Optimization of MPI Collective Communication Operations

Xi Luo
University of Tennessee, xluo12@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Xi Luo entitled "Optimization of MPI Collective Communication Operations." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Dali Wang, Michela Taufer, Yingkui Li

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

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

Optimization of MPI Collective Communication Operations

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Xi Luo

May 2020

To my parents Zhu Luo and Jiaxi Cao,

my wife Qijing Yu for their love, trust, and support.

Acknowledgments

I would like to thank my advisor, Dr. Jack Dongarra, for giving me the opportunity to join Innovative Computing Laboratory (ICL) as a graduate research assistant. I am also grateful to Dr. Dongarra for providing generous financial support for my research. It is a great privilege to work with him, and his experience and wisdom will always be the guidance for my future work and life.

I would also like to thank my co-advisor and group leader, Dr. George Bosilca, for his guidance, motivation, and support during my graduate study. His kindness, patience, and encouragement made my study a pleasant experience, and I feel lucky to be his student. He introduced me to the high-performance computing field and has been guiding me through my whole Ph.D. study. I could not finish this dissertation without his help.

I am grateful to Dr. Dali Wang for giving me the opportunity to work at Oak Ridge National Lab and mentoring me during the internship. I am also grateful to him, Dr. Michela Taufer, and Dr. Yingkui Li for serving on my dissertation committee. I greatly appreciate their time and invaluable guidance on my dissertation.

I would like to express my appreciation to my current and former colleagues at ICL, including Dr. Thomas Herault, Dr. Aurelien Bouteiller, Dr. Anthony Danalis, Dr. Wei Wu, Dr. Chongxiao Cao, Dr. Reazul Hoque, Dr. Thananon Patinyasakdikul, Dr. Panruo Wu, David Eberius, Zhong Dong, Yu Pei, Qinglei Cao, and more, for their help and company. I wish them all the best.

Lastly, I would like to express my deepest gratitude to my parents Zhu Luo and Jiaxi Cao, and my wife Qijing Yu for their love, trust, and support. I also thank my friends, Zheng Lu and Yunhe Feng for the friendship and fun.

Abstract

High-performance computing (HPC) systems keep growing in scale and heterogeneity to satisfy the increasing need for computation, and this brings new challenges to the design of Message Passing Interface (MPI) libraries, especially with regard to collective operations.

The implementations of state-of-the-art MPI collective operations heavily rely on synchronizations, and these implementations magnify noise across the participating processes, resulting in significant performance slowdowns. Therefore, I create a new collective communication framework in Open MPI, using an event-driven design to relax synchronizations and maintain the minimal data dependencies of MPI collective operations.

The recent growth in hardware heterogeneity results in increasingly complex hardware hierarchies and larger communication performance differences. Hence, in this dissertation, I present two approaches to perform hierarchical collective operations, and both can exploit the different bandwidths of hardware in heterogeneous systems and maximizing concurrent communications.

Finally, to provide a fast and accurate autotuning mechanism for my framework, I design a new autotuning approach by combining two existing methods. This new approach significantly reduces the search space to save the autotuning time and is still able to provide accurate estimations.

I evaluate my work with microbenchmarks and applications at different scales. Microbenchmark results show my work speedups MPI_Bcast and MPI_Allreduce up to 7.34X and 4.86X, respectively, on 4096 processes. In terms of applications, I achieve a 24.3% improvement for Hovorod and a 143% improvement for ASP on 1536 processes as compared to the current Open MPI.

Table of Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Propagation of Noise . . . 2

1.1.2 Hardware Heterogeneity and Hierarchy . . . 3

1.1.3 Parameter Autotuning . . . 3

1.2 Contributions . . . 5

1.2.1 Noise-Resistant Collective Operations . . . 5

1.2.2 Hierarchical Collective Operations . . . 5

1.2.3 Autotuning of Collective Operations . . . 6

1.3 Dissertation Outline . . . 7

2 Background and Literature Review of Related Work **9**

2.1 Overview . . . 9

2.2 MPI . . . 9

2.2.1 MPI Collective Communication Operations . . . 10

2.2.2 The Open MPI Library . . . 11

2.3 Noise Propagation . . . 13

2.3.1 Eliminate Noise from the Source . . . 13

2.3.2 Alleviate Noise Propagation . . . 13

2.4 Hierarchical Collective Operations . . . 14

2.4.1 Explore More Levels . . . 14

2.4.2 Communication Overlap . . . 14

2.5 Autotuning of Collective Operations . . . 15

2.5.1 Offline Tuning 15
2.5.2 Online Tuning 16

3 Noise Resistant Collective Operations **18**
3.1 Overview 18
3.2 Existing Implementations 19
3.2.1 Collectives Using Blocking Point-to-Point Routines 20
3.2.2 Collectives Using Nonblocking Point-to-Point Routines 24
3.3 ADAPT: Event-Driven Design 26
3.3.1 Implementation of ADAPT 26
3.3.2 Analysis of Dependencies in ADAPT 31
3.3.3 Extend ADAPT to Other Collective Operations 32
3.3.4 Support Different Collectives with Multiple Communication Trees . . 33
3.4 Evaluation 34
3.4.1 Experimental Setup 34
3.4.2 Noise Impact 34

4 Hierarchical Collective Operations - Topology-Aware Tree **40**
4.1 Overview 40
4.2 Topology-Aware Communication Tree 41
4.2.1 The Problems of Existing Tree-Based Algorithms 41
4.2.2 Build a Topology-Aware Tree 43
4.3 ADAPT vs. Other Implementations 45
4.4 Evaluation 47
4.4.1 Experimental Setup 47
4.4.2 Performance of Different Topology-aware Algorithms 47
4.4.3 Performance of MPI Implementations 49

5 Hierarchical Collective Operations - Combine Specialized Modules **53**
5.1 Overview 53
5.2 Specialized Modules in Open MPI 54

5.2.1 Inter-Node Modules 54

5.2.2 Intra-Node Modules 55

5.3 Combine Specialized Modules - HAN: Task-Based Design 61

5.3.1 MPI_Bcast 64

5.3.2 MPI_Allreduce 73

6 Autotuning of Collective Communication Operations 81

6.1 Overview 81

6.2 Task-Based Autotuning 82

6.3 Evaluation 86

6.3.1 Experimental Setup 86

6.3.2 Benchmark Results 86

6.3.3 Application Results 92

7 Conclusions and Future Work 95

7.1 Conclusions 95

7.2 Future Work 97

Bibliography 99

Vita 108

List of Tables

6.1 Inputs of Automatic Tuning 82
6.2 Available configurations in the HAN module (MPI_Bcast and MPI_Allreduce, $iralg$ and irs are only used for MPI_Allreduce) 83
6.3 Performance of ASP (1536 processes, 1M Matrix) 93

List of Figures

3.1 Implementation of MPI_Bcast using blocking point-to-point routines (red: noise source; orange: affected routines; child_num: number of childen of the process) 19
3.2 Noise propagation of dependencies (red: noise source; orange: affected processes/point-to-point routines) 21
3.3 Implementation of MPI_Bcast using nonblocking point-to-point routines (red: noise source; orange: affected routines; child_num: number of childen of the process) 23
3.4 Implementation of MPI_Bcast in ADAPT (red: noise source; orange: affected routines; child_num: number of childen of the process) 27
3.5 Performance of MPI_Bcast with noise injection (16 processes, binary tree, 1 process per node, 4 MB message, Saturn) 35
3.6 Performance of MPI_Bcast and MPI_Reduce with noise injection (1K cores on Cori, numbers on bars represent performance slowdown caused by noise) . . 37
3.7 Performance of MPI_Bcast and MPI_Reduce with noise injection (1.5K cores on Stampede2, numbers on bars represent performance slowdown caused by noise) 38

4.1 MPI_Bcast with a binary tree (16 processes) 41
4.2 Process placement with a binary tree (16 processes, 2 nodes, 2 sockets per node, 4 cores per socket) 42
4.3 Process placement with a topology-aware tree (16 processes, 2 nodes, 2 sockets per node, 4 cores per socket) 42

4.4 Topology-aware communication tree of MPI_Bcast on a multi-core cluster (2 sockets per node, 4 cores per socket) 44
4.5 Rank reorder of MPI_Bcast on a multi-core cluster (2 sockets per node, 4 cores per socket) 46
4.6 Performance of topology-aware MPI_Bcast and MPI_Reduce (1K cores, Cori) 48
4.7 Performance of topology-aware MPI_Bcast and MPI_Reduce (1.5K cores, Stampede2) 48
4.8 Performance of MPI_Bcast and MPI_Reduce varies by message size (1K cores, Cori) 50
4.9 Performance of MPI_Bcast and MPI_Reduce varies by message size (1.5K cores, Stampede2) 50
4.10 Strong scalability of MPI_Bcast and MPI_Reduce varies by number of nodes (4 MB message, Cori) 52

5.1 Performance of Intra-node MPI_Bcast (20 processes, Saturn) 56
5.2 Performance of Intra-node MPI_Reduce (20 processes, Saturn) 57
5.3 Performance of Intra-node MPI_Allreduce (20 processes, Saturn) 58
5.4 The reduction phase of MPI_Reduce and MPI_Allreduce in the SOLO module 59
5.5 Memcpy bandwidth on Saturn 60
5.6 Design of MPI_Bcast in HAN 63
5.7 An Example of MPI_Bcast on two nodes 64
5.8 Cost of tasks (ib, sb, sbib) on NaCl (0 is the root), SM module 67
5.9 Cost of tasks (ib, sb, sbib) on NaCl (0 is the root), SOLO module 68
5.10 Overlap Rate of Concurrent Intra-node and Inter-node Collective Operations (6 nodes, 12 ppn, SM module) 70
5.11 Overlap Rate of Concurrent Intra-node and Inter-node Collective Operations (6 nodes, 12 ppn, SOLO module) 71
5.12 Cost of tasks on one node leader on NaCl (64 nodes, 12 ppn) 72
5.13 Performance of MPI_Bcast on 64 nodes with the combinations of different submodules on NaCl 74

5.14 Design of MPI_Allreduce in HAN 75
5.15 The overlap between *ib* and *ir* (0 is the root) 77
5.16 An Example of MPI_Allreduce on two nodes 78
5.17 MPI_Allreduce on 64 nodes with combination of different submodules on NaCl 80

6.1 Time of total searches of MPI_Bcast and MPI_Allreduce on Nacl 84
6.2 Performance of MPI_Bcast and MPI_Allreduce on 64 nodes with different configurations of sub-modules on NaCl 85
6.3 Performance MPI_Bcast (Shaheen II, 4096 processes) 88
6.4 Bandwidth of MPI point-to-point communication routines (Shaheen II, 4096 processes) 88
6.5 Performance of MPI_Bcast (Stampede2, 1536 processes) 89
6.6 Performance of MPI_Allreduce (Shaheen II, 4096 processes) 90
6.7 Performance of MPI_Allreduce (Stampede2, 1536 processes) 91
6.8 Performance of Horovod on Stampede2 93

Chapter 1

Introduction

1.1 Motivation

Computation modeling and simulation were introduced two decades ago to save the experimentation cost and speed up the research process by using computer systems. Due to the limitations of memory and computing power of a single computer, scientists use High-Performance Computing (HPC) systems to model complex problems and execute large scale simulations in parallel. HPC systems are widely used in many scientific research areas, such as biochemistry, chemistry, physics, and geography, and to satisfy the increasing computational need of these research areas, HPC systems continue to grow in scale and heterogeneity.

By equipping the co-processors and Graphics Processing Units (GPU), the computing power of HPC systems grows rapidly. However, compared to the fast-growing computing power, the speed of communication falls behind, which causes communication to become the primary bottlenecks in many applications. Thus, it is crucial for communication libraries to provide optimal performance to sustain parallel applications. The Message Passing Interface (MPI) [26] is the most widely used communication standard in the HPC community, providing various communication primitives to facilitate the development of HPC applications.

Collective operations are one of the communication primitives available through the MPI standard. These operations provide a means to perform communications between multiple processes in a variety of communication patterns. As found in previous studies [50, 8], collective operations are used in most MPI applications, and their cost becomes a major factor in determining the performance and scalability of MPI applications. Hence, to achieve good performance in MPI-based applications, MPI libraries must provide highly efficient collective operations. However, with the increasing scale and complexity of HPC systems, performance optimization becomes more challenging. Overall, there are three challenges preventing collective operations from reaching optimal performance in large HPC systems: propagation of noise, hardware heterogeneity and hierarchy, and parameter autotuning.

1.1.1 Propagation of Noise

In the early days, the term noise only refers to operating system noise, which is the interference experienced by applications due to activities, such as cache misses, hardware interrupts, and page faults, inside operating systems [9, 35, 23]. With the increase in system size, noise is extended from operating system noise to the delays from other sources, including fault tolerance [24, 42], in-situ analytics [46], and power management [27].

As suggested in previous research [35, 22], noise can dramatically slow down large-scale parallel applications. The main reason for the slowdown is the propagation of noise from one process to others through communications in between. Compared with point-to-point communication operations, which involve only two processes, collective operations can easily propagate noise from one process to all the processes that participated in the communication. Therefore, even a little noise could be propagated via collective operations, and then significantly decrease the performance of applications. Previous study [22] shows that 2.5% noise can drop the performance of some applications more than $4\times$ on 500 nodes and $18\times$ on 2500 nodes. Thus, it is crucial to identify the reason for noise propagation within collective operations and to provide noise-resistant collective operations.

