVLinks between intra-socket GPUs and to the host processors, as shown in Figure 5.1. Mist uses the GNU/Linux distribution REHL 7.6. For our studies, we have used Open MPI 4.0.4rc2 with UCX 1.8.0, Spectrum-MPI 10.3.1, MVAPICH2-GDR 2.3.5.

The Cedar is a heterogeneous cluster located at Simon Fraser University. Cedar is a DELL C4140M with two sockets each with a 32-Core Intel Silver 4216 Cascade Lake CPU clocked at 2.1GHz. Each node has four Nvidia V100 (32GB) GPUs using the SMX2 package. The GPUs are fully-connected with two NVLinks between each GPU. These GPUs connect to the host using PCIe 3.0. Cedar uses the GNU/Linux distribution REHL 7.9 and for our studies we have used Open MPI 4.0.4rc2 with UCX 1.8.0.

### 4.4.2 UCX Put Results

As MPI point-to-point communication is implemented using UCX we first obtained results at the lowest software layer before scaling to MPI. The lowest layer would be the Put operation which send/recvs are implemented upon.

#### UCX Put Bandwidth Results

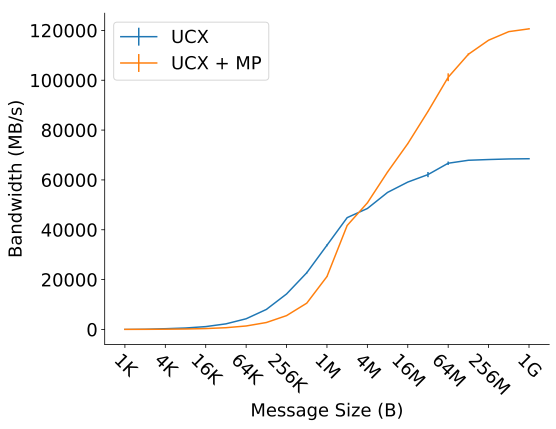
In this subsection we observe the results of UCX zero-copy Put operation for D2D transfers. In the previous section we saw that varying the number of streams had an impact on bandwidth measurements. Now that we are going to use a mixture of host-staged copies and D2D copies, the ratio in which we split the buffer will have an impact on bandwidth. We can see from Figure 4.1 that we achieve lower

Table 4.1: Example Tuning Table for Optimising UCX Put Bandwidth Results for Intra-Socket Multi-Path Transfers. For brevity only messages from 1M-1G is shown. All values were obtained but we only see performance improvement around the 1M mark.

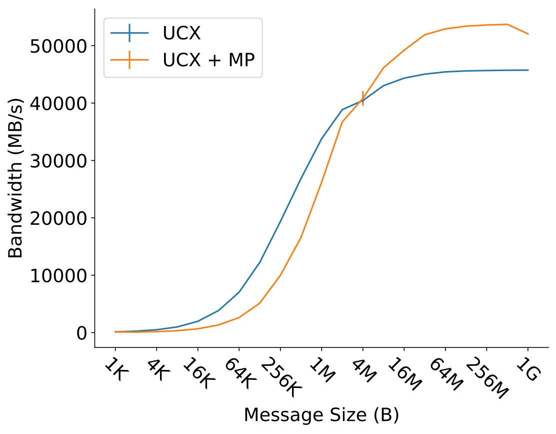
| **Message Size (B)** | **Percentage of Data Sent to Host** | **Number of Host Streams** |
|---|---|---|
| 1M | 25 | 1 |
| 2M | 25 | 1 |
| 4M | 25 | 1 |
| 8M | 25 | 1 |
| 16M | 25 | 1 |
| 32M | 30 | 2 |
| 64M | 30 | 2 |
| 128M | 30 | 3 |
| 256M | 30 | 6 |
| 512M | 30 | 6 |
| 1G | 30 | 6 |

bandwidth when using host-staged copies. Therefore, when partitioning the buffer we will send a smaller percentage of data via the host. For optimal performance in any of the benchmarks in this section, we will need to vary the percentage of data sent to the host and the number of streams used to copy to the host. Many tests were run and optimal values were found for each messages size. For each benchmark in this chapter we must generate a different static tuning table. An example tuning table is shown in Table 4.1. Within the UCX library this tuning table was placed and optimal values were used for each message size. A new tuning table must also be created for each benchmark in this chapter.

In Figure 4.3 and Figure 4.4 we observe the performance of the multi-path (MP) copy on both Cedar and Mist for intra and inter-socket data transfers. For intra-socket transfers on Mist we see peak bandwidth increase from around 72GB/s to 120GB/s (1.67x) when using this mechanism. We do not see a similar result for inter-socket bandwidth on Mist. This is due to two reasons. The X-Bus is the limiting factor when transferring data between sockets so our results are capped to

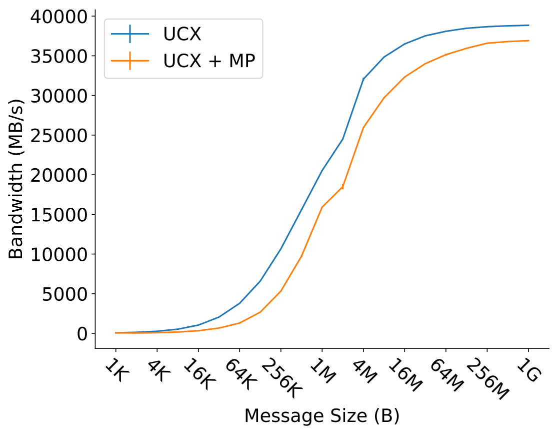


(a) Mist

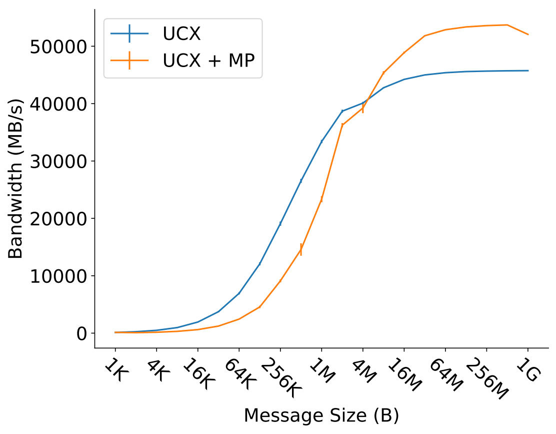


(b) Cedar

Figure 4.3: Intra-Socket UCX Put bandwidth measurement taken using `ucx_perftest`.



(a) Mist



(b) Cedar

Figure 4.4: Inter-Socket UCX Put bandwidth measurement taken using `ucx_perftest`.

64GB/s bandwidth. The second reason is that, although we can run the same code with the splitting of data and the use of multiple streams we do not see the same behaviour for inter-socket communication because the underlying hardware only has a single path. Thus the labels on Figure 4.4a state ‘multi-path’ but in reality it is not.

On Cedar we see some performance improvements for intra-socket and inter-socket transfers as we always have two paths which we can transfer data through.

The increased peak bandwidth we see on this platform is much smaller than Mist as we are using a mixture of PCIe and NVLink. We see a smaller performance increase from 46GB/s to 54GB/s (1.17x).

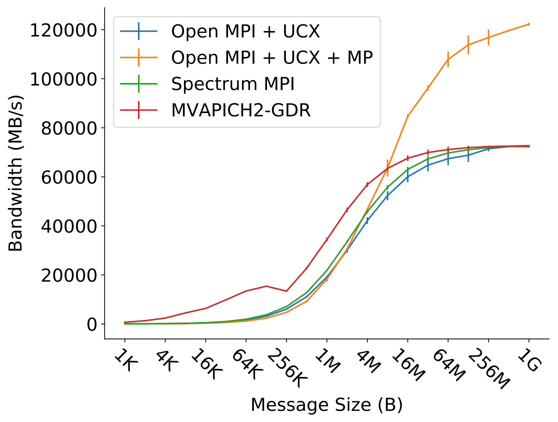
### 4.4.3 MPI Point-to-Point Results

After observing performance improvement for the UCX Put operation, the investigation was extended to MPI. For our MPI tests we are using the Ohio State University Micro-benchmarking Suite (OMB) [55] as the standard set of MPI tests used in our field. We have collected uni-directional and bi-directional bandwidth results. On Mist we have compared our results with MVAPICH2-GDR and Spectrum MPI. We did not do the same comparison on Cedar as Spectrum MPI is only for Open Power systems and cannot be used on Intel platforms. MVAPICH2-GDR was not used as the Cedar did not have the correct drivers or GDRCopy installed.

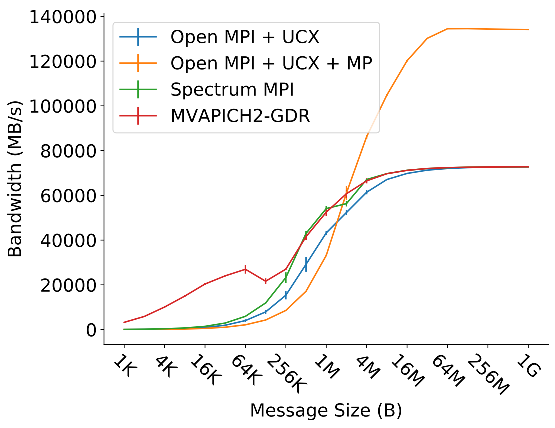
#### Uni-Directional Bandwidth Results

In Figure 4.5 and Figure 4.6 we present uni-directional bandwidth tests between two GPUs on the same socket for Mist and Cedar. On both platforms we see a good performance improvement in using this mechanism. In these figures we display bandwidth measurements for both a window size of one and 64. The default window size of `ucx_perftest` is one whereas with OMB it is 64. Changing OMB's window size allows us to directly correlate the performance we see with the Put operation with MPI point-to-point communication.

We see on both platforms, with a window size of one, very similar results to the `ucx_perftest`. This shows that we have a direct impact on Mist as we can achieve close to the 122GB/s (1.69x) peak bandwidth and 56GB/s (1.18x) on Cedar. When increasing the window size to 64 we obtain a slightly higher bandwidth of 59GB/s (1.23x) on Cedar and 134GB/s (1.84x) on Mist. This is because the increased

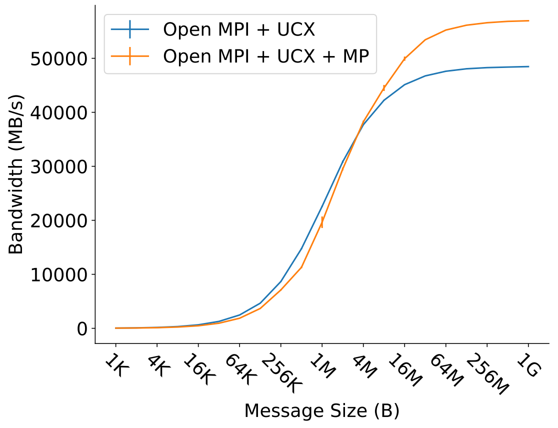


(a) Window Size 1

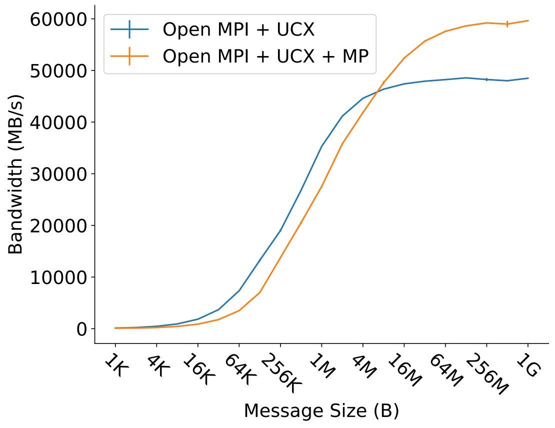


(b) Window Size 64

Figure 4.5: Intra-socket uni-directional bandwidth measurement taken using OMB on Mist



(a) Window Size 1

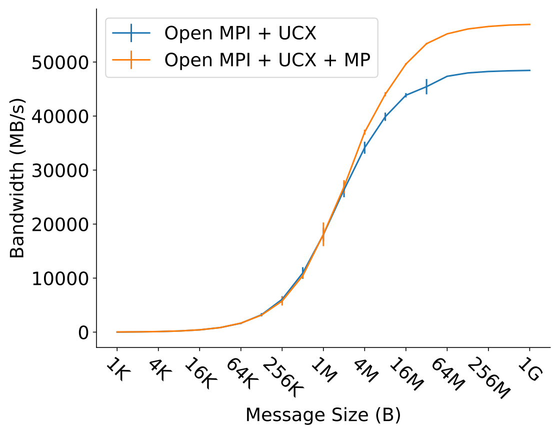


(b) Window Size 64

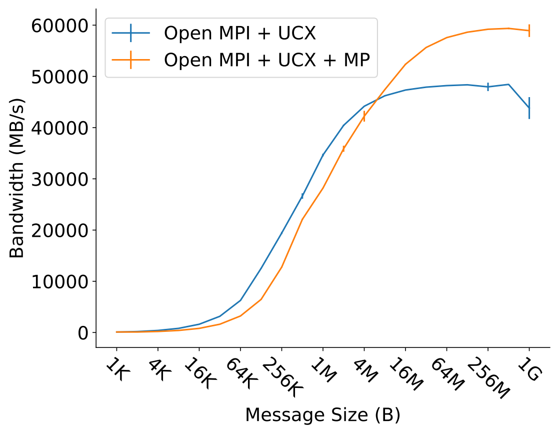
Figure 4.6: Intra-socket uni-directional bandwidth measurement taken using OMB on Cedar

window size allows us to better saturate the NVLinks.

In Figure 4.7 we see the same benchmarks but for inter-socket communication. Results for Mist have been omitted as we saw in Figure 4.4 that this multi-path copy does not improve performance for inter-socket transfer. On Cedar we see that we are able to gain some performance improvement by using this mechanism for inter-socket communication. This occurs because we have two paths to send messages: GPU to host via PCIe and across the QPI interconnect then back to the GPU and the other path would be using the NVLinks themselves. For a window size of 1, we



(a) Window Size 1



(b) Window Size 64

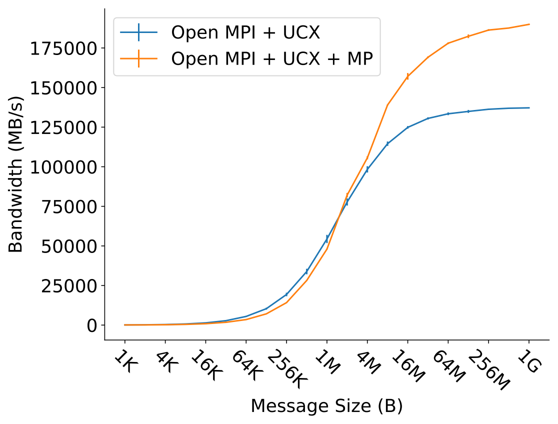
Figure 4.7: Inter-socket uni-directional bandwidth measurement taken using OMB on Cedar

see the same speedup of 1.18x to a peak bandwidth of 56GB/s. We also observe the same improvement as intra socket with a window size of 64: a speedup of 1.23x and a peak bandwidth of 59GB/s.

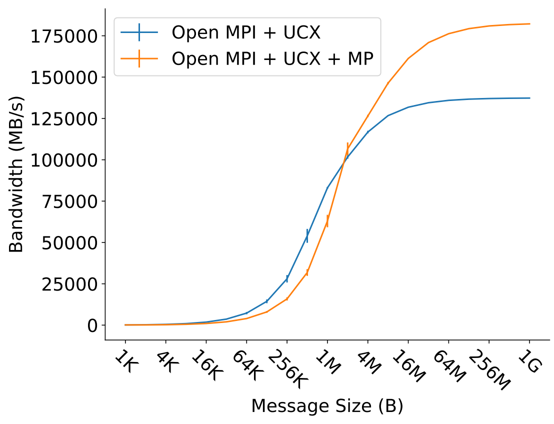
### Bi-Directional Bandwidth Results

As we saw performance improvements using the multi-path copy for uni-directional bandwidth tests, the study is extended to investigate bi-directional bandwidth tests. The reasoning for this is that our end goal is to apply this multi-path copy to collectives. Collectives frequently use bi-directional data transfers, therefore this may give some insight into the feasibility of using the mechanism in collectives.

In Figure 4.9, for both window sizes of 1 and 64, we see bandwidth increase from 92GB/s to 108Gb/s (1.18x) on Cedar for bi-directional bandwidth tests. We see a slight performance drop from 1.23x to 1.18x compared to the uni-directional tests. For Mist in Figure 4.8, we see a larger performance improvement than Cedar. This is expected due to the NVLinks to the Host. For a window size of 1 we get a peak bandwidth of 189GB/s (1.38x) and 182GB/s (1.33x) for a window size of 64. On Mist we see a large discrepancy between uni-directional and bi-directional bandwidth

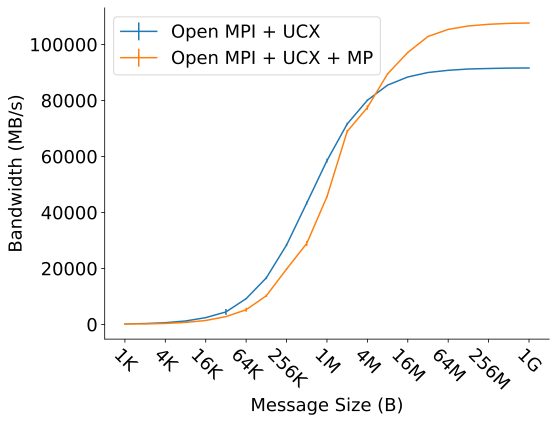


(a) Window Size 1

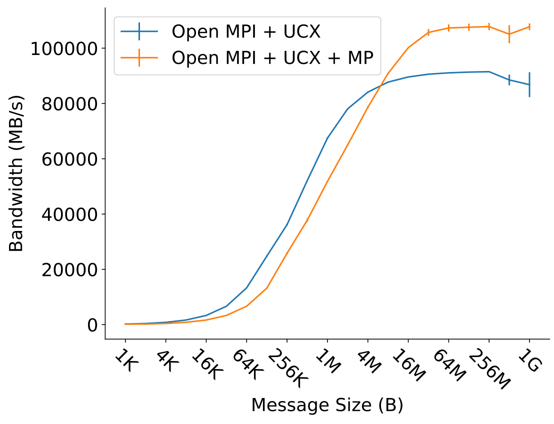


(b) Window Size 64

Figure 4.8: Intra-socket bi-directional bandwidth measurement taken using OMB on Mist



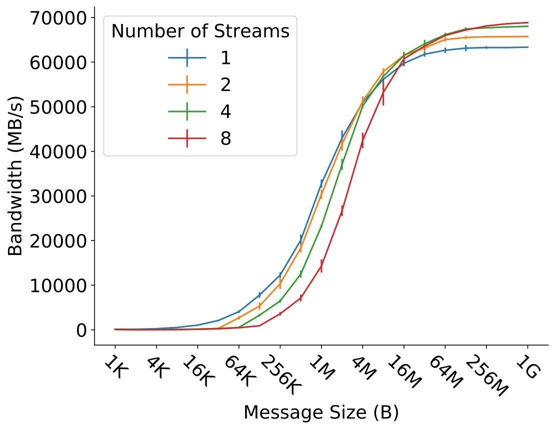
(a) Window Size 1



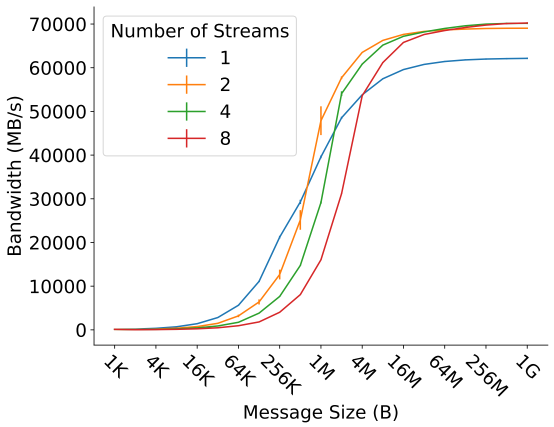
(b) Window Size 64

Figure 4.9: Intra-socket bi-directional bandwidth measurement taken using OMB on Cedar

tests. As we see up to 134GB/s for the uni-directional tests, we expected to achieve bandwidth of around 270GB/s for our bi-directional tests. The theoretical upper bound would have been 300GB/s assuming we have no delay for the host stage copy and that we utilised the full bandwidth of all channels. As we only got 189GB/s rather than 270GB/s, further tests were conducted to determine the reason behind this. To investigate this we conducted bi-directional bandwidth tests for the host stage copies only to see if there is a hardware limitation. We can see the results in Figure 4.10. Even with varying the number of streams, the maximum bandwidth

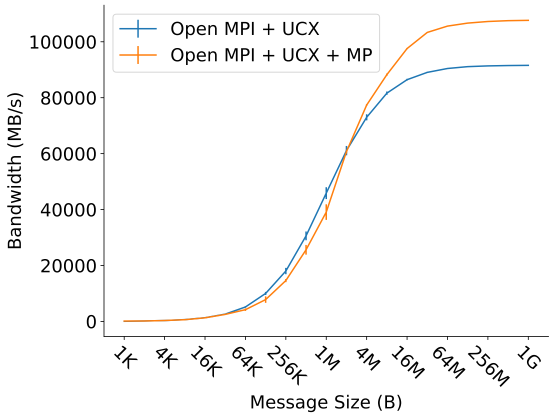


(a) Window Size 1

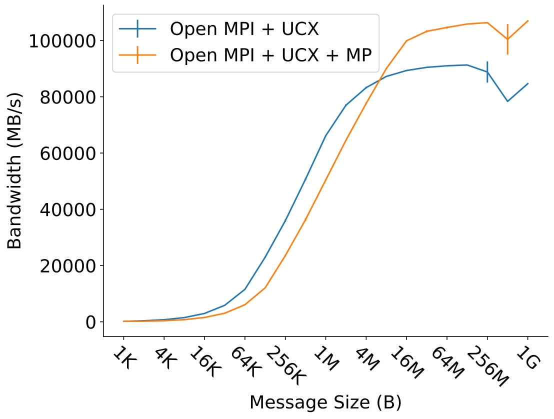


(b) Window Size 64

Figure 4.10: Intra-socket bi-directional bandwidth measurement taken using OMB on Mist using only the NVLinks connected to the host.



(a) Window Size 1



(b) Window Size 64

Figure 4.11: Inter-socket bi-directional bandwidth measurement taken using OMB on Cedar.

we achieve is 68GB/s when using the host-staged copies alone. This is true for both window sizes.

Using the NVLink between the GPUs, we archived 137GB/s and if we add the additional bandwidth we obtain from using the host-stage copies we should get 205GB/s (137GB/s + 68GB/s). So our measurement of 189GB/s in Figure 4.8 is around 93% of what we expect. Therefore, our results are somewhat reasonable. We think that the slight drop in performance is due to the delay of starting D2H copies and waiting for the H2D copies to start before we can issue the next chunk of data.