Usually, a collective operation consists of many fine-grained MPI one-sided or two-sided communication routines. Carelessly handling the dependencies of these routines brings extra synchronizations, and the extra synchronizations would propagate noise from one process to the other processes. Therefore, to minimize the noise propagation and amplification, it is important to identify and minimize unnecessary synchronizations in collective operations.

1.1.2 Hardware Heterogeneity and Hierarchy

Another important challenge as we approach larger scale machines is the increasing resource heterogeneity and the more complex hardware hierarchy. A compute node on such heterogeneous systems usually contains multiple CPU sockets that are connected by high-speed inter-socket connections (e.g., Intel QPI or AMD Hyper-transport). Then, several compute nodes are coupled together through the high-performance interconnects and organized into racks, and finally the racks are combined into super-computers.

Benefiting from massive parallelism with low power consumption, HPC systems increasingly incorporate accelerators (GPU, Intel KNL or specialized FPGA). Hence, more and more applications, including traditional scientific applications [62, 65] and deep learning applications [63], start to adopt accelerators to boost their performance. However, embracing accelerators complicates the already complex architecture hierarchy, as accelerators are connected via PCI Express bus or, in some cases, NVLink for NVIDIA GPUs.

All these advancements cause a drastic increase in performance and capability differences between levels of the hardware hierarchy [44, 30]. The cost of the communications between processes greatly varies depending on the physical distance and the types of hardware between the processes. Thus, maintaining good communication performance requires a holistic integration of process placements and architecture capabilities. Recent advances in MPI collective implementations start to integrate hardware topology information into collective operations [44, 36, 30] to achieve better performance. However, the insufficient cooperation of the communications on different hardware levels (i.e., intra-socket, inter-socket, PCI bridges and inter-node levels) leads to sub-optimal overlapping of the communications. Also, the algorithms of collective operations are not adaptable to various network conditions. This calls for a collaborative approach, between multiple levels of collective algorithms, dedicated to holistically managing all levels of the hardware hierarchy.

1.1.3 Parameter Autotuning

Autotuning is a well-known technique used to automatically find the best parameters (algorithms, segment sizes, etc.) of collective operations. It is a crucial feature for any

MPI collective framework as its accuracy of the selection of the best parameters directly affects the performance of collective operations.

There exist two methods to perform autotuning. The first approach is searching through the possible configurations of a collective operation. A naive implementation of this approach is to do an exhaustive search of every possible configuration, which is the only guaranteed way to find the best configuration. This method works well on small machines; however, its huge search space limits its usage on modern large scale systems. To address this, some efforts have been made to reduce the search space with heuristics [59, 20]. However, with more heuristics, more assumptions are made to limit the search space, which could reduce the accuracy of autotuning.

The second approach is using cost models [20, 50] to estimate the time of collective operations and select configurations based on the estimations. Instead of directly measuring the performance of whole collective operations, this method only needs to benchmark a few key factors of systems, such as overhead, gap, bandwidth, and latency, which significantly reduces the time it takes for autotuning. However, as stated in [20, 50], cost models are not accurate enough to find the best configuration as they oversimplify modern heterogeneous HPC systems. Hockney [33], LogP [13], LogGP [5], and PLogP [38] are four widely used cost models to describe the communications in HPC systems, and they all assume the cost of MPI point-to-point communications between any two processes is the same. But this assumption is no longer valid on heterogeneous systems, where the cost of the point-to-point communications varies a lot based on the locations of the participating processes. SALAR [8] extends the LogGP model with different gaps (Gs) for different hardware to model a hierarchical MPI_Allreduce. But its gap is fixed for each level, which limits this model to large messages when transferring one segment of the entire message can saturate the network bandwidth.

Besides network heterogeneity, some other factors are not considered in these cost models, such as the congestion on a busy process and the shared resources of the communications on different hardware levels. The experiments in [32] suggest when one process communicates with multiple processes concurrently, the congestion on that process would drastically affect the overall communication performance. Previous research [7, 8, 17] assumes the communications on different levels such as inter- and intra-node are totally independent when modelling hierarchical collective operations. However, they are not perfectly overlapped because of the shared resources such as memory buses, which is discussed in chapter 5.

Overall, neither of these methods works well on modern HPC systems. The first approach can find the correct configurations, but it may take too much time on large scale machines; while the second approach dramatically reduces the search time, but it is not accurate because it misses a lot of factors. This calls for a new approach to perform parameter autotuning, which can reduce the search space and find the best configurations accurately.

1.2 Contributions

In this dissertation, I divide my contributions into three parts (noise-resistant collective operations, hierarchical collective operations, and autotuning of collective operations), with each part addressing one of the challenges introduced in the previous section.

1.2.1 Noise-Resistant Collective Operations

To alleviate noise propagation, I propose "ADAPT," a new event-driven collective framework in Open MPI, which uses the completion of non-blocking point-to-point routines within collective operations as events, and each event triggers a callback to allow the high-level logic to issue dependent point-to-point routines. With the help of the event-driven design, the synchronizations in collective operations are relaxed; and by relaxing synchronizations, the ADAPT framework offers more potential to absorb system noise instead of propagating or even amplifying it further. The detailed design and implementation are discussed in chapter 3.

1.2.2 Hierarchical Collective Operations

To fully utilize the different characteristics of the hardware on each level of the hardware hierarchy, and improve the performance of collective operations on modern HPC systems, I present two approaches to perform hierarchical collective operations, both utilizing the hierarchical structure of HPC systems to optimize collective communications.

The first approach is combining the ADAPT framework with a carefully built topology-aware tree. The event-driven design makes the ADAPT framework provides a lot of opportunities to concurrently communicate over the hardware on different levels of heterogeneous systems. This approach is introduced in chapter 4.

The downside of the first approach is that the ADAPT framework is based on MPI point-to-point communications, which makes it unable to fully utilize the characteristics of the hardware on each level of the hierarchy. For example, to transfer data from one process to another process in the same node, a point-to-point communication needs two memory copies, one from the source to a temporary buffer and one from the temporary buffer to the destination; while with an one-sided communication, only one memory copy is enough. To address this issue, I present another approach, "HAN," a flexible hierarchical collective framework in Open MPI in chapter 5. The HAN framework picks proper collective frameworks as submodules to utilize the hardware specification of each level and combines the collective operations from these submodules to perform hierarchical collective communication operations with a task-based design.

1.2.3 Autotuning of Collective Operations

As seen from section 1.1.3, neither of the two existing autotuning methods (exhaustive search, conventional cost model) works well independently; therefore, I design a new autotuning approach by fusing these two methods to ensure a fast and accurate result. I create a novel cost model based on benchmarking of several independent communication tasks and select the best configuration with the help of the cost model. Compared with the exhaustive search, since the benchmarking is performed on the tasks instead of a whole collective operation, this method can reduce the search space significantly. And compared with the conventional cost models, the new cost model is more accurate since it considers many more factors including:

- Different bandwidths on different levels;
- Changing gap with increasing message sizes;

- Congestion on a process;
- Overlap rate of the communications on different levels;

All of these factors can affect the performance of collective operations, but they are hard to model; while in the new autotuning approach, instead of modelling them, their influences are directly measured on the tasks to provide better estimations.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows:

- Chapter 2 presents the background of this research. It introduces the MPI standard, MPI collective communication operations, and one implementation of MPI standard - Open MPI. In this chapter, I also review some previous work related to noise propagation, hierarchical collective operations, and autotuning of collective operations.
- Chapter 3 first introduces a few state-of-the-art MPI implementations and shows how noise is propagated through the dependencies in such implementations. Then it presents and evaluates the design of a new collective framework, “ADAPT,” which adopts an event-driven design to relax the unnecessary dependencies and alleviate noise propagation.
- Chapter 4 shows the first approach I used to implement hierarchical collective operations. I extend the ADAPT framework with a topology-aware communication tree to utilize hardware topology information and maximize the concurrent communications on different levels.
- Chapter 5 presents my second approach to perform hierarchical collective operations. Compared with the first approach, this approach can better utilize the hardware capability on each level and is more flexible to adapt to hardware updates. In this chapter, I present ”HAN,” a new hierarchical autotuned collective framework in Open MPI, which selects the suitable homogeneous collective communication modules as

submodules for each hardware level, uses the collective operations from the submodules as tasks, and organizes these tasks to perform efficient hierarchical collective operations.

- Chapter 6 introduces a new way to perform autotuning in the HAN framework. The new approach merges two existing autotuning methods by using a novel cost model based on the costs of tasks. It can significantly reduce the search space to save the tuning time and provide accurate estimations.
- Chapter 7 concludes this dissertation and discusses some future directions.

Chapter 2

Background and Literature Review of Related Work

2.1 Overview

I discuss the background of this study in this chapter and review some related work. In section 2.2, I introduce the MPI standard, MPI collective communication operations, and the Open MPI library that is one implementation of the MPI standard and the foundation of my work. Then, in section 2.3, section 2.4, and section 2.5, I review previous work related to noise propagation in collective operations, hierarchical collective operations, and autotuning of collective operations, respectively.

2.2 MPI

MPI stands for Message Passing Interface, which defines a library interface to describe the communication in HPC systems. It is based on the message-passing parallel programming model, where data needs to be moved between the address spaces of processes. MPI was first introduced in 1993, and at that time it mainly focused on point-to-point communications. Later, more functionalities were added to the MPI standard, such as collective operations, remote-memory access operations, dynamic process creation, parallel I/O, etc. The latest version of the MPI standard (MPI-3.1) was published in June, 2015. In this dissertation, I focus on the collective communication operations in MPI.

2.2.1 MPI Collective Communication Operations

Collective communication operations are communication routines that involve multiple processes, in which all the participating processes need to call the function to execute the collective operation.

A key argument of collective functions is a communicator that defines the group or groups of the participating processes. Each process can belong to one or multiple communicators, and it has an ID (rank) to identify itself in each communicator. The communicators can be divided into two types: intra-communicators and inter-communicators, based on the number of groups in a communicator. An intra-communicator contains a single group, while an inter-communicator has a pair of groups. In this dissertation, I only consider the collective operations on the intra-communicators. However, the idea of my framework can be applicable to optimize other types of collective operations.

The common collective communication operations can be divided into three types:

1. One-to-all: this type of collective operation transfers data from one process to all the processes in a communicator. It includes:

   MPI_Bcast. This operation broadcasts a message from the root to all the processes in a communicator. After this operation, all the processes get the whole message.

   MPI_Scatter. This operation scatters a message from the root to all the processes in a communicator. Suppose the number of processes in a communicator is n, then this message is split into n equal parts, and the process i gets the i-th part.

2. All-to-one: this type of collective operation transfers data from all the processes in a communicator to one process. It includes:

   MPI_Reduce. This operation performs a global reduce operation (maximum, minimum, sum, average, etc.), and returns the result of the reduction to the root process. It is the inverse operation to MPI_Bcast.

MPI_Gather. This operation gathers messages from all the processes in a communicator to the root process. The root process receives these messages and stores them in the same order as the ranks of the processes. This operation is the inverse operation to MPI_Scatter.

3. All-to-all: this type of collective operation transfers data from all the processes in a communicator and stores the result back to all the processes.

   MPI_Barrier. This is a special kind of collective operation. It blocks the caller processes until all the processes in a communicator enter the barrier function. This operation is not used for message transmission, but synchronization.

   MPI_Allgather. This operation is similar to MPI_Gather. It gathers messages from all the processes in a communicator and returns the result to all the processes.

   MPI_Allreduce. This operation is similar to MPI_Reduce. It performs a global reduce operation and returns the result of the reduction to all the processes.

   MPI_Alltoall. In this operation, the message is divided into n equal segments, where n is the number of participating processes. It transfers the i-th segment on process j to the j-th segment on process i.

In the MPI collective operations introduced before, each process sends or receives the same amount of data. This works for many cases, but some applications may require each process in the MPI collective operation to transfer a different amount of data. Hence, the variable-size-input or v-version MPI collectives are introduced, such as MPI_Gatherv, MPI_Scatterv, MPI_Allgatherv, and MPI_Alltoallv.

While this study is generic and can be applied to other collective operations, I use MPI_Bcast and MPI_Allreduce as examples to show the design of my collective frameworks.

2.2.2 The Open MPI Library

The Open MPI library [31] is the foundation of my work, and all my collective operation frameworks introduced in this dissertation are implemented in Open MPI.

Open MPI is an open source implementation of the MPI standard. It starts from a merger of LAM/MPI [57], LA-MPI [6], FT-MPI [18], and PACX-MPI [28] code bases. It is designed, developed, and maintained by an active community of volunteers that come from both academia and industry. To make the Open MPI library well organized and easy to extend, the community adopts a component architecture called the Modular Component Architecture (MCA) to design Open MPI. There are three main components in Open MPI:

- Open MPI component (OMPI). This component contains the implementations of MPI functions.
- Open Run Time Environment (ORTE). This component supports different back-end run-time systems. Recently, it is replaced by PRRTE.
- Open Portable Access Layer (OPAL). This component glues the code of OMPI and ORTE.