When looking at the inter-socket bandwidth tests in Figure 4.11 for Cedar our results are very similar to the intra-socket results. Just like the uni-directional test, the results are consistent between intra and inter-socket. For both windows sizes we get roughly 108GB/s peak bandwidth and a speed up of 1.18x over using NVLink.

## 4.5 Summary

Point-to-point communication is the building block of MPI collective communication and many applications. Many GPU-based point-to-point communication designs have been developed to take advantage of NVLink or PCIe. Traditionally, these designs only use the interconnect that directly connects two devices together. We explored using multiple interconnect paths between devices to see if it would be possible to increase the total available bandwidth. Our proposed idea shows considerable performance improvement in bandwidth at the UCX layer and the MPI layer. We saw the most performance increase for unidirectional bandwidth measurements with large window sizes. With smaller window sizes for bidirectional bandwidth micro-benchmarks, we were still able to obtain nearly a 20% bandwidth increase. Although this mechanism can only be applied to messages larger than 4MB, we do see consistent performance improvement to 1GB. To extend this work for applications in Deep Learning we will design a hierarchical collective in Chapter 5 which uses this multi-path copy as a part of its design.

# Chapter 5

# GPU-Aware `MPI_Allreduce` Design

In Chapter 3, we saw that MPI communication is a major bottleneck in Deep Learning applications. These applications use a mixture of point-to-point and collective communication. The GPU versions of the frameworks which were profiled heavily relied on the usage of `MPI_Allreduce` and up to 90% of communication time was spent in this collective. In Chapter 4 we explored using multiple paths simultaneously to maximise bandwidth in point-to-point operations.

Research [54, 56] and state of the art MPI implementations [3, 33] offer GPU acceleration for their collectives. Open source options such as Open MPI + UCX are not well optimised for collective communication on modern platforms. These open source implementations do not take advantage of all available NVLink interconnects for `MPI_Allreduce` and they also use a CPU based reduction operation. In addition, their collective design is general purpose and is not designed for specific hardware platforms.

Given our prior knowledge that `MPI_Allreduce` is a major bottleneck for Horovod and that we have been able to improve point-to-point communication, In this chapter we propose a new `MPI_Allreduce` collective design which uses our multi-path copy mechanism. We focus our design to be optimised for the Mist platform as that is where we saw the largest performance improvement for the multi-path copy design.

In this chapter we make the following contributions [16]:

- Investigate the performance difference between a CPU-based and GPU-based reduction operation for `MPI_Allreduce` when data resides in GPU global memory.
- We present the importance of platform and accelerator optimised tuning tables for `MPI_Allreduce` and show there is significant performance improvements to be made with a small change to the MPI library.
- We show that our Proposed Hierarchical `MPI_Allreduce` using the Multi-Path Copy, from Chapter 4, can help reduce latency of `MPI_Allreduce` micro-benchmarks.
- Finally, we show that our Proposed Hierarchical `MPI_Allreduce` using the Multi-Path Copy mechanism has a large impact on Deep Learning applications.

## 5.1 Related Work

For CPU-based clusters there is an extensive amount of research regarding collective algorithms for all-to-all communication patterns such as `MPI_Allreduce` and `MPI_Allgather`. In [57], Bruck et al. present all-to-all communication exchange for multi-port CPU systems which allows for $k$ messages to be sent/received at each step of the algorithm. The Reduce-Scatter-AllGather (RSA) algorithm presented in [58] by Rabenseifner tries to optimise bandwidth for any buffer size or process counts for `MPI_Allreduce`. Bandwidth optimised approaches have been designed for the broadcast and reduce stages of an allreduce collective which two binary trees span all processes [59]. This approach allows for collectives to achieve nearly twice the bandwidth of other algorithms. In [60], Thakur et al. investigate flat algorithms implemented in MPICH such as: ring, recursive doubling, RSA, and Bruck for different message sizes and process counts. They observed that the ring algorithm performed better for large messages. The Bruck algorithm was better optimised for short messages.

As multi-node systems are hierarchical in their physical topology, a large amount of research into hierarchical algorithms has been conducted [61–64]. Works investigate the impact of hierarchy on derived data types [63], CPU cache hierarchy [61], PAP-aware algorithms [62] and using Mellanox multi-connection features [64].

For applications in Deep Learning we are mostly interested in GPU specific hierarchical collectives. Research in this area is also not new [56, 65]. With GPU based collectives, data resides in GPU global memory and must be moved between the nodes via the network card. In [65], Chu et al. studied collectives which include a computation component alongside the usual communication part such as `MPI_Allreduce`, `MPI_Reduce`, and `MPI_Scan`. It was investigated whether host-based or GPU-based reductions were more performant in large clusters. The location of the reduction operation affects the data transfer step of the collective. They showed that smaller GPU messages perform better with a CPU-based reductions but larger GPU messages perform better with a GPU-based reduction. Hierarchical GPU-Aware collectives have been studied by investigating the various algorithms one can use at each level; intra-GPU, inter-GPU (intra-node), and inter-node [66].

With the emergence of new high bandwidth interconnects such as NVLink, collectives have been designed for Deep Learning workloads on these platforms. In [46], Awan et al. focused on `MPI_Allreduce` and offloaded the reduction computation on GPUs resulting in significant improvements in Horovod's synthetic benchmarks. Alongside kernel based reductions, in [54] Chu et al. studied the underutilisation of NVLink connections by taking the physical topology into account.

## 5.2 Motivation

We saw in Section 5.1 that there has been many investigations into GPU based reductions for collectives. Currently, open source MPI implementations such as

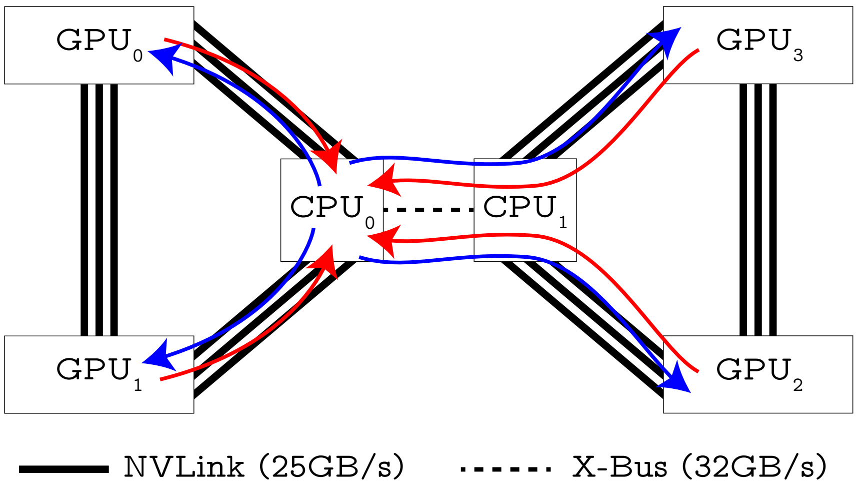


Figure 5.1: Single node physical topology of SciNet system Mist. Uni-directional bandwidth is labelled for each interconnect. Red arrow shows the first step of of copying the data from D2H in the Open MPI implementation of `MPI_Allreduce`. The CPU based reduction occurs using the appropriate collective algorithm. Then the blue arrow shows the second step copy the data from H2D.

Open MPI or MVAPICH2 do not use GPU kernel reductions even when data resides on the GPU. This is problematic for MPI collective performance due to the new NVLink interconnects on modern platforms. On Mist (Figure 5.1), we hypothesise that CPU reductions are a bottleneck as data must be copied to the host, reduced, then copied back to the device. So to send data from $\text{GPU}_0$ to $\text{GPU}_1$ two data transfers occur when using CPU reductions. A GPU reduction would result in only a single data transfer and theoretically halve the latency for a given message size. Aside from the issue of multiple copies, a CPU based reduction also cause NVLink to be underutilised. On Mist the NVLink between the GPUs on each socket are not utilised.

## 5.3 Design and Implementation

In this chapter we explore three additional designs alongside the original Open MPI implementation of `MPI_Allreduce`. These three designs are targeted for `MPI_Allreduce` with the goal of impacting Horovod. As Horovod is our target, we will be focusing on when `MPI_Allreduce` is called with the following call parameters:

- The source buffer is `MPI_IN_PLACE`.
- The receive buffer is allocated on GPU global memory.
- The count parameter gives a message size larger than 1MB.
- The `MPI_Datatype` is `MPI_FLOAT`.
- The `MPI_Op` is `MPI_SUM`.

Horovod uses other `MPI_Datatype` values and `MPI_Op` operations. As we are targeting what we suspect is the major bottleneck in the application, when other use cases of `MPI_Allreduce` are called in the library, we will use the existing Open MPI implementation for all design.

### 5.3.1 `MPI_Allreduce` with GPU Kernel Reduction

As discussed in Section 5.2, a CPU reduction operation leads to under utilised NVLinks and multiple data transfers between processes. We first want to modify the implementation of the collective so that it no longer uses host memory. Then we modified it to execute the reduction operation using a GPU kernel.

#### GPU Buffer Allocation

In the file `coll_cuda_allreduce.c`, Open MPI copies the GPU buffers into host memory if the send or receive buffers are allocated on the GPU. We modified the code so that, if our collective invocation does not match our specific scenario, this copy is executed as described; otherwise, we leave the data in the GPU buffer for the collective to use. This modification alters Open MPI's CUDA Aware `MPI_Allreduce` implementation to keep data in GPU memory during the collective's execution.

When we enter the code of the collective (in the file `coll_base_allreduce.c`), we check the buffer again to verify that it is allocated on the GPU. This may seem like a redundant check but the same algorithm code is used for CPU and GPU buffers, thus it has multiple entry points. For the buffer checking we use the function which already exists inside Open MPI called `opal_cuda_check_bufs()`. If we have a send or receive buffer allocated on the GPU, we allocate the temporary buffer in GPU global memory; otherwise, we allocate a buffer on host memory. The code can be seen in Listing 5.1. This buffer allocation logic handles the multiple entry points into the collective. Upon exit of the collective we free the memory region on host memory. If we allocated the memory on the GPU we wait until `MPI_Finalize` as repeatedly allocating and freeing GPU buffers are fairly slow.

The function labelled `my_CUDA_malloc()` in Listing 5.1 is a wrapper around CUDA memory allocation function. As noted earlier, memory allocation on GPU is fairly slow. Therefore we allocate memory the first time we use the temporary buffer and use the same pointer until the application finishes. Each time the collective runs we use the same pointer.

```
int isCudaBuffer = opal_cuda_check_bufs((char *) sbuf, (char *) rbuf);

void *tmp_buf = isCudaBuffer
                ? (void*) my_CUDA_malloc()
                : (void*) malloc(data_size);
```

Listing 5.1: Buffer Location Allocation Logic

```
extern void * cuda_buff;
static int is_allreduce_malloced = 0;

static inline void * my_cudaMalloc() {
  if (!is_allreduce_malloced) {
    char * env = getenv("PINNED_GPU_BUFFER_SIZE");
    size_t pinned_gpu_buffer_size = (size_t) atol(env);

    cudaMalloc(&cuda_buff, pinned_gpu_buffer_size);
    is_allreduce_malloced = 1;
  }
  return cuda_buff;
}
```

Listing 5.2: Lazy Buffer Allocation Logic

The memory allocation function can be seen in Listing 5.2. Here we read an environment variable which statically chooses the temporary GPU buffer size. Then we use the CUDA Run-time API call `cudaMalloc()` to create the GPU memory region. This functionality is implemented in other MPI libraries but we are adding it to Open MPI so that we can use the designs which are discussed later in this chapter. Pining these temporary buffers allows UCX to send directly between GPUs. Previously, UCX would always send data from GPU to Host or Host to GPU. Changing these buffers allows us to use the CUDA-Aware features of UCX and use its CUDA IPC transport layer. Aside from better utilising the NVLink connections we wanted to use UCX's CUDA IPC code as that is where we implemented the multi-path copy mechanism in Chapter 4.

### GPU Kernel Design

As our new design no longer copies data to the host we cannot use the existing code to reduce our data for `MPI_Allreduce`. Therefore, we designed a simple GPU kernel for the `MPI_FLOAT` and `MPI_SUM` use case. The code of this kernel can be seen in Listing 5.3. This design is fairly simple as the reduction operation required by Open MPI library only reduces two data points at once. When reducing two data points $A$ and $B$ the sum is stored in the original location of the two inputs. Storing the

```
__global__ void vecAddImpl(float *a, float *b, int n)
{
  // Get our global thread ID
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  // Make sure we do not go out of bounds
  if (id < n) {
    float tmp_buf = a[id] + b[id];
    a[id] = tmp_buf;
    b[id] = tmp_buf;
  }
}
```

Listing 5.3: Kernel reduction for `MPI_FLOAT` and `MPI_SUM` use case.

result in one or both does not make a difference to the performance of the kernel as the message sizes we use in Horovod do not saturate HBM2 memory. It was chosen to store the result in the location of both inputs to allow for flexibility in writing the `MPI_Allreduce` collective code.

When compiling this kernel to create a library, which can be called by Open MPI, the Nvidia compiler `nvcc` outputs a C++ library. As Open MPI uses C in its implementation this kernel could not be called from inside the library. To handle this we used the `extern ''C''` keyword so that Open MPI could be linked to this kernel library. The kernel is linked during the configuration phase of building Open MPI using `LIBS=''-lkernel'' ./configure --other-flags`.... Then the header for the kernel was included inside the library and called like a regular function invocation.

### 5.3.2 `MPI_Allreduce` with GPU Kernel Reduction with Tuning Table

Using the buffer modification and the kernel reduction code in Section 5.3.1 we applied it to all other algorithms implemented by Open MPI which were not used by the default execution path. We made the modifications to the following algorithms:

- Recursive Doubling
- Ring
- Segmented Ring
- Basic Linear (Reduce to $P_0$ then Broadcast)
- Reduce Scatter Allgather

Then each algorithm was run for all messages size from 8B to 1GB. For each message size we calculated the average latency for each algorithm Then we chose the optimal algorithm for each message size and created the tuning table. This tuning was completed for four processes on a single node. The tuning would be different for each process and node count. We restricted to this process count as our target is to improve single-node performance. The previous tuning table was designed for CPU based collectives.

### 5.3.3 Proposed Hierarchical `MPI_Allreduce` with Multi-Path Copy

This algorithm was designed to use the multi-path copy mechanism we presented in Chapter 4. This algorithm also uses the same buffer and kernel design as before. As our multi-path copy focuses on intra-socket copy, we will use that feature for intra-socket communication within the collective. In Chapter 4, we saw the multi-path copy had minimal impact for inter-socket copies on Mist. There is a large difference between intra- and inter-socket bandwidth of around 130GB/s to 56GB/s respectively. Therefore we aim to minimise inter-socket transfers. Figure 5.2 shows

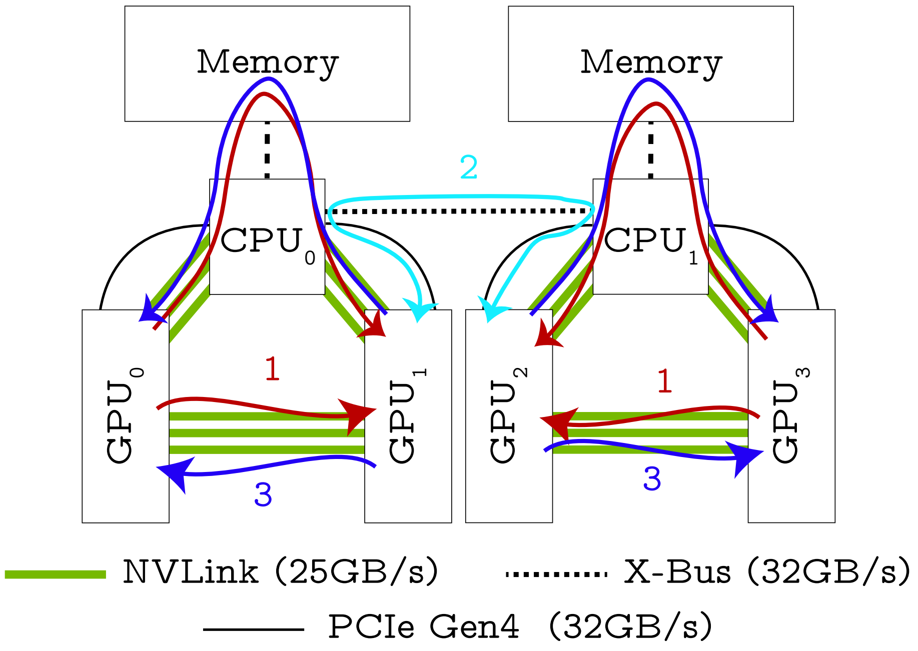


Figure 5.2: Physical topology of Mist with our proposed algorithm overlaid. Uni-directional bandwidth is labelled for each interconnect.

the algorithm in 3 steps:

1. $GPU_0$ and $GPU_3$ sends their data to $GPU_1$ and $GPU_2$ and reduce their data.
2. $GPU_1$ and $GPU_2$ exchange their data and reduce their data.
3. $GPU_1$ and $GPU_2$ sends their data to $GPU_0$ and $GPU_3$.

Due to the large bandwidth discrepancy, mentioned earlier, Step 2 of this algorithm takes roughly 2-3 times longer than Step 1 or 3. Thus it would be beneficial to overlap Step 1 with 2 and Step 2 with 3. To do this, we used pipelining by chunking the send buffer into multiple chunks and having each chunk execute the algorithm independently. Although this causes the steps to overlap it does not result in any overlap between the steps themselves as the CUDA API serialises data transfers between two memory regions. Algorithm 5.1 presents the implementation of pipelining inside Open MPI. The three steps for the algorithm are obscured by our effort to pipeline. The send operation in Line 4 and the recv operation in Line 8 correspond to Step 1. We see additional recvs posted in Line 10 and 5. These are used by Steps 3 and 2 respectively. We post these receives prematurely as this allows for UCX to register the CUDA IPC handle during Step 1 before we need to open the IPC handles Step 2 and Step 3. Next in Lines 14 to 27 we execute Step 2 and Step 3 as the message chunks arrive. In Line 15 we use `MPI_Testany` to check if any phase 1 chunks have been received. If a chunk has arrived, then we reduce the data and send a message as per Step 2. The same dynamic message checking and sending is used in Lines 21-26 to execute Step 3. Finally we use `MPI_Wait` to wait for any outstanding message transfers in Line 39.

There are many tuning tables we created for this collective. The full table can be seen in 5.1. For each message size there is a different number of chunks which gives us the optimal latency for the collective. We collected `MPI_Allreduce` latency for this algorithm for all messages sizes for 2-32 chunks. Then we created a tuning table for the optimal number of chunks.

**Algorithm 5.1** Hierarchical `MPI_Allreduce` with Multi-Path Copy Algorithm

```
for i ← 0 to num_chunks by 1 do
    if 0 == rank or 3 == rank then
        remote_rank = 0 == rank ? 1 : 2; MPI_Isend chunk to remote_rank post
        MPI_Irecv from remote rank (used in phase 3)
    else
        remote_rank = 1 == rank ? 0 : 3; post MPI_Irecv from remote rank (used
        in phase 1). remote_rank = 1 == rank ? 2 : 1; post MPI_Irecv from remote
        rank (used in phase 2).
    end
end
if 1 == rank or 2 == rank then
    while chunks to be sent in either phase 2 or 3 do
        MPI_Testany to check if any phase 1 chunks have been received if chunk
        from phase 1 received then
            reduce data remote_rank= 1 == rank ? 2 : 1; MPI_Isend chunk to
            remote_rank
        end
        MPI_Testany to check if any phase 2 chunks have been received if chunk
        from phase 2 recited then
            reduce data remote_rank = 1 == rank ? 0 : 3; MPI_Isend chunk to
            remote_rank
        end
    end
end
MPI_Wait for any outstanding transfers
```

In [50, 52] the authors noted that using `cudaDeviceDisablePeerAccess()` and `cudaDeviceEnablePeerAccess()` allows us to switch between PCIe and NVLink on platforms with both interconnects. Although this behaviour is not documented within the CUDA API, our experiments gave similar results to what was described in those papers. Therefore, we decided to explore this feature to further reduce inter-socket congestion. We created another tuning table to dynamically switch between PCIe and NVLink in Step 2 of our algorithm.

Finally we added the usage of our multi-path copy. During implementation we noticed that using the multi-path copy while sending data between sockets caused an overall performance degradation. Therefore, we chose chunks which did not have any overlap to use the multi-path copy.

| Message Size (B) | Use Multi-Path? | Inter-Socket Communication Path | Number of Chunks |
|---|---|---|---|
| 32K | No | NVLink | 2 |
| 64K | No | NVLink | 2 |
| 128K | No | NVLink | 2 |
| 256K | No | NVLink | 2 |
| 512K | No | NVLink | 2 |
| 1M | No | NVLink | 2 |
| 2M | No | NVLink | 2 |
| 4M | No | NVLink | 2 |
| 8M | No | NVLink | 4 |
| 16M | Yes | NVLink | 4 |
| 32M | Yes | NVLink | 8 |
| 64M | Yes | PCIe | 8 |
| 128M | Yes | PCIe | 16 |
| 256M | Yes | PCIe | 16 |
| 512M | Yes | PCIe | 16 |
| 1G | Yes | PCIe | 16 |

Table 5.1: Tuning Table for the proposed algorithm

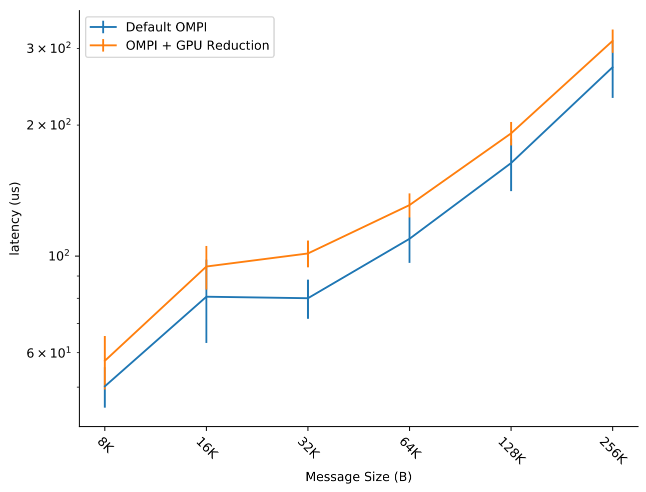
## 5.4 Performance Evaluation and Analysis

### 5.4.1 Experimental Setup

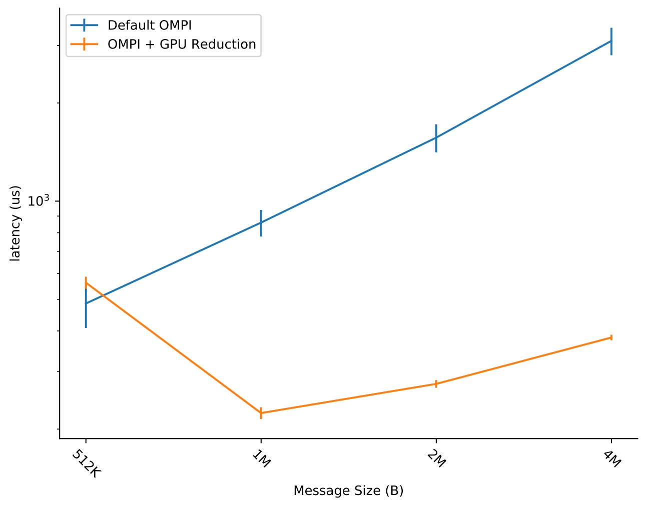
Experiments in this chapter were conducted on Mist using the setup described in Chapter 4. In addition to that environment, we are using Open MPI + HPC-X (with UCX and HCOLL) from HPC-X v2.7 NCCL 2.5.6, and Horovod 0.20.3 with TensorFlow 1.15.2. For our application studies with HPC-X, we used Horovod 0.19.2 as we had runtime issues with Horovod 0.20.3.