The collective operation frameworks belong to OMPI, and they are located at OMPI/COLL. The following is a partial list of existing MPI collective frameworks in Open MPI:

- Base - the basic collective framework in Open MPI, containing the homogeneous implementations of all the collective operations. For each collective operation, it provides multiple algorithms.
- Tuned - an autotuned framework, which can automatically select the algorithm and segment size in the Base framework.
- Libnbc - a collective framework that supports non-blocking collective operations.
- SM - a specialized intra-node collective framework, which can utilize the shared memory space in a node to reduce the number of memory copies.
- CUDA - a framework that supports collective operations on GPU data.

Some of these frameworks are used as submodules in my design of hierarchical collective operations, which is discussed in chapter 5.

2.3 Noise Propagation

As presented in the introduction, with the increasing scale of HPC systems and more potential sources of noise, finding a way to alleviate the effects of noise becomes a crucial problem for large-scale parallel applications. There are two approaches to alleviate the effects of noise in previous research. One way is to eliminate noise from the source, and another way is to alleviate the propagation of noise.

2.3.1 Eliminate Noise from the Source

The first method for eliminating noise is to identify the source of the noise and remove the noise from the source. For instance, noise introduced by operating systems can be reduced by system designers. In [9], Beckman et al. investigate the noise introduced by operating systems and demonstrate that synchronizing the noise can significantly reduce its negative influence; and in [66], Yoshii et al. present a method that uses extremely large memory pages available on PowerPC CPU to reduce system noise. With the increasing scale of HPC systems, noise is extended from operating system noise to delays from other sources. In [67], Zheng et al. minimize noise between a simulation and an in-situ analytics by using fine-grained scheduling to get idle resources.

This method works well for certain types of noise; however, there exist many types of noise that cannot be eliminated from the source, and hence some research try to alleviate noise propagation.

2.3.2 Alleviate Noise Propagation

Typically, local noise is very small; however, it can greatly hurt the performance of large scale applications. As demonstrated in [9] and [22], the main reason for significant slowdown from noise is that the local noise is propagated and even amplified through collective operations. Therefore, the second method to reduce the negative effects of noise is to alleviate noise propagation in collective operations. Vishnoi et al. [60] show how system noise impacts the performance of collective operations, and Widener et al. [64] introduce how non-blocking collective operations have the potential to mitigate certain types of noise.

Despite in-depth research on the causes and impacts of noise mentioned above, not many previous work focus on reducing the noise propagation. My work is the first to focus on the implementation of noise resistant MPI collective operations.

2.4 Hierarchical Collective Operations

2.4.1 Explore More Levels

To take advantage of the communication differences in the hardware hierarchy, some research manages to minimize the data transfers on the slow communication channels by grouping the processes based on their locations. MagPie [39] optimizes collective operations for wide area systems, where the processes are grouped by clusters; while MPICH2 [68] groups the processes by compute nodes to limit the number of inter-node communications. Later, the groups are further divided to explore more levels of the hardware hierarchy [37]. In MVAPICH2 [36, 58], the researchers add one more level to group the processes based on the network switch information.

Some other research focuses on the strategies to select the leader of each group. Parsons et al. [49] select the leader dynamically to overcome the imbalanced process arrival times, and Bayatpour et al. [7] create multiple leaders to better explore the parallelism in networks for MPI_Allreduce. These methods provide better performance compared to the isotropic approaches that assume the cost for any pair of processes is equal [30], but since they are not able to overlap the communications on different levels, their performance for big messages would be sub-optimal.

2.4.2 Communication Overlap

Some research manages to overlap the communications on two levels, intra-node and inter-node. HierKNEM [45] makes intra-node communications asynchronous by offloading its intra-node communications with KNEM [29]; while SALaR [8] implements an inter-node allreduce with non-blocking one-sided communication routines to make its inter-node communications asynchronous.

The two new approaches presented in this dissertation are able to overlap the communications on different levels. With a carefully built topology-aware tree, the ADAPT framework supports three levels: inter-node, inter-socket, and intra-socket. Since the ADAPT framework uses non-blocking point-to-point routines on all the levels, it can overlap the communications on the three levels. As for the HAN framework, it currently supports two levels: inter-node and intra-node. This framework makes its inter-node communications asynchronous by using non-blocking collective operations.

Cheetah [30] uses a Directed Acyclic Graph (DAG) to describe hierarchical collective operations, which is similar to the HAN framework. However, my framework provides two major advantages over Cheetah. First, the HAN framework has a pipelining mechanism that can overlap the communications on different levels, which is missed in Cheetah. Second, Cheetah lacks an autotuning component. Without an autotuning component, its best performance are difficult to be achieved on a given machine.

2.5 Autotuning of Collective Operations

There are many configurations, such as algorithms and segment sizes, in any framework for collective operations. As suggested in previous research [59, 19], choosing the right configurations is important for the performance of collective operations.

In previous research, many methods are created to tune a collective framework to find the right configurations automatically. These methods can be divided into two categories: offline tuning and online tuning. In offline tuning, autotuning happens before executing applications; while in online tuning, collective frameworks tune themselves while applications are running.

2.5.1 Offline Tuning

In [59], Vadhiyar et al. notice collective operations may not give good performance in all situations. Hence, they perform an exhaustive search to find the best arguments for every possible case and use the results to tune collective operations. It also provides some heuristic ideas and multiple gradient descent methods to limit the search space. These heuristics are from different angles than my approach and can be combined with mine to further reduce the testing time as discussed in chapter 6. The tuned [19] module is the current default collective framework in Open MPI, its default decisions are made by running benchmarks on a cluster of AMD64 processors with Gigabit Ethernet interconnects. Since HPC systems have changed drastically, its default decisions are not optimized for modern hardware.

With the increasing scale of HPC systems, the search space of the exhaustive search approach grows exponentially, making this approach unrealistic and resulting in researchers trying to use cost models to guide the autotuning processes. In [50], Pješivac-Grbović et al. try to use multiple models to estimate the cost of MPI_Bcast. However, as the authors point out in the paper, the models are not accurate enough to tune collective operations. SALaR [8] improves the LogGP model with different gaps for different levels. However, even though this model is more accurate than previous cost models for hierarchical collective operations, it still can not find the best configuration directly. In the SALaR paper, the authors only use the cost model to provide a starting point for its online tuning. Eller et al. [17] further improve the accuracy of a postal model of MPI_Allreduce by considering network congestion, network distance, communication and computation overlap, and process mappings. This model's assumption of the perfect overlap of communications on different levels and only supporting one algorithm make it not suitable for autotuning.

2.5.2 Online Tuning

Online tuning is another way to figure out the best configuration. It times collective operations and changes the decision accordingly while MPI applications are running. STAR-MPI [21] maintains a set of algorithms for each MPI collective operation. As an application executes, STAR-MPI measures the performance of the algorithms and dynamically selects the best algorithm for the application at runtime. SALaR [8] implements a two-level hierarchical MPI_Allreduce with a pipelined design. To find the best segment size in the pipelined design, it adopts online tuning. SALaR keeps the latency, message size, and segment size of previous calls to MPI_Allreduce for each message range and uses a heuristic-based adaptive strategy to find the best segment size for the new MPI_Allreduce.

In the two previous research, based on the initial guess, the time to find the best configuration is uncertain; and both inevitably bring overhead, including the cost of timing and maintaining the decision matrix online. These downsides can hurt the performance of collective operations, which limits the application of this approach to the general case, and that is why I choose to use offline tuning approach in this dissertation.

Chapter 3

Noise Resistant Collective Operations

3.1 Overview

With the increasing scale of HPC systems, there are more and more sources of noise that can impact the performance of applications. Even though local noise often causes very little delay per process, such delays can affect the overall performance of applications significantly when the noise is propagated to other processes through the communications between the processes [35].

Compared to point-to-point communication routines, collective communications are more easily affected by noise for two reasons. First, noise propagation increases with the number of participants in collective communications. Since the number of processes is determined by application developers, limiting the number of processes is not an option to reduce noise propagation. Second, there are many dependencies in the implementations of collective operations that allow noise to propagate.

In this chapter, I first identify the dependencies in the implementations of collectives operations in mainstream MPI libraries and analyze how these dependencies propagate noise in section 3.2. Then, I introduce the ADAPT collective operations framework in section 3.3, which adopts an event-driven design to relax dependencies. Finally, I evaluate the performance impact of noise on the ADAPT framework and other MPI libraries on three clusters in section 3.4.

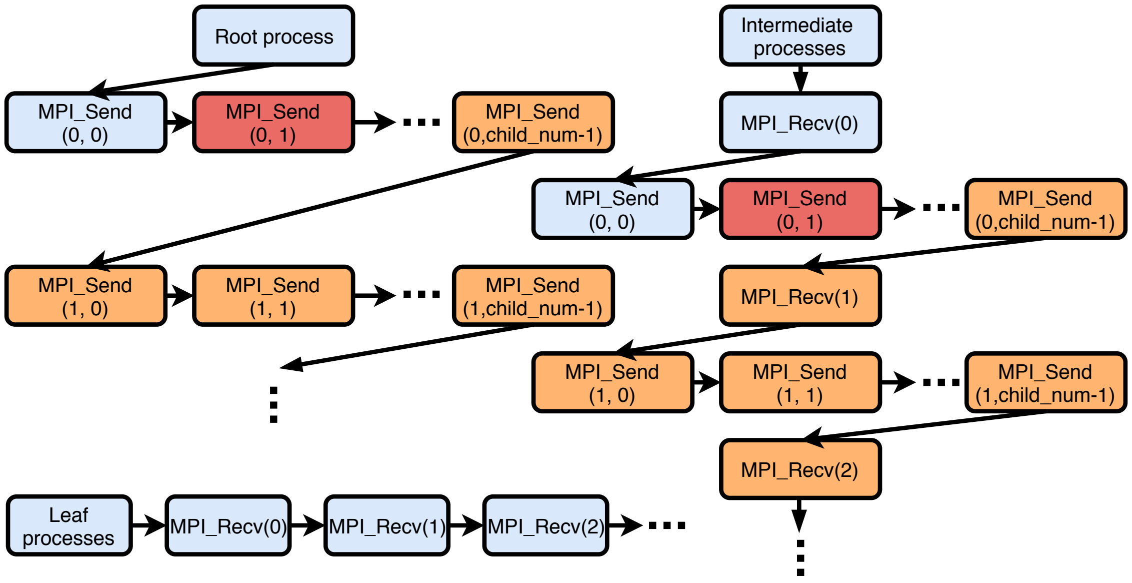


Figure 3.1: Implementation of MPI_Bcast using blocking point-to-point routines (red: noise source; orange: affected routines; child_num: number of childen of the process)

3.2 Existing Implementations

In general, the collective operations implemented in major MPI libraries are based on point-to-point communication routines, either blocking or nonblocking. Carelessly managing these point-to-point routines introduces unnecessary dependencies between them, and such dependencies bring synchronizations between otherwise independent point-to-point routines. As discussed in [35], noise on one process can delay other processes. For example, noise on one process delays its point-to-point routines, and then delayed routines propagate noise to other point-to-point routines via dependencies between them. Later, all the delayed point-to-point routines stall other participating processes. In this fashion, noise is propagated from a single process to the others and slows down the performance of entire collectives.

I analyze two implementations of MPI_Bcast, one using blocking point-to-point routines and one using nonblocking point-to-point routines, to identify hidden dependencies and highlight their noise propagation patterns.

3.2.1 Collectives Using Blocking Point-to-Point Routines

Figure 3.1 presents a pipelined implementation of MPI_Bcast using blocking point-to-point routines, which can support any tree-based algorithms. In the figure, MPI_Send(x,y) means sending segment x to child y, and MPI_Recv(x) means receiving segment x from the parent. With pipelining, big messages are divided into several segments and propagated in order.

In this implementation, the root process issues an MPI_Send for each of its children to transfer a segment. After they are finished, the same procedure applies to the following segments. Intermediate processes post an MPI_Recv to receive a segment from their parent and then issue multiple MPI_Sends to send the received segment to their children. After these MPI_Sends are done, they start to receive the next segment until all segments are processed. Leaf processes work similarly to intermediate processes without sending received segments. Algorithm 1 shows the pseudocode of this implementation. In the pseudocode, P means the current process, *seg_num* means the number of segments, and *child_num* means the number of children of current process.