### 5.4.2 Micro-Benchmark Results

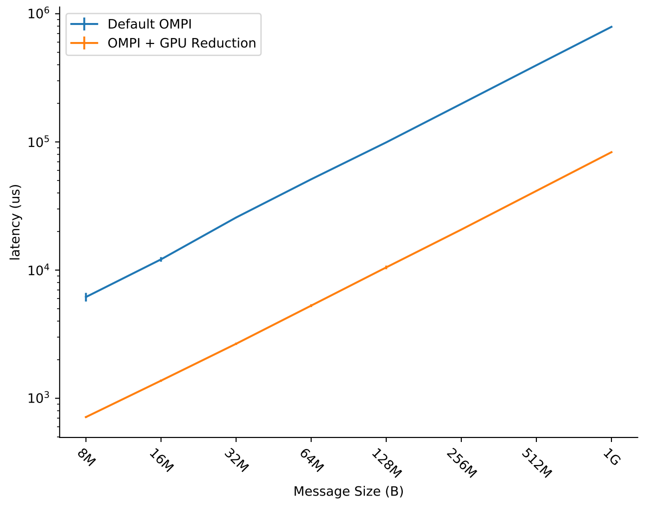
We split this subsection into two parts; first we compare the default Open MPI implementation to the algorithm described in Section 5.3.1, then we compare the three designs together. This is done for the clarity of the figures which will be displayed in this section. For our Micro-benchmark tests we use the OSU Micro-



(a) Medium



(b) Large



(c) Very Large

Figure 5.3: OMB results for `MPI_Allreduce` comparing the default Open MPI implementation with a CPU reduction to our GPU kernel based reduction design on Mist.

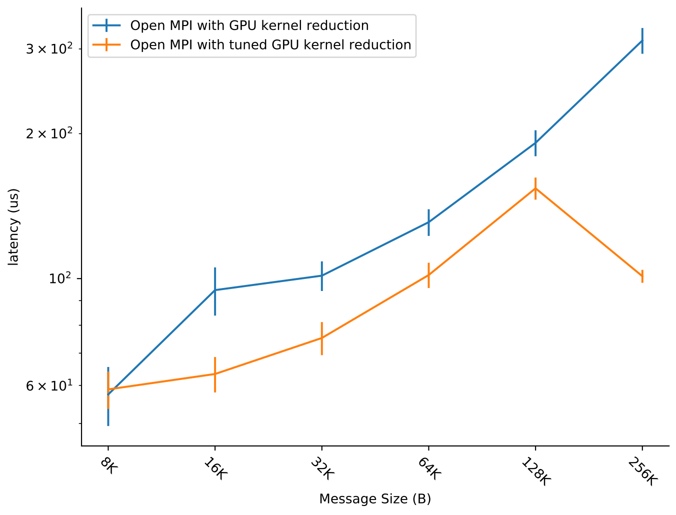
Benchmarking suite for their `MPI_Allreduce` test. We have configured data to be allocated on the GPU for the receive buffer. For the send buffer we have modified the benchmark so that it uses `MPI_IN_PLACE` to mimic the Horovod MPI API calls.

### Default Open MPI vs. Open MPI with GPU kernel reduction

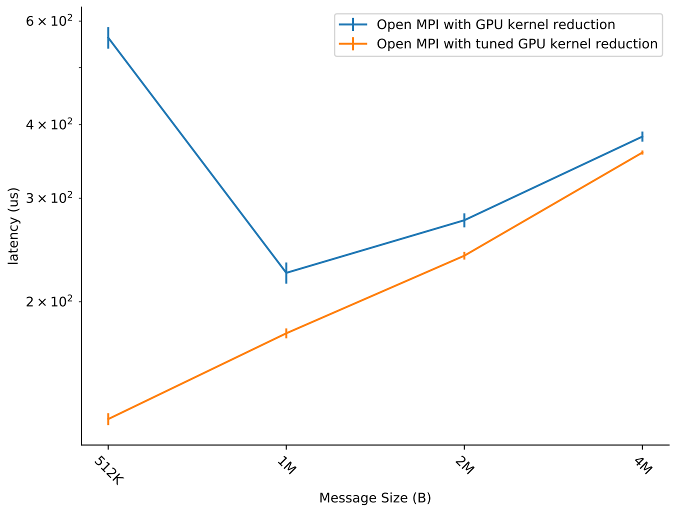
We compare the default Open MPI with a CPU reduction to our first design in Section 5.3.1, where we use a GPU reduction and the idle NVLinks between the GPUs. The micro-benchmark results for this collective design is shown in Figure 5.3. There is a performance degradation for medium sized messages. The CPU based reduction still outperforms the GPU kernel. After 1MB using the additional NVLink and the GPU kernel reduction outperforms the default implementation. For large messages the performance improvement is significant. For a 1MB, there is a 3.8x speedup and up to 9.5x speedup for a 1GB in this collective. This could be due to CPUs being latency optimised and GPUs to be throughput optimised therefore the large messages can take advantage of the parallel kernel reduction. It could also stem from removing the D2H and H2D copies and replacing it with a single D2D copy which halves the number of copies. It is difficult to determine if this performance improvement stems from using the idle NVLinks or the GPU kernel reduction alone, as we cannot use one without the other. We suspect that all of these have contributed to the performance improvement in some capacity.

### Open MPI with GPU kernel reduction vs. Open MPI with tuned GPU kernel reduction

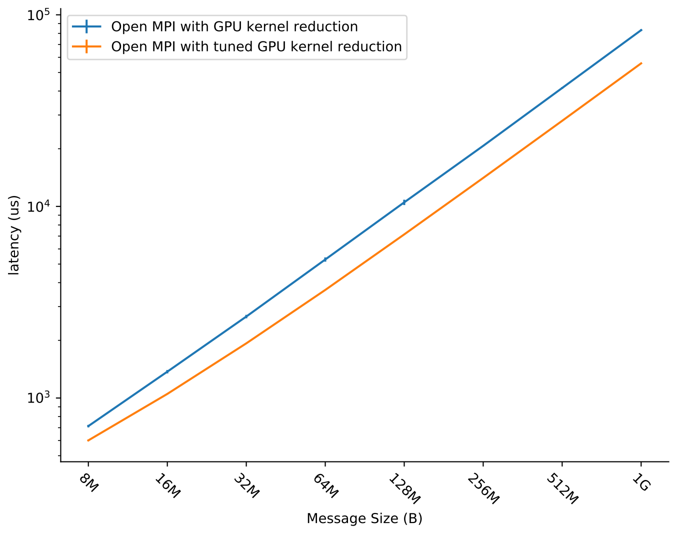
In this section we now study the second design option where with further optimise the collective by using a new static tuning table. The old static tuning table used in the Open MPI with GPU kernel reduction result were based off of the original tuning table for a CPU based reduction. The new tuned GPU kernel reduction



(a) Medium



(b) Large



(c) Very Large

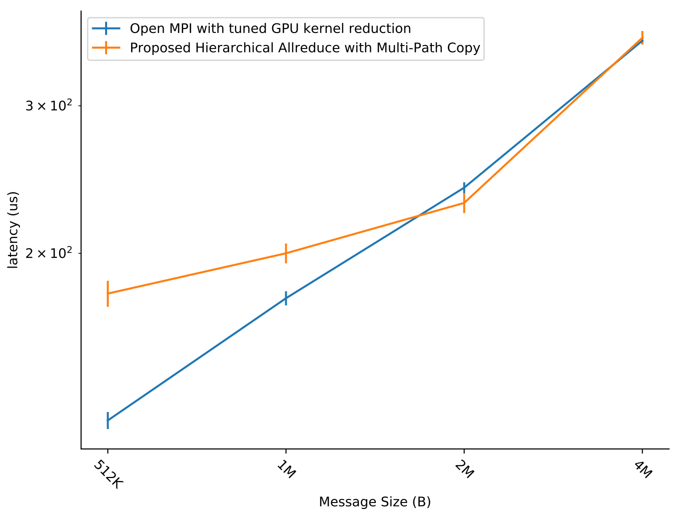
Figure 5.4: OMB results for `MPI_Allreduce` comparing the GPU kernel based reduction design with and without a algorithm tuning table on Mist.

uses a tuning table that was created to optimise the results we see with our GPU kernel design.

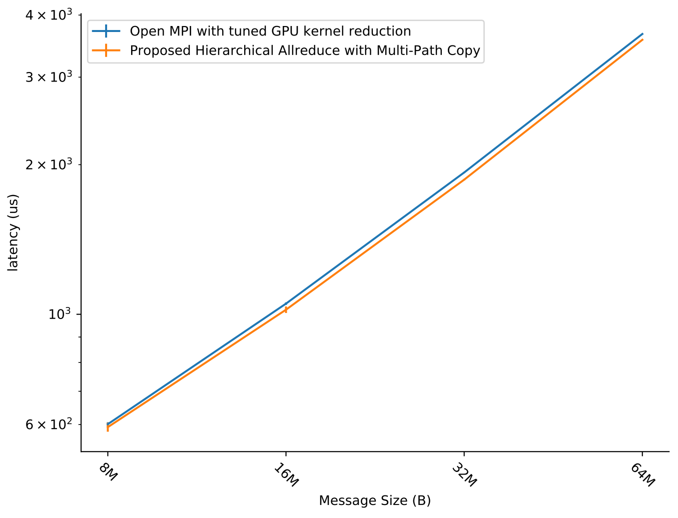
We do not compare it to the original CPU design to provide some extra resolution in our figures. In Figure 5.4 we can see the results of this tuning table. Small messages have been omitted as the default CPU tuning table was already using the optimal algorithm for those message sizes, thus no difference was observed at those sizes. From 16KB all the way to 1GB this new tuning table shows a good performance improvement. That said, the improvement is more incremental than the previous design. We see the largest difference at 512KB with a speedup of 4.5x. We believe this to be a product of the default CPU tuning table choosing a bad algorithm for a particular message size. Improving the chosen algorithms to select the best performance of the GPU kernel reduction collective is what we think is the main impact of our design. At 64MB we observe a speedup of 1.44x and at 1GB we get a 1.49x speedup. Although this is close to a 50% performance improvement we can see we are starting to reach the limitations of the hardware as our improvement are much smaller than the previous section.