Algorithm 1: MPI_Bcast using blocking point-to-point communication

```
   Input: MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root,
          MPI_Comm comm)
1  if $P$ is root then
2      for $i \leftarrow 0$ to $seg\_num - 1$ do
3          for $j \leftarrow 0$ to $child\_num - 1$ do
4              MPI_Send segment i to its child j;
5  else if $P$ is leaf then
6      for $i \leftarrow 0$ to $seg\_num - 1$ do
7          MPI_Recv segment i from its parent;
8  else
9      for $i \leftarrow 0$ to $seg\_num - 1$ do
10         MPI_Recv segment i from its parent;
11         for $j \leftarrow 0$ to $child\_num - 1$ do
12             MPI_Send segment i-1 to its child j;
```

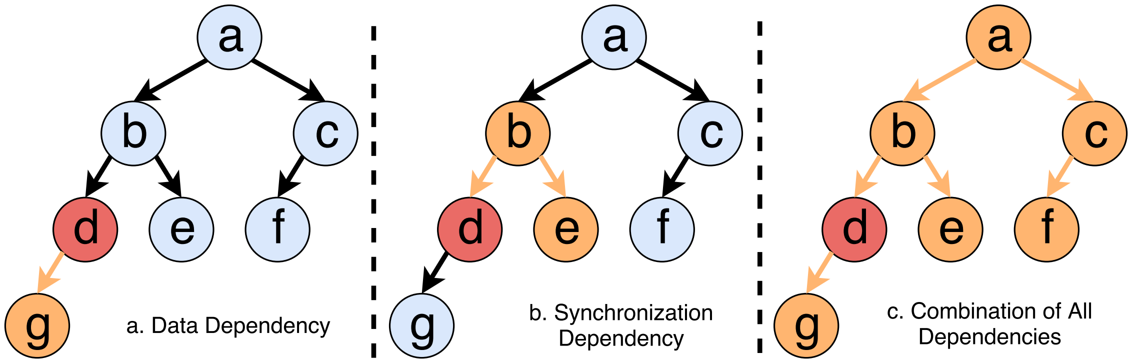

a
b
c
d
e
f
g
a. Data Dependency
b. Synchronization Dependency
c. Combination of All Dependencies

Figure 3.2: Noise propagation of dependencies
(red: noise source; orange: affected processes/point-to-point routines)

Since a blocking point-to-point communication routine involve synchronizations like handshakes between the sender and the receiver, noise on any of these two processes can slow down the blocking point-to-point routine and further delay the process on the other side. When there are dependencies between the point-to-point routines, noise can be propagated from one to the others, resulting in a slowdown of the entire collectives. In the implementations of MPI_Bcast using blocking point-to-point routines, I identify two kinds of dependencies, which can propagate noise significantly:

- **Data Dependency**. If the input data of some point-to-point routines depends on the output data of other point-to-point routines, then there is a data dependency between them. Thus, they have to be executed in order to get the correct results. In the implementations of MPI_Bcast using blocking point-to-point routines, intermediate processes have to receive one segment before sending it to their children. As in figure 3.1, MPI_Recv(i) must occur before MPI_Send(i, m) ($m \in [\mathit{0}, \mathit{child\_num} - \mathit{1}]$) for any segment i. This dependency is necessary for the correctness of a MPI_Bcast operation. With data dependency, noise on intermediate processes can be propagated to all their children. Figure 3.2.a represents the noise propagation pattern caused by data dependency with a binomial tree MPI_Bcast. If the noise on process d delays the MPI_Recv from b to d, then following MPI_Send from d to g is delayed, leading to the delay of g.

- **Synchronization Dependency**. This kind of dependency is caused by the synchronizations between point-to-point routines. A blocking point-to-point routine waits until the operation is done, which naturally brings a hidden synchronization. Such synchronization leads to a dependency between a blocking point-to-point routine and all the future routines, which is called Synchronization Dependency.

  This dependency brings unnecessary ordering of the routines and can propagate noise from one process to other processes. As in figure 3.1, root and intermediate processes always send a segment to child m before child n, for all $m < n$ (i.e., MPI_Send(0,0)) always before MPI_Send(0,1), even though there is no data dependency between them. Thus, if MPI_Send(0, 0) is delayed (marked red), all the following MPI_Sends and MPI_Recvs are affected (marked orange). Figure 3.2.b presents how noise is propagated from one process to other processes as a result of this kind of dependency. If the noise on process d delays the MPI_Recv of segment 0 from b to d (MPI_Recv(0)) on process d), then MPI_Send(0, d) on process b is also delayed since noise can be propagated through blocking point-to-point routines, and thus the parent of process d, process b, is delayed. Later, because of the synchronization dependency, the delay of MPI_Send(0, d) on process b affects the MPI_Send(0, e) on process b, resulting in the delay of process e, the sibling of process d. Therefore, a delayed process can affect its siblings and parent in this case.

Based on the analysis above, both types of dependencies can propagate noise. Via data dependency, noise on a process is unavoidably propagated to its children and further to the grandchildren, and via the unnecessary synchronization dependency, noise on one process is propagated to its parent and siblings. With the combination of these two types of dependencies, after a few iterations, the noise on one process can be propagated to the grandchildren, grandparents, and descendants of grandparents, and eventually, to all the processes (figure 3.2.c). Therefore, this implementations of MPI_Bcast using blocking point-to-point routines is able to amplify noise.

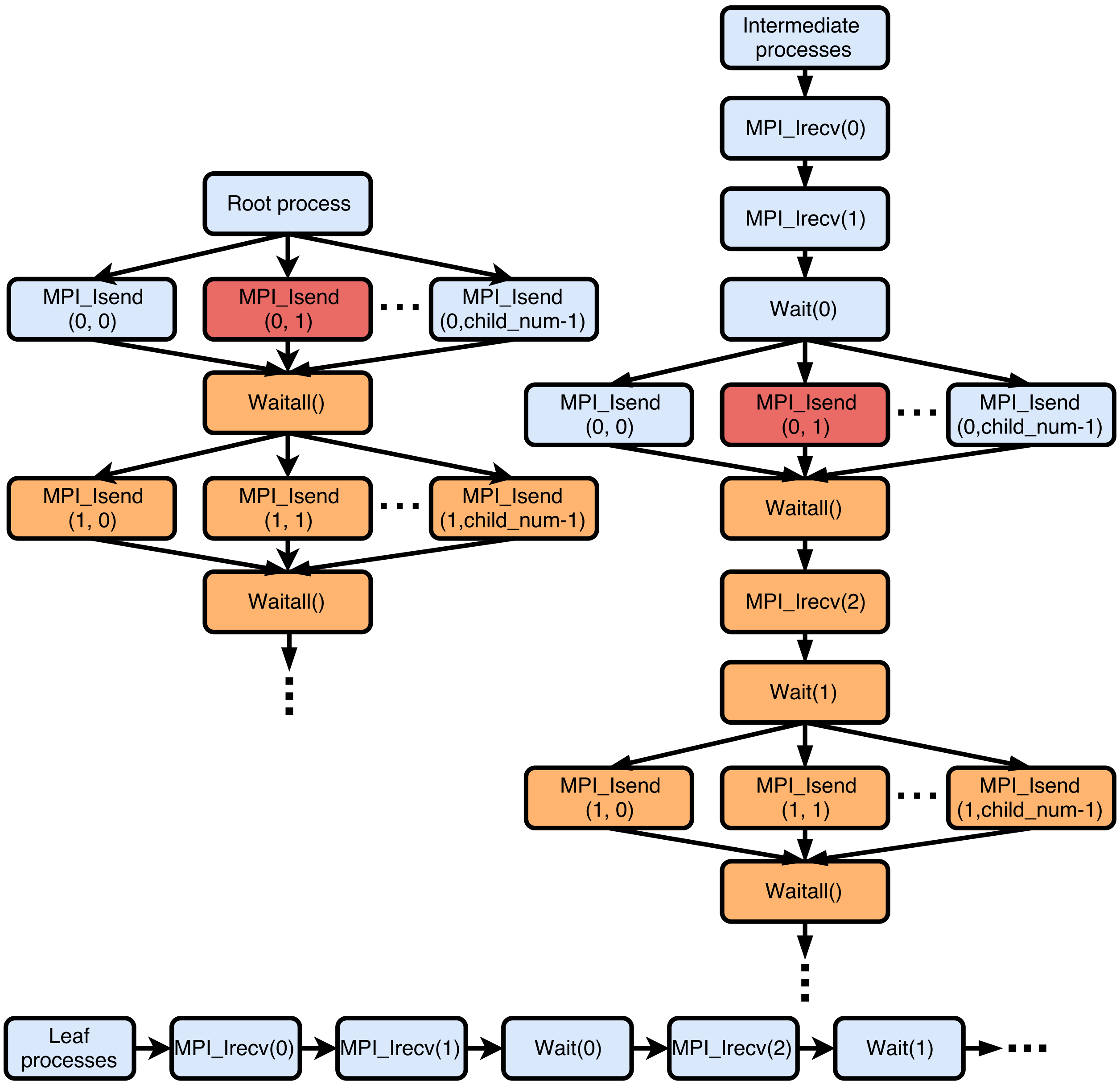


Figure 3.3: Implementation of MPI_Bcast using nonblocking point-to-point routines (red: noise source; orange: affected routines; child_num: number of childen of the process)

3.2.2 Collectives Using Nonblocking Point-to-Point Routines

An improvement over the previous implementation is using nonblocking point-to-point routines (MPI_Isend, MPI_Irecv) instead of the blocking ones. Figure 3.3 presents a pipelined implementation of MPI_Bcast using nonblocking point-to-point routines, and this implementation is the default implementation in Open MPI. In the figure, MPI_Isend(x,y) means sending segment x to child y, MPI_Irecv(x) means receiving segment x from the parent, and Wait(x) means waiting for segment x.

In this implementation, the root process issues multiple MPI_Isends to send a segment to all of its children and uses Waitall to wait for all these MPI_Isends to finish. After these MPI_Isends are completed, the root process starts to send the next segment. Leaf processes post two MPI_Irecvs for the first two segments, but only waits for the first segment. When it receives the first segment, it posts an MPI_Irecv for the next segment and waits for the second segment. Intermediate processes behave similarly to leaf processes except that they need to send the received segment to its children in the same fashion as the root process. The reason for the non-root processes post two MPI_Irecvs instead of one is to handle out of order segments. Pseudocode of this implementation is in algorithm 2.

Unlike blocking point-to-point communication routines, nonblocking point-to-point routines are more noise resistant. In a nonblocking point-to-point routine, if one process is delayed, the process on the other side is still able to progress other nonblocking routines without hanging. Thus, in most cases, noise on one process is less likely to be propagated to the other process via nonblocking point-to-point routines [35], except when the other process has nothing to work but waits for the delayed nonblocking point-to-point communication. Even though a nonblocking point-to-point routine has a higher potential to absorb noise, in the implementation of MPI_Bcast using nonblocking routines, there are still dependencies that can propagate noise. The following describes the two dependencies and noise propagation patterns in this implementation:

- **Data Dependency**. It is the same as the previous implementation using blocking point-to-point routines, and this dependency is required for the correctness of a

Algorithm 2: MPI_Bcast using nonblocking point-to-point communication

```
Input: MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root,
       MPI_Comm comm)
1  if P is root then
2      for $i \leftarrow 0$ to $seg\_num - 1$ do
3          for $j \leftarrow 0$ to $child\_num - 1$ do
4              MPI_Isend segment i to its child j;
5          wait for all MPI_Isends;
6  else if P is leaf then
7      MPI_Irecv segment 0 from its parent;
8      for $i \leftarrow 1$ to $seg\_num - 1$ do
9          MPI_Irecv segment i from its parent;
10         wait for previous segment i-1 to arrive;
11     wait for the last segment to arrive;
12 else
13     MPI_Irecv segment 0 from its parent;
14     for $i \leftarrow 1$ to $seg\_num - 1$ do
15         MPI_Irecv segment i from its parent;
16         wait for previous segment i-1 to arrive;
17         for $j \leftarrow 0$ to $child\_num - 1$ do
18             MPI_Isend segment i-1 to its child j;
19         wait for all MPI_Isends;
20     wait for the last segment to arrive;
21     for $j \leftarrow 0$ to $child\_num - 1$ do
22         MPI_Isend of the last segment to its child j;
23     wait for all MPI_Isends;
```

MPI_Bcast operation. Therefore, like the previous implementation, the noise on intermediate processes can be propagated to all their children with data dependency.

- **Synchronization Dependency**. As seen in figure 3.3, by using MPI_Isends, data movements from one process to all of its children become independent, and they can be progressed in any order by MPI's progress engine. However, the Waitalls and the Waits act as synchronizations that order the point-to-point routines between them. Thus, any delays to the Waitalls and the Waits can affect the following routines, which causes this type of dependency can propagate noise to a process's siblings and parent.

For example, in figure 3.2, if the noise on process d delays the MPI_Irecv of segment 0 from b to d (MPI_Irecv(0) on process d), then process b is not delayed directly since b can still progress other MPI_Isend, such as MPI_Isend(0, e) on process b. However, because of the Waitall, process b can be affected if all other nonblocking point-to-point routines are completed, except for the delayed one. In this case, the Waitall on process b only waits for the delayed point-to-point routine, and hence, process b is delayed. Later, because of synchronization dependency, the delay of Waitall on process b affects the following MPI_Isend(1, e) on process b, resulting in the delay of Wait(1) on process e. Thus, process e, the sibling of process d, is also delayed.

With the combination of data dependency and synchronization dependency, noise can be propagated to all the processes. Compared with the blocking point-to-point implementation discussed before, this implementation of MPI_Bcast is more tolerant to noise since nonblocking routines offer out-of-order executions, instead of waiting for the delayed point-to-point routines. However, the Waitall and the Wait in the nonblocking version still bring heavy synchronizations, and thus this nonblocking version is not sufficient to absorb noise and minimize noise propagation.

3.3 ADAPT: Event-Driven Design

3.3.1 Implementation of ADAPT

In this section, to better exploit the available parallelisms in collective operations and minimize noise propagation, I present the ADAPT collective communication framework. The key idea of the ADAPT framework is to design collective communications operations with events and callbacks, which eliminates the need to wait for point-to-point routines to complete. This type of programming model is called "event-driven."

Event-driven programming is a long-existing programming model, and it is used to solve various problems, including reducing the memory overhead in embedded systems [16], achieving high throughput in server applications [48] and forming service objects across mobile networks [14]. This model is also used to handle I/O operations (event-driven

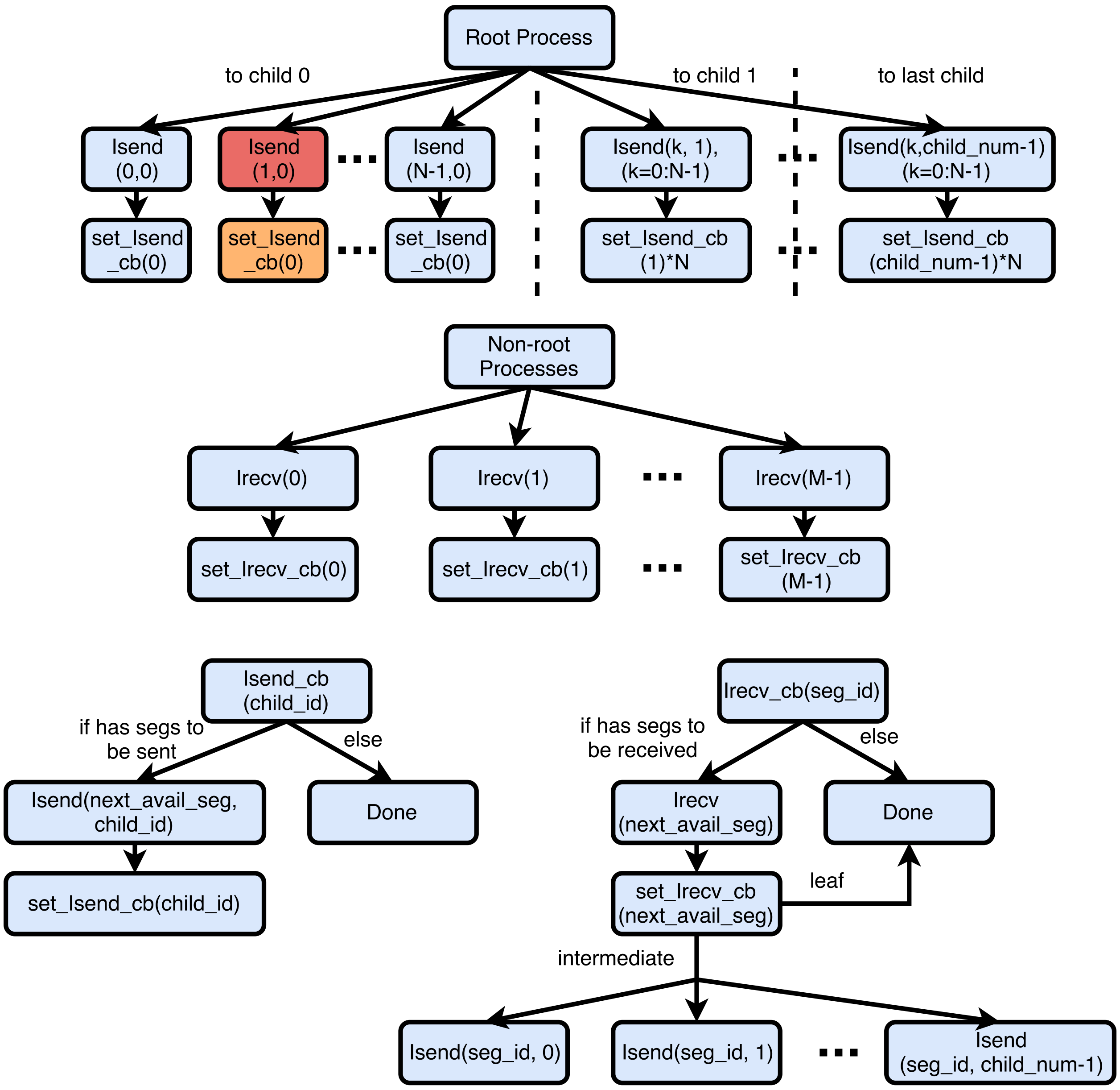


Figure 3.4: Implementation of MPI_Bcast in ADAPT
(red: noise source; orange: affected routines; child_num: number of childen of the process)

I/O [10, 15]). Normally, I/O operations are extremely slow compared to the processing of data. Therefore, the CPU's computing power is wasted if it is blocked to wait for I/O operations. Alternatively, with the event-driven design, instead of waiting for I/O operations, CPU can continues to work on other jobs until it gets a notification of the completion of an I/O operation.

In a typical event-driven program, there is an event loop to detect events, and when an event occurs, the corresponding callback is triggered. In this way, the execution flow of a program is determined by events and their callbacks. To implement events and callbacks, the ADAPT framework is deeply integrated with the communication engine in Open MPI. The completion of a nonblocking point-to-point routine is used as an event, which triggers a callback contains a detailed analysis of the state of the collective algorithm, and if necessary, the process posts new point-to-point routines to start the following data movements. One thing worth mentioning is that the nonblocking point-to-point routines, where callbacks are attached, are at a lower level than MPI_Isend/MPI_Irecv (shown as "Isend/Irecv" in the following) since MPI_Isend/MPI_Irecv does not support callbacks. Instead of waiting for each nonblocking point-to-point routine as in the previous implementation, I create a request for each collective operation and do not mark it as complete until the collective operation is finished. Therefore, Open MPI's progress engine keeps progressing all the nonblocking point-to-point routines until this request is completed.

The implementation of the MPI_Bcast algorithm in the ADAPT framework is shown in figure 3.4, and the pseudocode is presented in algorithm 3, algorithm 4 and algorithm 5. Following the event-driven pattern, all segments are propagated to all processes via a series of the Isends/Irecvs and their callbacks.

- Root: the root process posts N Isends to send the first N segments to each child, then uses set_Isend_cb to attach a callback to each of these Isends. When any Isend is completed, the corresponding Isend_cb is called to post another Isend for sending the next available segment.

- Non-root: a non-root process posts M Irecvs to receive the first M segments from its parents and attaches callbacks to these Irecvs with set_Irecv_cb. When any Irecv is completed, the corresponding Irecv_cb is called to post another Irecv for receiving the next available segment. If the process is an intermediate process, besides receiving the next available segment, it posts multiple Isends to send the received segment to its children in Irecv_cb.

Algorithm 3: ADAPT MPI_Bcast Algorithm

```
   Input: MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root,
          MPI_Comm comm)
1  create request;
2  create recved_segs_num;
3  create recv_array[segs_num];
4  create send_array[child_num];
5  if P is root then
6      recved_segs_num ← segs_num;
7      for $i \leftarrow 0$ to $seg\_num - 1$ do
8          recv_array[i] ← i;
9      for $j \leftarrow 0$ to $child\_num - 1$ do
10         send_array[i] ← N;
11     for $i \leftarrow 0$ to $N - 1$ do
12         for $j \leftarrow 0$ to $child\_num - 1$ do
13             Isend segment i to its child j and attach send_cb(j) to Isend;
14 else
15     recved_segs_num ← 0;
16     for $i \leftarrow 0$ to $seg\_num - 1$ do
17         recv_array[i] ← 0;
18     for $j \leftarrow 0$ to $child\_num - 1$ do
19         send_array[i] ← 0;
20     for $i \leftarrow 0$ to $M - 1$ do
21         Irecv segment i from its parent and attach recv_cb(i) to Irecv;
22 wait(request);
```

In the ADAPT framework, I issue N Isends to a single child and M Irecvs from the parent for multiple segments simultaneously to maximize the usage of the network resources (discussed in the next chapter 4) and absorb noise (discussed in the next section 3.3.2).

Algorithm 4: ADAPT MPI_Bcast Algorithm (send_cb)

```
   Input: send_cb(int child_id)
1  if send_array[child_id] < recved_segs_num then
2      Isend segment recv_array[send_array[child_id]] to the same child and attach
         send_cb(child_id) to Isend;
3      send_array[child_id]++;
4  if P is root and has sent all the segments then
5      complete(request);
```

Algorithm 5: ADAPT MPI_Bcast Algorithm (recv_cb)

```
   Input: recv_cb(int seg_id)
1  recved_segs_num++;
2  recv_array[recved_segs_num-1] ← seg_id;
3  if recved_segs_num + M - 1 < seg_num then
4      Irecv segment recved_segs_num + M - 1 from the parent and attach
         recv_cb(recved_segs_num + M - 1) to Irecv;
5  for $i \leftarrow 0$ to $child\_num - 1$ do
6      if recved_segs_num-1 = send_array[i] then
7          Isend segment seg_id to child i and attach send_cb(i) to Isend;
8          send_array[i]++;
9  if P is not root and has received all the segments then
10     complete(request);
```

Usually, M is set to be larger than N. This is because a problem of matching an Isend (of a segment) to a corresponding Irecv. If the segment arrives on the receiver side before the receiver posts a corresponding Irecv, the segment will be considered "unexpected." In this problem, MPI needs to store it into a temporary buffer and match it later when the receiver posts the corresponding Irecv. This problem introduces significant latency, as the procedure requires memory allocation and data copying; thus, it is very important to ensure an Irecv is always posted before the arrival of its corresponding Isend. To address this issue, I need to make sure M is bigger than N to minimize the chance of unexpected segments.

3.3.2 Analysis of Dependencies in ADAPT

As discussed in section 3.2, in the existing implementations of MPI_Bcast, there are two types of dependencies: data dependency and synchronization dependency. In this section, I demonstrate how the ADAPT framework relaxes synchronization dependencies and minimizes noise propagation by making every segment and every child independent of each other using the event-driven design.

- **Data Dependency**. In an MPI_Bcast implementation, data dependency means any process needs to receive the data before starting to send the data to its children. ADAPT has this kind of dependency as the same as the previous implementations, and this dependency is necessary for the correctness of a MPI_Bcast operation.

- **Synchronization Dependency**. In the ADAPT framework, the completion of a nonblocking point-to-point routine triggers a callback, and in the callback, the process may post another nonblocking point-to-point routine. For example, on the root process, Isend(N, 0) can only be issued after the earliest one of Isend(i, 0) ($i \in [0, N-1]$) is completed. This leads to a synchronization dependency between these two point-to-point routines. However, this synchronization dependency can hardly propagate noise because of the following two reasons.

  First, in the ADAPT framework, segments can be handled in any order (**segment independence**). I use the root process as an example to show the segment independence. At the beginning of a MPI_Bcast operation, all segments are put into a virtual "segment pool." The root process then posts N Isends to send the first N segments to child 0 (Isend(i, 0) ($i \in [0, N-1]$)). If any of them is done, its callback (Isend_cb) issues another Isend to send the next available segment of the segment pool. Thus, there are always N concurrent Isends between the root and each child. If any Isend is delayed, segments can be re-balanced to other Isends. Therefore, the delay of one segment can hardly delay the communication of other segments between the root and one child; and thus, noise can be limited. Also, the receiving of the next segment is decoupled with the sending of the current segment, which prevents the noise from being propagated to a process's grandparent.

Second, each child can transfer data segments independently from each other (**child independence**). In existing implementations of MPI_Bcast, a process always waits for a segment to be sent to all its children before transferring the next segment. While in the ADAPT framework, every child keeps its own state and transfers data segments independently. In this way, noise cannot be propagated to a process's siblings. In a word, segment independence absorbs the noise magnitude and child independence limits the range of noise propagation, and thus the ADAPT framework can reduce noise propagation.

Based on the above analysis, I conclude that, benefiting from the event-driven design, the ADAPT framework relaxes synchronization dependencies and minimizes noise propagation.

3.3.3 Extend ADAPT to Other Collective Operations

From the three implementations of tree-based MPI_Bcast mentioned in section 3.2 and section 3.3, I notice that there is a common communication pattern: a process sends data to its children or receives data from its parents, which is the **basic building block** of MPI collective operations.

Algorithm 6: Blocking point-to-point Implementation

```
1 for $i \leftarrow 0$ to $k$ do
2     MPI_Send($i$)/MPI_Recv($i$);
```

In the implementation using blocking point-to-point routines, the basic building block can be shown as algorithm 6, where k is the number of needed point-to-point routines. A similar pattern appears in the implementation of collective operations in MPICH and MVAPICH.

In the implementation using nonblocking point-to-point routines, the basic building block becomes algorithm 7; this pattern exists in MVAPICH and Open MPI. Compared with the previous implementation, it provides more parallelism by adopting nonblocking point-to-point routines, such as MPI_Isend/MPI_Irecv.

Algorithm 7: Nonblocking point-to-point Implementation

```
1 for $i \leftarrow 0$ to $k$ do
2     MPI_Isend($i$)/MPI_Irecv($i$);
3 Waitall()
```

Algorithm 8: Adapt Implementation

```
1 for $i \leftarrow 0$ to $k$ do
2     Isend($i$)/Irecv($i$);
3     set_Isend_cb($i$)/set_Irecv_cb($i$);
```

The basic building block of ADAPT is algorithm 8, which uses non-blocking point-to-point routines as same as algorithm 7. It provides parallelism to allow concurrent communication and reduces synchronizations by removing the Waitall to alleviate noise propagation.

For MPI_Bcast algorithms that are not based on trees, the event-driven design can still be used as long as the basic building blocks exist in the algorithms. For example, a scatter followed by an allgather is a common algorithm to perform MPI_Bcast for big messages. In the scatter phase, a process may send data to multiple other processes, which is similar to the MPI_Bcast discussed above, and the same technique can be applied to it. For other one-to-all, all-to-one and some all-to-all collectives, a process always need to send or receive data from other processes. Therefore, the basic building block is a common part of various collectives, and the event-driven design can be extended to them.