#### Open MPI with tuned GPU kernel reduction vs. Proposed Hierarchical Allreduce with Multi-Path Copy

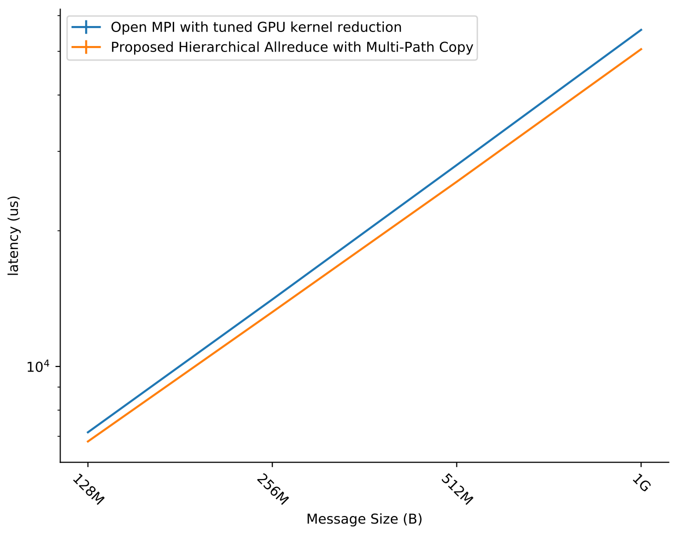
We now compare our proposed algorithm to the tuned GPU Kernel design. The results can be seen in Figure 5.5. As previously stated, we are are getting closer to the hardware limitation and the improvement we see is smaller than before. This proposed algorithm only benefits messages larger than 2MB. From 2MB to 1GB we see a performance improvement of 1.4% to 6.4% reduction in latency.



(a) Medium



(b) Large



(c) Very Large

Figure 5.5: OMB results for `MPI_Allreduce` comparing the tuned GPU kernel based reduction design with our proposed algorithm on Mist.

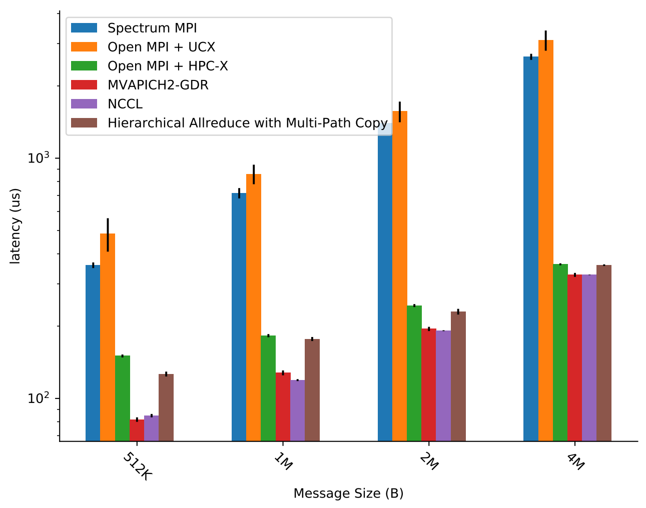
**Proposed Hierarchical Allreduce with Multi-Path Copy compared to existing MPI implementations**

So far, we have observed the incremental improvement to our collective with each design change. Although we see a decent performance improvement with respect to the original implementation, it is important to make comparisons to other MPI implementations to see how our design compares. In this section we have included both the original Open MPI + UCX implementation and the our proposed hierarchical allreduce with multi-path copy.

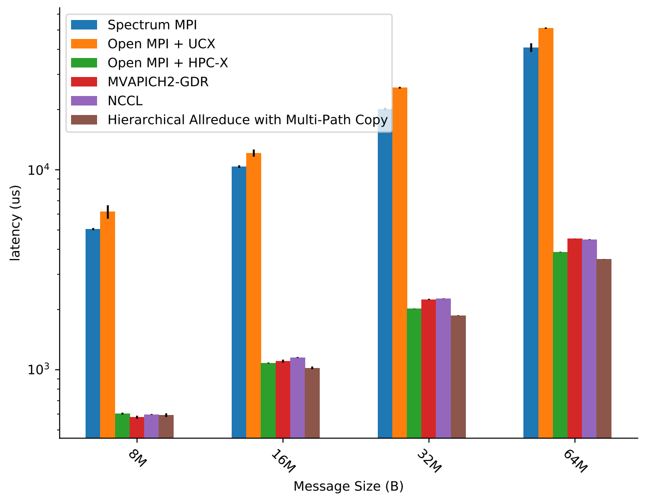
Spectrum MPI marginally outperforms the original Open MPI + UCX. Thus our proposed design outperforms Spectrum MPI for all message sizes. Our proposed hierarchical allreduce with multi-path copy outperforms NCCL and Open MPI + HPC-X for message sizes greater than 8MB.

The proposed algorithm outperforms Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, MVAPICH2-GDR, and NCCL with a speedup of 11.47x, 14.33x, 1.09x, 1.26x, 1.25x, for 64MB messages, respectively. This is the message size Horovod + TensorFlow uses extensively as seen in Chapter 3. We outperform Spectrum MPI, Open MPI + UCX, MVAPICH2-GDR, and NCCL at 1GB messages by 12.25x, 15.63x, 1.47x, and 1.38x, respectively. Results for Open MPI + HPC-X at 512MB and 1GB are not present as we faced CUDA 'out of memory' errors.

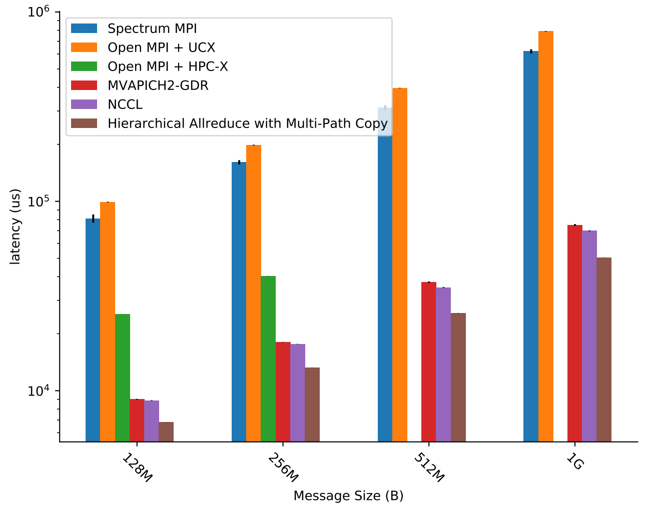
Our design shows significant performance improvements over Open MPI + UCX and Spectrum MPI. We suspect that this is because `MPI_Allreduce` in these libraries use a CPU based reduction even for GPU resident data. Our design also performs better than NCCL, Open MPI + HPC-X and MVAPICH2-GDR, which are both GPU-optimised communication libraries.



(a) Medium to Large



(b) Large



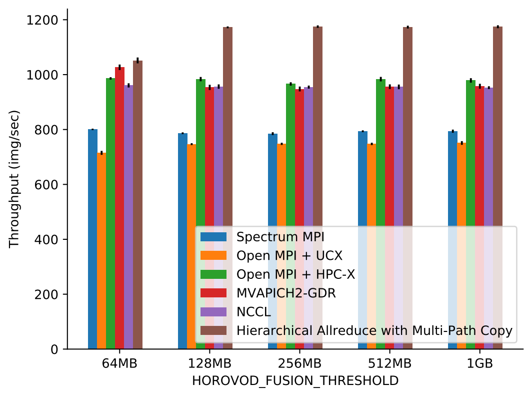
(c) Very Large

Figure 5.6: `MPI_Allreduce` results comparing our proposed hierarchical allreduce with multi-path copy against Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, MVAPICH2-GDR, and NCCL on Mist

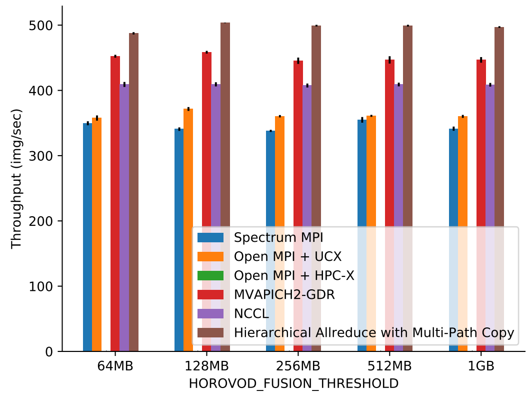
### 5.4.3 Application Results

In this section we are extending our evaluations to Deep Learning applications. Figure 5.7 presents the performance of our proposed design with Horovod with TensorFlow and four Deep Learning models: ResNet50, ResNet152, DenseNet201, and VGG16. We varied the Horovod tuning parameter `HOROVOD_FUSION_THRESHOLD` to see if any additional performance could be gained for the proposed MPI designs [67]. The default value of this parameter is 64MB.

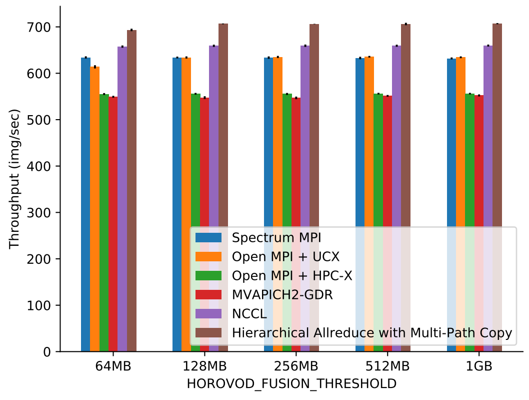
For ResNet50, we see a throughput speedup of up to 1.49x, 1.56x, 1.19x, 1.23x, and 1.21x over Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, MVAPICH2-GDR, and NCCL, respectively. With a fusion threshold larger than 128MB we see



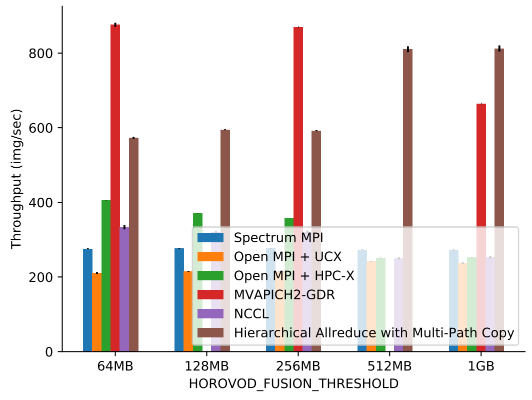
(a) ResNet50



(b) ResNet152



(c) DenseNet201



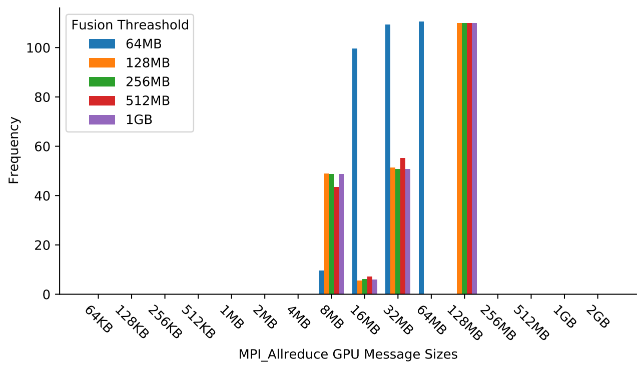
(d) VGG16

Figure 5.7: Horovod + TensorFlow throughput with different models, a batch size of 32, and different values of `HOROVOD_FUSION_THRESHOLD` for each MPI implementation and NCCL on Mist

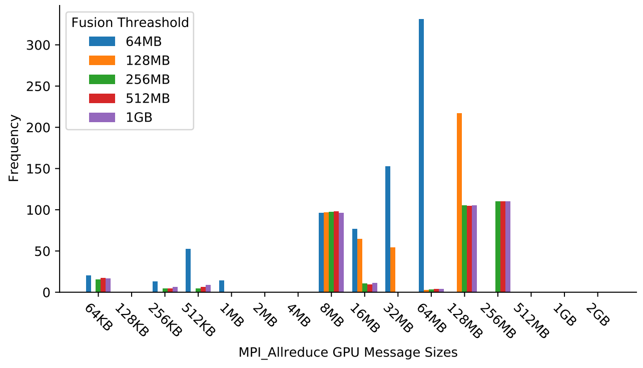
minimal change in our performance. With ResNet152 and for a fusion threshold of 64MB, we observe a throughput speedup of 1.40x, 1.36x, 1.08x, and 1.19x over Spectrum MPI, Open MPI + UCX, MVAPICH2-GDR, and NCCL, respectively. Our design outperforms other implementations, but we noticed for this model, there was a slight performance drop using our proposed hierarchical algorithm compared to our allreduce with GPU kernel reduction. For ResNet152, we were unable to obtain results for Open MPI + HPC-X as this model would cause the application to seg-fault. A modest performance speedup of 1.09x, 1.13x, 1.25x, 1.26x, and 1.06x is seen over Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, MVAPICH2-GDR, and NCCL with DenseNet201, respectively.