3.3.4 Support Different Collectives with Multiple Communication Trees

In collective operations, the parents-children relationships form a communication tree. In the ADAPT framework, the communication tree of a collective operation can be any type of tree, e.g., a chain, binary tree, binomial tree, or other advanced trees [54]. The design of the ADAPT framework allows most operations to be independent of the underlying communication tree. Therefore, it is easy to adapt the topology-aware trees, which are created based on hardware topology, to boost performance, and this part is discussed in chapter 4.

3.4 Evaluation

3.4.1 Experimental Setup

This section studies the performance impact of noise on the ADAPT framework and other MPI libraries to demonstrate ADAPT's noise absorption abilities. I use Intel MPI benchmark (IMB) [1] to test the performance of the collective operations. The noise absorption ability of the ADAPT framework is tested on three clusters:

- Saturn, a CPU cluster. In this section, node d00 to d15 are used, on which each node has 2 Intel Xeon E5520 CPUs, and the nodes are connected by Infiniband.
- Cori, a CPU cluster, on which each node is equipped with 2 Intel Xeon E5-2689 v3 CPUs, and the nodes are connected by Cray Aries;
- Stampede2, a CPU cluster, on which each node is equipped with 2 Intel Xeon 8160 CPUs, and the nodes are connected by Intel Omni-Path.

3.4.2 Noise Impact

Noise Injection on Single Process

Figure 3.5 shows the performance of 4 MB MPI_Bcast on 16 processes with one process per node on Saturn. The x-axis represents the rank of participating processes, and the y-axis represents the time of each process spent on the MPI_Bcast. The orange line shows the performance of the ADAPT framework, and the green line presents the performance of the default Open MPI implementation that uses the nonblocking point-to-points routines. This figure shows that even though using the same algorithm (binary tree), the ADAPT framework can achieve better performance than the implementation using nonblocking point-to-point, which is explained in the next chapter 4.3.

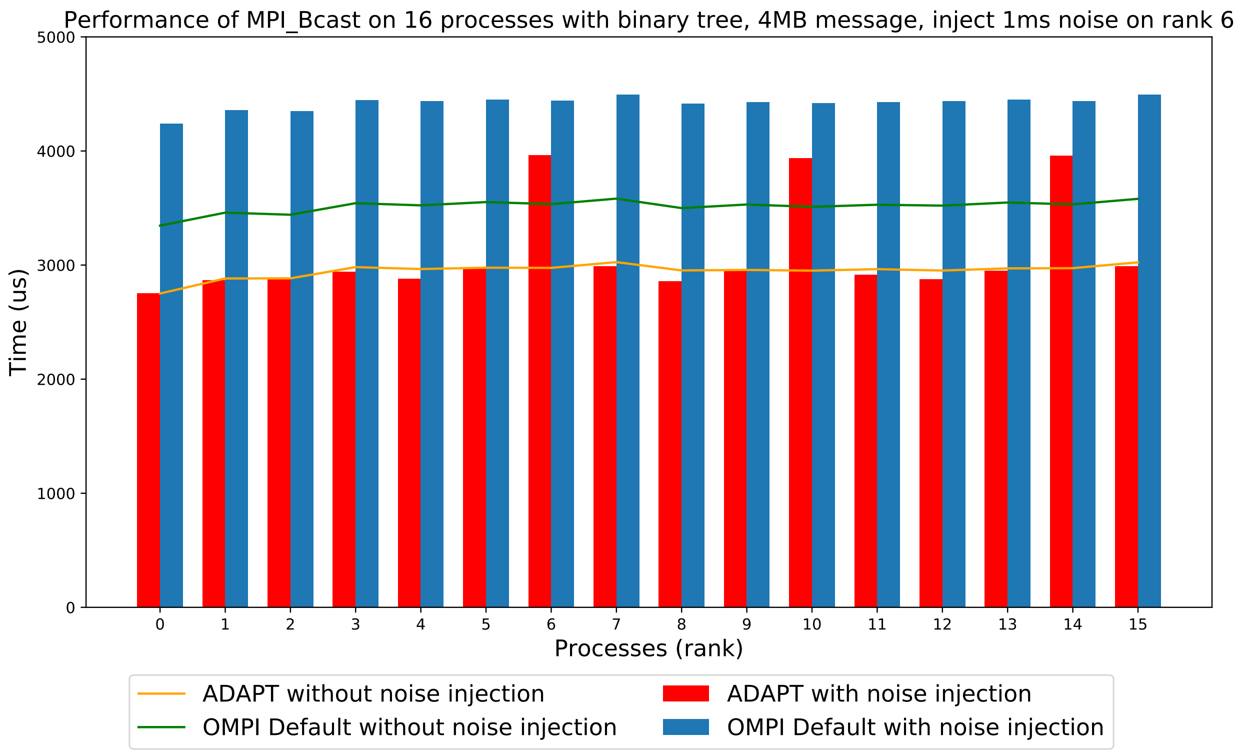


Figure 3.5: Performance of MPI_Bcast with noise injection (16 processes, binary tree, 1 process per node, 4 MB message, Saturn)

To clearly show the effect of noise propagation, in the red bars and blue bars in figure 3.5, I inject 1 ms noise on processes 6 using a method similar to [9]. Comparing the orange line with the red bars and the green line with the blue bars, it is easy to observe that the delay on process 6 is propagated to all the processes in the default Open MPI implementation, while in the ADAPT framework, the noise only can reach the decedents (process 10 and 14) of the delay processes. This result verifies the analysis in section 3.3.

Noise Injection on Multiple Processes

Figures 3.6 and figure 3.7 present the noise impact on the performance of MPI_Bcast and MPI_Reduce operations in different MPI implementations on the Cori and the Stampede2 using 1024 and 1536 processes, respectively. In this experiment, message size is set to 4 MB to allow enough segments to fill the pipeline and highlight the noise effects on performance. As suggested in [35, 22], noise usually has greater impact on larger systems. To show the delays of collective operations caused by noise with fewer processes, I use a method similar to the one in [9] to inject noise. I randomly inject 0–10ms (average 5%) and 0–20ms (average 10%) noise following a uniform distribution with a fixed frequency of 10 Hz, since low-frequency, long-duration noise has great impact on performance [22]. In figure 3.6 and figure 3.7, the blue bars show the average time to do a collective operation with no noise injected; the red bars and the green bars present the time of that collective operation after 5% and 10% noise injection, respectively; and numbers above red and green bars show the performance slow down percentages.

In figure 3.7 after 10% noise injection, the slowdown of MVAPICH is 868%, which is beyond the range of the Y-axis in the figure. MVAPICH's MPI_Reduce encounters segmentation fault with IMB on Stampede2 when message size is 4MB, so there are no results for the MPI_Reduce operation. As seen in figure 3.6 and figure 3.7, benefiting from the event-driven design, the ADAPT framework relaxes synchronization dependencies, and thus it is largely unaffected by the noise as compared to other MPI libraries. As a result, the ADAPT framework only slows down up to 24% and 16% on MPI_Bcast and MPI_Reduce with 10% noise injected on both machines. The MPI_Bcast and MPI_Reduce of the default Open MPI (OMPI-default) use non-blocking point-to-point routines, so as analyzed in section 3.2.2, it

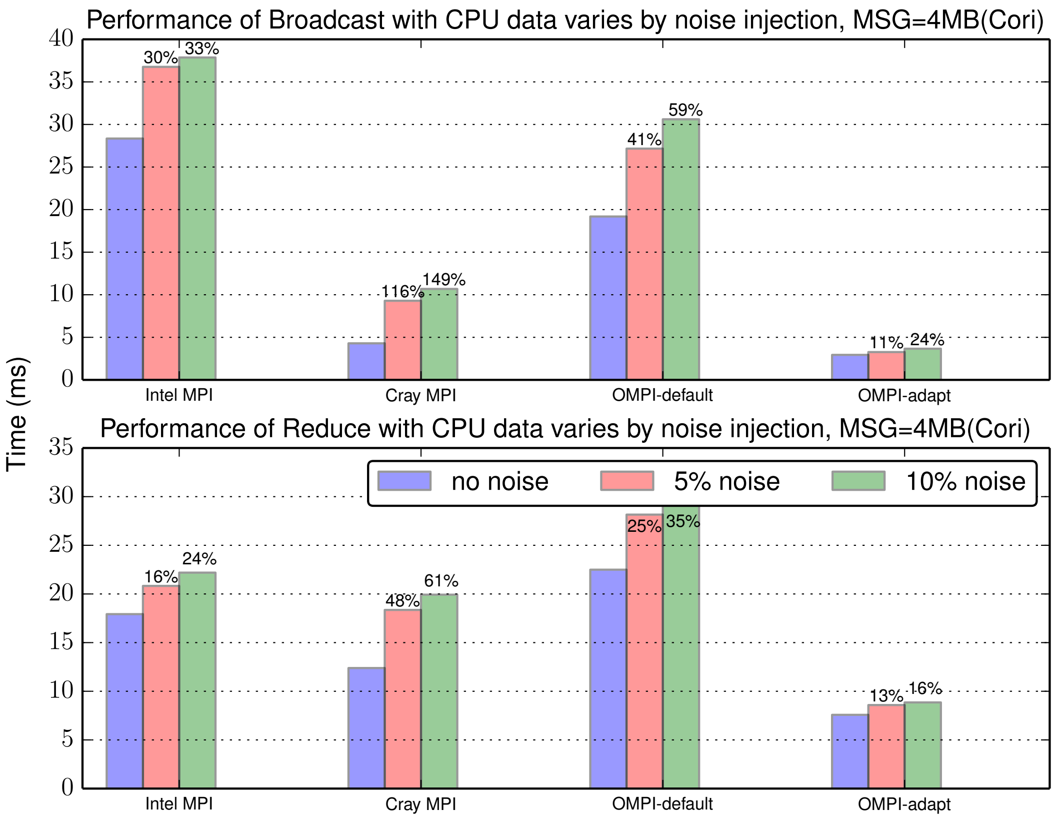


Figure 3.6: Performance of MPI_Bcast and MPI_Reduce with noise injection (1K cores on Cori, numbers on bars represent performance slowdown caused by noise)

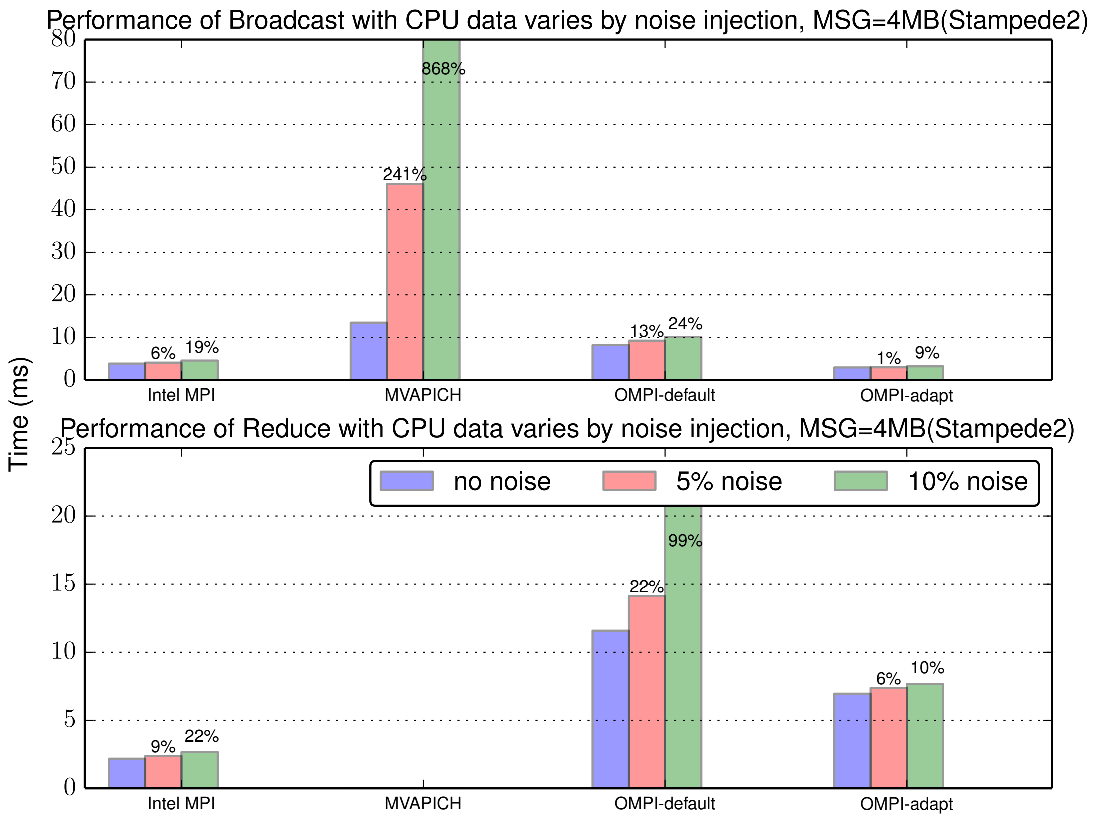


Figure 3.7: Performance of MPI_Bcast and MPI_Reduce with noise injection (1.5K cores on Stampede2, numbers on bars represent performance slowdown caused by noise)

propagates noise because of the synchronization dependencies. Thus, it slows down up to 59% and 99% on MPI_Bcast and MPI_Reduce, respectively. Since Cray and Intel MPI are not open-source, I do not know their detailed implementations, but Cray MPI slows down up to 149% and 61%, and Intel MPI slows down 33% and 24%, both are less noise-resistant than the ADAPT framework.