VGG16 shows the highest performance improvement when using our proposed collective. We observe a speedup of 2.08x, 2.72x, 1.84x, and 1.72x over Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, and NCCL for 64MB buffers, respectively. With this model, we saw that tuning the framework improved the performance further. This tuning gave us 2.98x, 3.42x, 3.20x, and 3.21x speedup for 1GB buffers over Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, and NCCL, respectively. For VGG16, we present the results for MVAPICH2-GDR for completeness, as MVAPICH2-GDR would often hang and not return results. We had a 5% success rate for job submissions for this model. This is why only 3 out of 5 data points are presented in this figure. That said, for a buffer size of 64MB we observe a speedup of 0.67x and 1.23x at 1GB. Overall, it is clear from these results that the performance improvement for Horovod + TensorFlow with the proposed `MPI_Allreduce` is significant, but fairly dependent on the model.

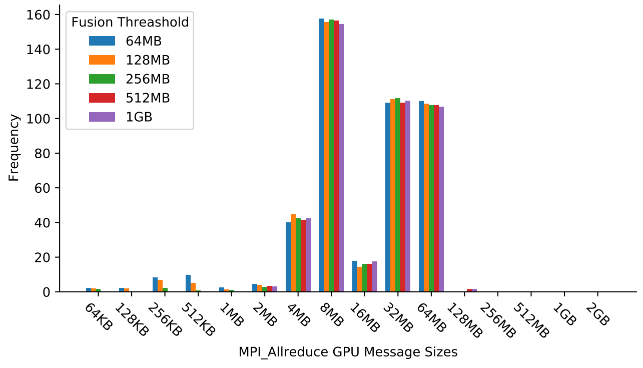
We profiled `MPI_Allreduce` during Horovod's execution for different Deep Learning models to draw a connection between our proposed design and the application performance. Figure 5.8 presents the frequency of message sizes used by `MPI_Allreduce`. With ResNet50, increasing the fusion buffer from 64MB to 128MB



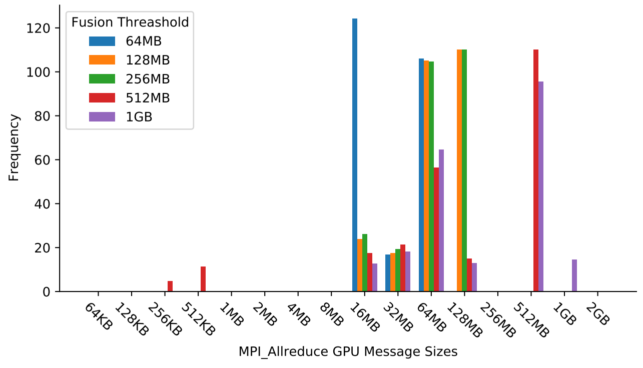
(a) ResNet50



(b) ResNet152



(c) DenseNet201



(d) VGG16

Figure 5.8: Message sizes used by `MPI_Allreduce` for Horovod + TensorFlow with different models, a batch size of 32, and `HOROVOD_FUSION_THRESHOLD` on Mist.

changes the message sizes. We see a reduction in messages in 16-64MB range but an increase for 128MB. As we increase the threshold we see no further changes. When comparing this observation to the throughput results in Figure 5.7, we see that throughput increases for our designs when the threshold changes from 64MB to 128MB but stays constant afterwards. For ResNet152 we observe similar results to ResNet50 but increasing the threshold past 128MB also results in generating smaller messages. For DenseNet201, we see no impact on message size when changing the threshold. This is also reflected in the throughput measurements being consistent with different thresholds values. Finally, with VGG16 we see mostly message sizes of 16MB and 64MB with a 64MB tensor fusion threshold. It is evident that in this model increasing the threshold directly increases the message sizes used in `MPI_Allreduce`. This is also reflected in the throughput results for our proposed algorithms, since increasing the thresholds yield a better overall performance for VGG16.

## 5.5 Summary

Collectives such as `MPI_Allreduce` are an important part of Deep Learning applications. Focusing research on this topic is important in accelerating these emerging applications. As the `MPI_Allreduce` usage in Horovod is GPU-based we focused on algorithms using GPU buffers and modern GPU features such as NVLink.

In this chapter we incrementally changed the `MPI_Allreduce` collective and observed the source of the performance improvements. When we evaluated Default Open MPI vs. Open MPI with GPU kernel reduction we saw the largest performance improvement for large messages. Since our GPU kernel reduction design allows data to be transferred directly from one GPU to another without staging we reduce the number of copies required for the collective. We next created a new algorithm tuning table for our `MPI_Allreduce` collective. This tuning table improved performance as the tuning decision were based of GPU collective performance rather than the previous implementation basing their tuning of CPU collectives. Finally we added our Proposed Hierarchical Allreduce with Multi-Path Copy algorithm to our tuning table and we outperformed both Open MPI + UCX and Spectrum MPI for all message sizes. We also outperformed MVAPICH2-GDR, Open MPI + HPC-X and NCCL for very large message sizes.

Then we evaluated our proposed algorithm with Horovod to observe its impact on Deep Learning applications. We generally saw some performance improvement across a few Deep Learning models. With additional tuning of Horovod we were able to get a significant performance improvement for certain Deep Learning models. Finally we profiled the Deep Learning application again to draw a connection between the performance improvement we saw from the additional framework tuning.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

Communication is one of the most critical issues facing HPC and Deep Learning applications today. As most applications in these domains depend upon MPI for their communication, investigating MPI implementations themselves is incredibly important. In this thesis, we engage in a few research directions with the goal of improving the performance of Deep Learning applications on HPC systems. We begin with a workload characterisation of some Deep Learning Frameworks to understand which parts of the MPI library are used. Then we develop an improved point-to-point communication mechanism, and finally we integrate the point-to-point communication mechanism within the context of a collective and apply it to some Deep Learning applications. The rest of this section will conclude each chapter in greater detail.

### Communication Characterisation of Distributed Deep Learning Frameworks

In Chapter 3, we investigated MPI communication characteristics of a few Deep Learning applications. Similar to traditional HPC applications, DL applications use a mixture of point-to-point and collectives. We found that `MPI_Allreduce` and

`MPI_Bcast` were the most frequently used collectives in both Horovod and CNTK. Horovod relied heavily on `MPI_Allreduce` during run-time and spent the majority of its communication time in that collective. We also explored Horovod's NCCL configurations where the application uses a mixture of NCCL and MPI at run-time. We saw the application use MPI for small messages. Horovod offload GPU communication for `MPI_Allreduce` to the NCCL library where it uses NCCL allreduce. We observed that Horovod's offloading of large GPU message to NCCL resulted in an increased throughput for applications using 64 GPUs. CNTK was the only framework that uses point-to-point communication with all four training methods. We measured some metrics regarding point-to-point communication such as message queues traversal times and point-to-point communication pairs. We found that CNTK frequently sends and receives data from rank 0. This made us suspect rank 0 could have issues handling the MPI message queue. When profiling CNTK, we observed that CNTK frequently checks the message queue but we never saw more than a couple messages present in the queue. Therefore, we believe that CNTK will not benefit from using any message matching techniques.

## Multi-Path Point-to-Point GPU Communication

In Chapter 4 we proposed an intra-socket multi-path point-to-point communication algorithm for a UCX Put. With modern NVLink platforms we saw that there are often additional communication paths between GPU pairs that are not used at run-time. Our design is indented to use all available paths to aggregate the bandwidth between GPUs. We striped the data across NVLinks or PCIe connections and when transferring the data via the host we would stage it in host memory. We further improved this message striping by placing each chunk of data on different CUDA streams so that the CUDA run-time could pipeline our design further.

We evaluated this design on two HPC systems, one with PCIe to the host and the other with NVLink to the host. We observed that on the platform with NVLink to the host performed significantly better than the system with PCIe. This was largely due to the higher bandwidth that NVLinks provide. That said, on both platforms we obtained significant performance improvement when comparing our results to the default single-path UCX. We applied this put operation to Open MPI + UCX and we saw its merit for MPI point-to-point communication with minimal code changes.

### GPU-Aware `MPI_Allreduce` Design

In Chapter 5 we proposed an `MPI_Allreduce` with GPU Kernel Reduction, then extended that design by creating a new algorithm tuning table, and finally we added our proposed Hierarchical `MPI_Allreduce` with Multi-Path Copy algorithm to that tuning table. The design change to use a GPU kernel based reduction saw significant performance improvement for large messages. We believe it is partially due to GPU kernels performing better than CPUs for large data computations but we also think that it is due to never moving data to the host. The direct copies between devices allowed us to better utilise UCX's GPU communication optimisations. The addition of our tuning table for algorithm selection further improved this design as we were now choosing the collective based on its performance on GPUs, whereas previously the decision was based of CPU collectives. Our proposed Hierarchical Allreduce with Multi-Path Copy algorithm performed very well compared to other existing MPI implementations, especially for very large messages. We evaluated the performance of our proposed `MPI_Allreduce` collective with Horovod + TensorFlow and various models. For Horovod with TensorFlow and VGG16, we observe up to 2.98x, 3.42x, 3.22x, 1.23x, and 3.24x speedup over Spectrum MPI, Open MPI + UCX, Open MPI + HPC-X, MVAPICH2-GDR, and NCCL, respectively. Finally, we profiled the Deep Learning application again to draw a connection between the performance improvement we saw from the additional framework tuning.

## 6.2 Future Work

Challenges in MPI communication will continue to exist for years to come. We plan to extend our work in a few ways for new applications and compute systems. In the following section we will discuss how our algorithms can be extended.

### Communication Characterisation of Distributed Deep Learning Frameworks

In Chapter 3, our work was limited to a few Deep Learning frameworks. There are many more available frameworks which we would like to study to see if `MPI_Allreduce` is still the dominant collective in Deep Learning. Model and hybrid parallelism still appears to be in the research stage of development [35, 36]. As the availability of these frameworks increase, we hope to see how this changes MPI communication. Also, we only studied CNN as our focus was Deep Learning, It would be interesting to investigate different areas within Machine Learning such as Reinforcement Learning.

### Multi-Path Point-to-Point GPU Communication

Chapter 4 showed that our multi-path copy was limited by the available bandwidth to the host. As new GPU platforms with PCIe Gen 5 become available it would be interesting to apply this design to those systems. We expect PCIe Gen 5 to offer more bandwidth to the host than our PCIe 3 platform. This could overcome the issues we observed with PCIe.

A limitation of this work was that we created many static tuning tables for the multi-path copy. Using an auto-tuning mechanism to decide the data sizes sent through each path and the number of CUDA streams could make this design more portable. Alongside portability, this would also allow for the multi-path copy to adjust at run-time to handle any congestion.

The new MPI-4.0 standard has introduced partitioned point-to-point communication. In short, this new interface partitions a send buffer into multiple parts and the MPI run-time transfers the data as each partition is ready. We could extend this multi-path design for this new interface by sending each partition across a different path to better utilise bandwidth and reduce congestion.

### GPU-Aware `MPI_Allreduce` Design

Our collective design in Chapter 5 was limited to a single node using four processes. Deep Learning applications often perform better with one process per GPU. If we were to step outside of Deep Learning and into traditional HPC we could extend this design to work with multiple process per GPU. We saw in Chapter 4 that increasing the window size in bandwidth tests would result in improved bandwidth utilisation. Therefore, increasing the number of processes per node could yield some improvements. We could also extend this design for a cluster wide collective. As this design is restricted to four processes we would have to either design a generalised flat algorithm or a hierarchical approach that uses the multi-path copy for the intra-node portion of the algorithm.

This algorithm could also be extended with similar auto-tuning approaches to what we discussed for our extensions in Chapter 4. The number of chunks used for pipelining of this collective, switching between PCIe or NVLink, and enabling and disabling the multi-path copy were decided on static metrics.

# Bibliography

[1] (2021) Message Passing Interface. [Online]. Available: http://www.mpi-forum.org

[2] (2021, March) High-Performance Portable MPI. [Online]. Available: http://www.mpich.org

[3] (2021, March) MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. [Online]. Available: http://mvapich.cse.ohio-state.edu/

[4] (2021, January) Open Source High Performance Computing. [Online]. Available: https://www.open-mpi.org/

[5] B. Klenk and H. Fröning, "An Overview of MPI Characteristics of Exascale Proxy Applications," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Cham: Springer International Publishing, 2017, pp. 217–236.

[6] S. Kamil, J. Shalf, L. Oliker, and D. Skinner, "Understanding Ultra-Scale Application Communication Requirements," in *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, Oct 2005, pp. 178–187.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens,

B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[8] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," in *NeurIPS Autodiff Workshop*, 2017, pp. 1–4.

[9] A. Sergeev and M. D. Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," *arXiv:1802.05799*, pp. 1–10, 2018.

[10] 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*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 2135–2135. [Online]. Available: http://doi.acm.org/10.1145/2939672.2945397

[11] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 66–77.

[12] C. 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)*, Aug 2017, pp. 161–170.

[13] A. A. Awan, A. Jain, C. Chu, H. Subramoni, and D. K. Panda, "Communication Profiling and Characterization of Deep Learning Workloads on Clusters with High-Performance Interconnects," *IEEE Micro*, pp. 35–43, 2019.

[14] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler,

"SparCML: High-performance Sparse Communication for Machine Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019, pp. 11:1–11:15. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356222

[15] (2021, March) NVLINK AND NVSWITCH. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[16] Y. H. Temucin, A. Sojoodi, P. Alizadeh, and A. Afsahi, "Efficient Multi-Path NVLink/PCIe-Aware UCX based Collective Communication for Deep Learning," in *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2021, p. 1–10.

[17] Top500 - June 2020. [Online]. Available: https://www.top500.org/lists/top500/2020/06/

[18] J. A. Anderson, J. Glaser, and S. C. Glotzer, "HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations," *Computational Materials Science*, vol. 173, p. 109363, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0927025619306627

[19] W. M. Brown, "GPU Acceleration in LAMMPS," pp. 1–35, 2011.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[21] R. Raina, A. Madhavan, and A. Ng, "Large-Scale Deep Unsupervised Learning using Graphics Processors," vol. 382, 01 2009, p. 110.

[22] D. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, High Performance Convolutional Neural Networks for Image Classification." 07 2011, pp. 1237–1242.

[23] T. Groves, Y. Gu, and N. J. Wright, "Understanding Performance Variability

on the Aries Dragonfly Network," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 809–813.

[24] 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 *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 1–9.

[25] (2021) InfiniBand Trade Association. [Online]. Available: http://www.infinibandta.org/

[26] "NVIDIA TESLA V100 GPU ARCHITECTURE," Nvidia, Tech. Rep., August 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[27] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, 2012, pp. 1848–1857.

[28] D. D. Sharma, "PCI Express® 6.0 Specification at 64.0 GT/s with PAM-4 signaling: a low latency, high bandwidth, high reliability and cost-effective interconnect," in *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2020, pp. 1–8.

[29] (2021, March) Optimized Primitives for Inter-GPU Communication. [Online]. Available: https://github.com/NVIDIA/nccl

[30] A. Skjellum, N. E. Doss, and K. Viswanathan, "Inter-communicator Extensions to MPI in the MPIX (MPI eXtension) Library," Tech. Rep., 1994.

[31] K. Kandalla, D. Knaak, K. McMahon, N. Radcliffe, and M. Pagel, "Optimizing Cray MPI and Cray SHMEM for Current and Next Generation Cray-XC Supercomputers," *Cray User Group (CUG)*, vol. 2015, 2015.

[32] (2021, March) IBM Spectrum MPI. [Online]. Available: https://www.ibm.

com/products/spectrum-mpi

[33] (2021, March) Nvidia HPC-X. [Online]. Available: https://developer.nvidia.com/networking/hpc-x

[34] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "UCX: an Open Source Framework for HPC Network APIs and Beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[35] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-TensorFlow: Deep Learning for Supercomputers," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018, pp. 10 414–10 423. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/3a37abdeefe1dab1b30f7c5c7e581b93-Paper.pdf

[36] A. A. Awan, A. Jain, Q. Anthony, H. Subramoni, and D. K. Panda, "HyParFlow: Exploiting MPI and Keras for Scalable Hybrid-Parallel DNN Training with TensorFlow," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 83–103.

[37] A. Vishnu, C. Siegel, and J. Daily, "Distributed TensorFlow with MPI," *arXiv:1603.02339*, pp. 1–6, 2016.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv:1408.5093*, pp. 1–4, 2014.

[39] 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 US Exascale Computing Project," *Concurrency and*

*Computation: Practice and Experience*, vol. 32, no. 3, p. e4851, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4851

[40] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI Usage on a Production Supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 30:1–30:15. [Online]. Available: https://doi.org/10.1109/SC.2018.00033

[41] R. Zamani and A. Afsahi, "Communication Characteristics of Message-Passing Scientific and Engineering Applications," in *International Conference on Parallel and Distributed Computing Systems, PDCS 2005, November 14-16, 2005, Phoenix, AZ, USA*, 2005, pp. 644–649.

[42] S. M. Ghazimirsaeed, S. H. Mirsadeghi, and A. Afsahi, "Communication-Aware Message Matching in MPI," *Concurrency and Computation: Practice and Experience*, vol. 32, pp. 1–17. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4862

[43] 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," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2019, pp. 49–53.

[44] 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*, ser. EuroMPI 2016. New York, NY, USA: ACM, 2016, pp. 15–22. [Online]. Available: http://doi.acm.org/10.1145/2966884.2966912

[45] A. A. Awan, C.-H. Chu, H. Subramoni, and D. K. Panda, "Optimized Broadcast for Deep Learning Workloads on Dense-GPU InfiniBand Clusters:

MPI or NCCL?" in *Proceedings of the 25th European MPI Users' Group Meeting*, ser. EuroMPI'18. New York, NY, USA: ACM, 2018, pp. 2:1–2:9. [Online]. Available: http://doi.acm.org/10.1145/3236367.3236381

[46] A. A. Awan, J. Bédorf, C. Chu, H. Subramoni, and D. K. Panda, "Scalable Distributed DNN Training using TensorFlow and CUDA-Aware MPI: Characterization, Designs, and Performance Evaluation," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2019, pp. 498–507.

[47] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. Panda, "Performance Characterization of DNN Training using TensorFlow and PyTorch on Modern Clusters," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2019, pp. 1–11.

[48] N. Laanait, J. Romero, J. Yin, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Exascale Deep Learning for Scientific Inverse Problems," pp. 1–13, 2019.

[49] 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*, ser. ESPM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 47–50. [Online]. Available: https://doi.org/10.1145/2832241.2832247

[50] C. Pearson, A. Dakkak, S. Hashash, C. Li, I.-H. Chung, J. Xiong, and W.-M. Hwu, "Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 209–218. [Online]. Available: https://doi.org/10.1145/3297663.3310299

[51] N. R. Tallent, N. A. Gawande, C. Siegel, A. Vishnu, and A. Hoisie, "Evaluating

On-Node GPU Interconnects for Deep Learning Workloads," in *8th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, ser. Lecture Notes in Computer Science, vol. 10724, 2017, pp. 3–21.

[52] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker, "Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 191–202.

[53] J. A. Stuart, P. Balaji, and J. D. Owens, "Extending MPI to Accelerators," in *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2377978.2377981

[54] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. D. K. Panda, "NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3392717.3392771

[55] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda, "OMB-GPU: A Micro-Benchmark Suite for Evaluating MPI Libraries on GPU Clusters," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 110–120. [Online]. Available: https://doi.org/10.1007/978-3-642-33518-1_16

[56] I. Faraji and A. Afsahi, "GPU-Aware Intranode MPI_Allreduce," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 45—50. [Online]. Available: https://doi.org/10.1145/2642769.2642773

[57] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[58] R. Rabenseifner, "Optimization of Collective Reduction Operations," in *Computational Science - ICCS 2004*, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–9.

[59] P. Sanders, J. Speck, and J. L. Träff, "Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan," *Parallel Comput.*, vol. 35, no. 12, pp. 581—594, Dec. 2009. [Online]. Available: https://doi.org/10.1016/j.parco.2009.09.001

[60] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49—66, Feb. 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[61] 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)*, 2008, pp. 130–137.

[62] Y. Qian and A. Afsahi, "Process Arrival Pattern Aware Alltoall and Allgather on InfiniBand Clusters," *International Journal of Parallel Programming*, vol. 39, pp. 473–493, 08 2011.

[63] J. L. Träff and A. Rougier, "MPI Collectives and Datatypes for Hierarchical All-to-All Communication," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 27—32. [Online]. Available:

https://doi.org/10.1145/2642769.2642770

[64] Y. Qian, M. Rashti, and A. Afsahi, "Multi-Connection and Multi-Core Aware All-gather on InfiniBand Clusters," *IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 1–7, 01 2008.

[65] C. Chu, K. Hamidouche, A. Venkatesh, A. A. Awan, and D. K. Panda, "CUDA Kernel Based Collective Reduction Operations on Large-scale GPU Clusters," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 726–735.

[66] I. Faraji and A. Afsahi, "Design Considerations for GPU-Aware Collective Communications in MPI," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 17, p. e4667, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4667

[67] Q. Anthony, A. A. Awan, A. Jain, H. Subramoni, and D. K. D. Panda, "Efficient Training of Semantic Image Segmentation on Summit using Horovod and MVAPICH2-GDR," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 1015–1023.