Chapter 4

Hierarchical Collective Operations - Topology-Aware Tree

4.1 Overview

HPC systems are becoming more heterogeneous, resulting in an increasingly complex hardware hierarchy. To fully utilize the different characteristics of the hardware on each level of the hierarchy, and to improve the performance of collective operations on these systems, hierarchical collective operations are proposed. The hierarchical collective operations can utilize the hierarchical structure of HPC systems to optimize collective communications.

This chapter describes my first approach to implementing hierarchical collective operations. In this approach, I extend the ADAPT framework with a topology-aware tree to handle the complex hardware hierarchy in heterogeneous systems. In section 4.2, I introduce the idea and implementation of the topology-aware tree. Then in section 4.3, I explain why the ADAPT framework can have the best performance compared to other implementations with the same topology-aware tree. Finally, in section 4.4, I evaluate the performance of this implementation of hierarchical collective operations with other MPI implementations.

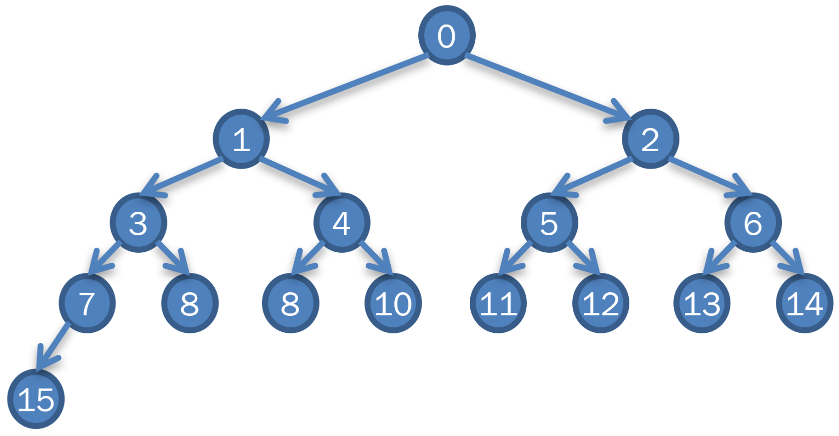


Figure 4.1: MPI_Bcast with a binary tree (16 processes)

4.2 Topology-Aware Communication Tree

4.2.1 The Problems of Existing Tree-Based Algorithms

The default collective framework in Open MPI supports multiple tree-based algorithms for each collective operation, and all these algorithms are designed and optimized based on the assumption of the equal cost of MPI point-to-point communications between any two processes. However, this assumption is not valid anymore because the cost of MPI point-to-point routines on modern HPC systems varies hugely based on the placement of the processes.

Figure 4.1 shows an example of MPI_Bcast with a binary tree algorithm on 16 processes, and figure 4.2 shows the placement of processes after mapping them to a actual machine, which has two nodes, two sockets per node, and four cores per socket. After the processes mapping, the 15 MPI point-to-point communication routines in figure 4.1 become 8 inter-node point-to-point routines, 4 inter-socket point-to-point routines, and 3 intra-socket point-to-point routines, which are shown in figure 4.2 as red arrows, green arrows, and black arrows, respectively. Since the inter-node communication cost much more time than the inter-socket communication and the inter-socket communication cost more time than the intra-socket communication, to improve the performance, I rebuild the tree based on the hardware topology to minimize the slow point-to-point routines (inter-node and inter-socket)

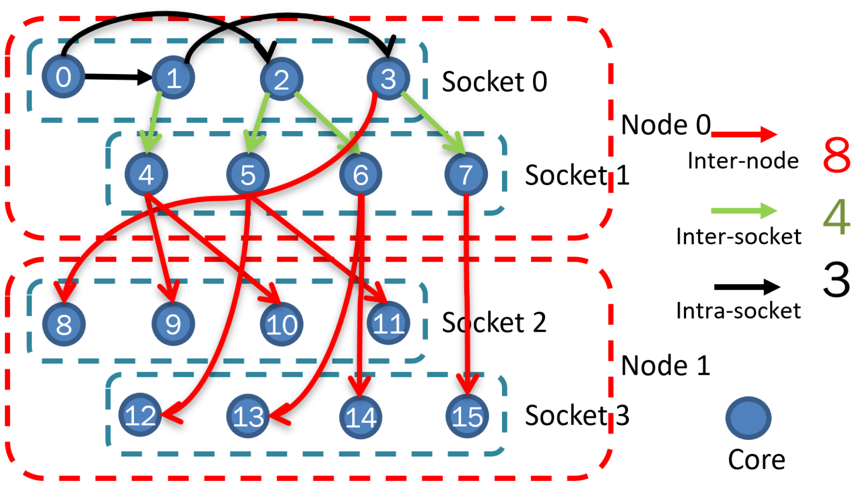


Figure 4.2: Process placement with a binary tree
(16 processes, 2 nodes, 2 sockets per node, 4 cores per socket)

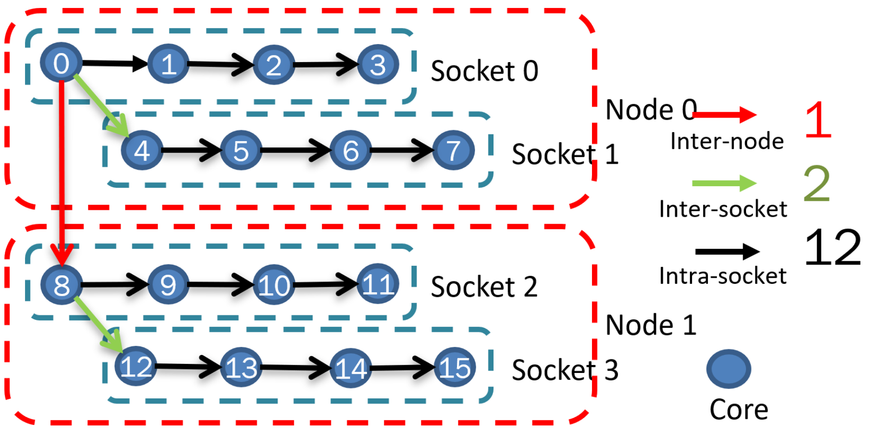


Figure 4.3: Process placement with a topology-aware tree
(16 processes, 2 nodes, 2 sockets per node, 4 cores per socket)

and maximize the fast point-to-point routines (intra-socket). After the rebuilding, the new communication tree, which is called a topology-aware communication tree, only has 1 inter-node point-to-point routine, 2 inter-socket point-to-point routines, and 12 intra-socket point-to-point routines, as shown in figure 4.3.

4.2.2 Build a Topology-Aware Tree

This section describes how to build such a topology-aware tree. The first step to build a topology-aware tree is to get the placement of each process. To gather the hardware topology information, each process in a given communicator firstly collects its placement using the Portable Hardware Locality (hwloc) [11] framework. The hwloc software package provides an abstraction of the hierarchical topology of modern architectures, including nodes, sockets and cores. Because of the limitation of hwloc, I cannot get further network topology information such as network switch or router. Hence, I only consider three hardware hierarchy levels in this work: node level, socket level and core level and record the location of each process with a topology tuple (Node_ID, Socket_ID, Core_ID). After every process gathers its topology tuple, all processes exchange their local information so that all of them have topology information about other processes to form a topology table. Later, the topology table is cached in memory, so that all following collective operations within the same or duplicated communicator can directly use the cached topology table instead of gathering it again, no matter what kind of collective operations they are.

Based on the topology table, a topology-aware tree can be built to perform collective operations. At the core level, all processes belong to a socket form a group, which is called a core group. As in figure 4.4, P0, P1, P2, and P3 are in the same core group. From each core group, a socket leader is selected. In figure 4.4, I choose the process with the lowest Core_ID on the same socket as the socket leader. However other leader selection strategies could also be used to further improve the performance, but it is out of the scope of this dissertation. At the socket level, all socket leaders in the same node form a socket group. As shown in figure 4.4, P0 and P4 belong to a socket group. Furthermore, a node leader is selected among all the socket leaders in the same node, and all the node leaders form a

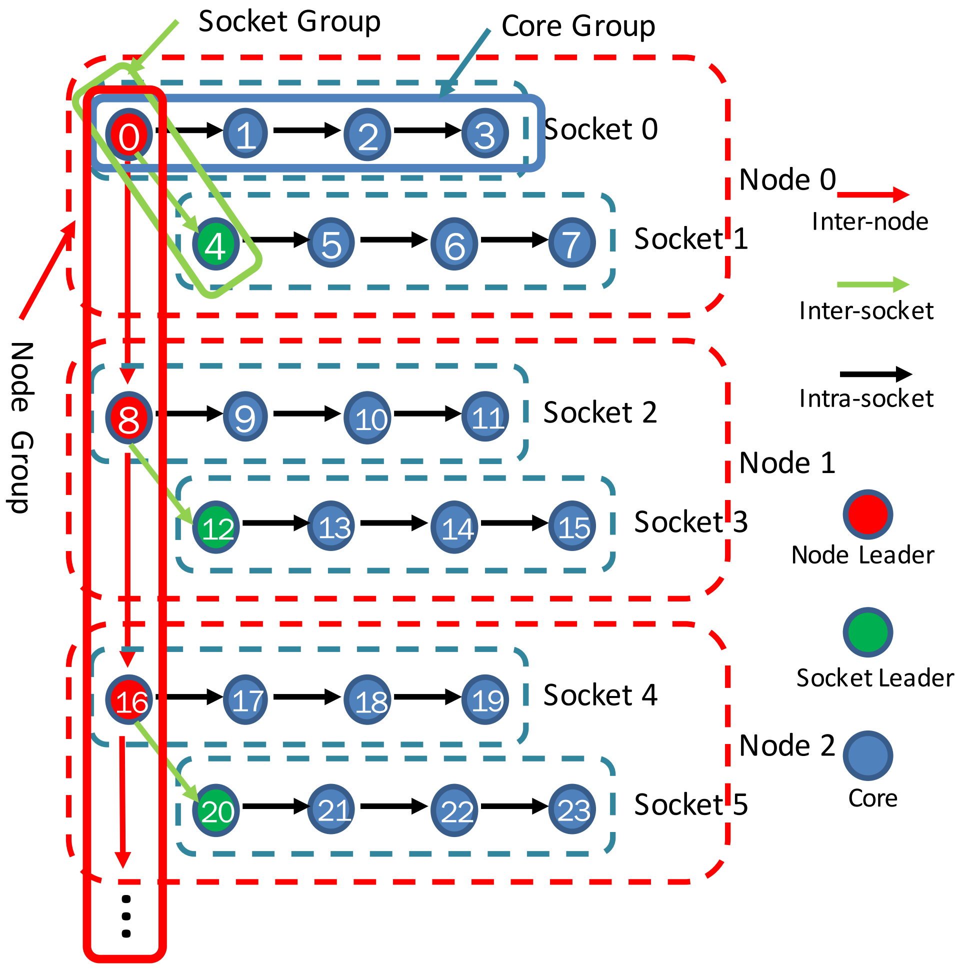


Figure 4.4: Topology-aware communication tree of MPI_Bcast on a multi-core cluster (2 sockets per node, 4 cores per socket)

node group, which includes P0, P8, and P16. In a word, each group has a leader, which is a member of upper-level group.

To set up the root of a topology-aware tree, the topology-aware tree is re-shaped. In figure 4.5, I assume process 14 is the root of a broadcast operation. In the root's core group, if the root is not a socket leader, I swap the socket leader with the root as swapping P14 and P12. After that, if the root is not a node leader, the root's whole socket is swapped with node leader's socket logically like swapping socket 3 and socket 2 in the tree. At last, if the root of the broadcast operation is still not the root of the topology-aware tree, then at the node level, I swap the root's node with the first node in the tree. For example, in figure 4.5, node 0 and node 1 is swapped in the last step.

4.3 ADAPT vs. Other Implementations

The collective operation implementations using blocking/non-blocking point-to-point routines, which are discussed in section 3.2, can support different kinds of tree-based algorithms. Hence, it is possible to plug the topology-aware tree discussed above into these implementations to make them topology-aware. Also, as discussed in section 3.3.4, the ADAPT framework can also be plugged in with a topology-aware tree as well to improve its performance.

After equipping with a topology-ware tree, the implementation using nonblocking point-to-point routines performs better than the version using blocking point-to-point routines. This is because the previous one can relax the dependencies and exploit more parallelism by posting several MPI_Isends to transfer data concurrently if the MPI_Isends occupy different hardware. For example, in a MPI_Bcast using a topology-aware tree as in figure 4.4, P0 posts three MPI_Isends to P1, P4, and P8, which occupy inter-node, inter-socket, and intra-socket communication channels, respectively, and hence these MPI_Isends can be progressed independently at different speeds. However, the Waitall in the implementation using nonblocking point-to-point routines forces these three MPI_Isends to complete at the same time, resulting they all run at the slowest speed of the three communications, which

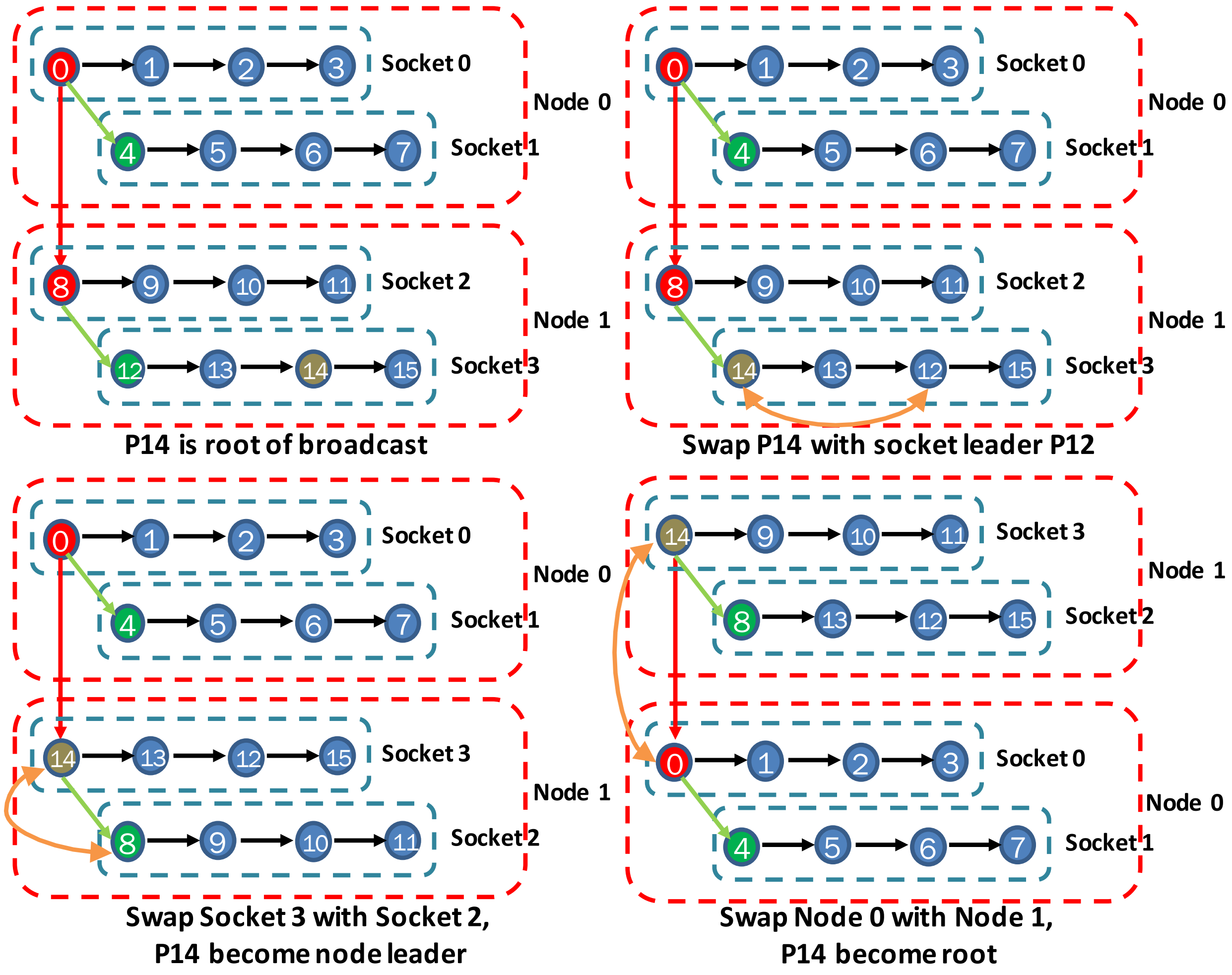

Socket 0
Socket 1
Node 0
Socket 2
Socket 3
Node 1
P14 is root of broadcast
Swap P14 with socket leader P12
Swap Socket 3 with Socket 2, P14 become node leader
Swap Node 0 with Node 1, P14 become root

Figure 4.5: Rank reorder of MPI_Bcast on a multi-core cluster
(2 sockets per node, 4 cores per socket)

is the inter-node communication. Therefore, this implementation can not fully utilize the hardware.

The ADAPT framework removes the Waitall, so in the previous example, the three Isends can be progressed independently, and data can be propagated at full bandwidth. Therefore, the ADAPT framework is able to utilize the hardware more efficiently and provide a better performance, which is shown in the next section.

4.4 Evaluation

4.4.1 Experimental Setup

This section evaluates the performance of topology-aware broadcast and reduce operations in the ADAPT framework against state-of-the-art MPI implementations on two clusters:

- **Cori**, a CPU cluster, on which each node is equipped with 2 Intel Xeon E5-2689 v3 CPUs, and nodes are connected by Cray Aries;

- **Stampede2**, a CPU cluster, on which each node is equipped with 2 Intel Xeon 8160 CPUs, and nodes are connected by Intel Omni Path.

4.4.2 Performance of Different Topology-aware Algorithms

Figures 4.6 and figure 4.7 present the performance of the ADAPT framework (shown as OMPI-adapt) with CPU data on Cori and Stampede2, compared with all the topology-aware algorithms in Intel MPI. I also integrate the topology-aware communication tree to the default collective module of Open MPI (shown as OMPI-default-topo) to demonstrate that ADAPT is better at hardware resource utilization.

The figures show that the Intel MPI performs much better on Stampede2 than Cori, and this may be because the underlying interconnects of Stampede2 is Intel Omni Path, whereas Intel MPI has its own optimizations for its hardware. Even so, on both machines, the topology-aware MPI_Bcast of the ADAPT framework performs the best over others for big messages. When messages are smaller than 1 MB, the ADAPT framework's MPI_Bcast

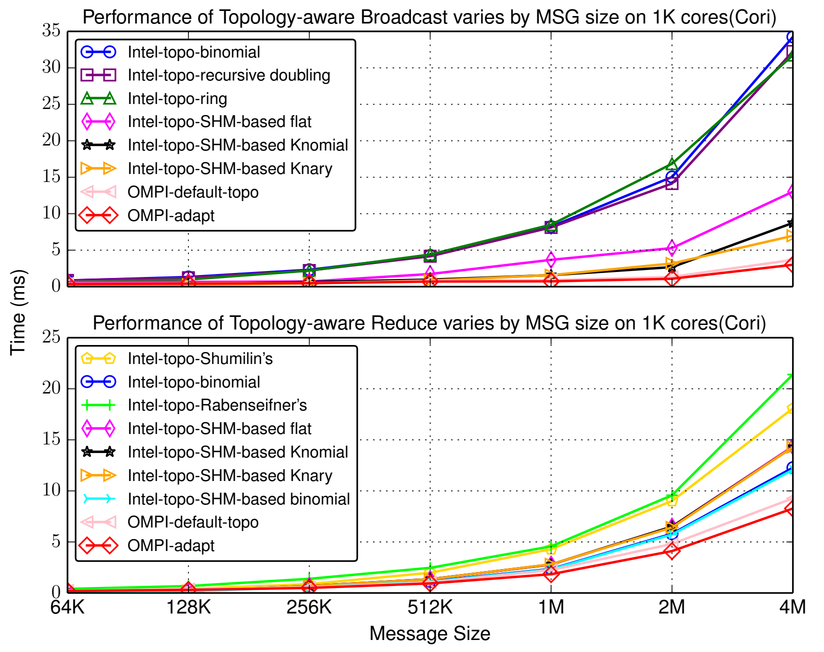


Figure 4.6: Performance of topology-aware MPI_Bcast and MPI_Reduce (1K cores, Cori)

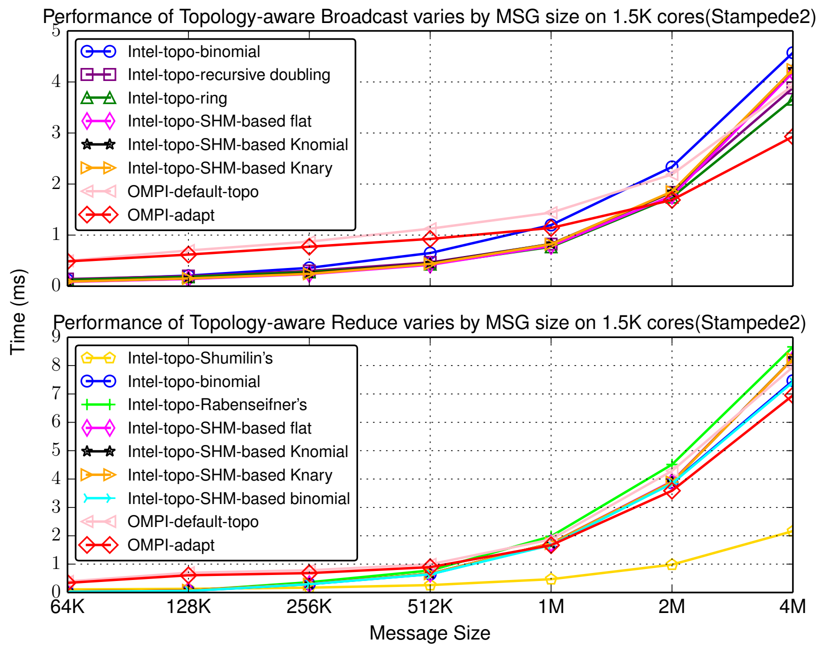


Figure 4.7: Performance of topology-aware MPI_Bcast and MPI_Reduce (1.5K cores, Stampede2)

is a little slower than Intel MPI on Stampede2; the reason for that is the topology-aware algorithms in the ADAPT framework require enough segments to fulfill the pipeline. On the MPI_Reduce with large messages, ADAPT performs better than most topology-aware algorithms in Intel MPI, except Shumulin's. A possible reason for the better performance of Shumulin's algorithm is it may be optimized for Intel Omni-Path since the performance of Shumulin's on Cori (without Omni-Path) is much slower than on Stampede2 (with Omni Path).

Comparing OMPI-adapt and OMPI-default-topo, even though they have the same topology-aware communication tree, ADAPT still performs 20% better than the default framework in Open MPI since the ADAPT framework is able to support independent communications over different hardware. Overall, the ADAPT framework eliminates boundaries between different levels in the hardware hierarchy and supports independent communications, and hence it can fully utilize hardware resources and deliver better performance than most topology-aware MPI implementations, especially for big messages.

4.4.3 Performance of MPI Implementations

To study the overall impact of the combination of the event-driven design and the topology-aware tree in the ADAPT framework, two types of experiments are performed: first, the total number of processes is fixed and I test the performance for different message sizes; second, I look at the strong scalability, which measures the performance by varying the number of processes with a fixed message size.

Figure 4.8 and figure 4.9 present the time of MPI_Bcast and MPI_Reduce of the ADAPT framework compared with other state-of-the-art MPI implementations with different message sizes on the Cori and Stampede2. Cray MPI does not support Omni-Path interconnects, so it is not tested on Stampede2. Similarly, MVAPICH does not support Cray Aries interconnects, so it is not tested on Cori. The default collective module in Open MPI is the Tuned module, which can switch algorithms based on different message sizes. This is shown as the green lines in figure 4.8, where the algorithm used by the default framework in Open MPI is changed after 256KB. In the ADAPT framework, both MPI_Bcast and MPI_Reduce are pipelined algorithms, in which messages are split into several segments. To better understand this

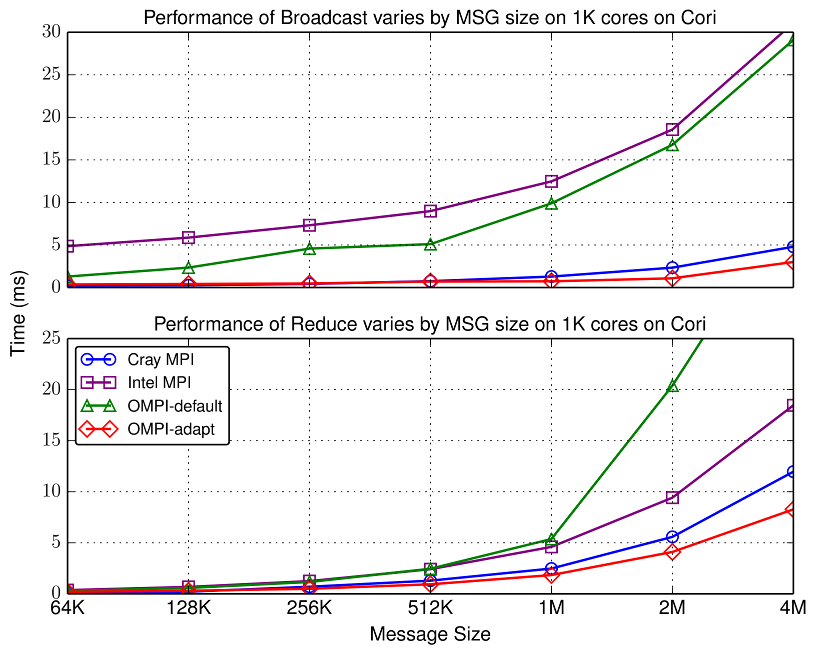


Figure 4.8: Performance of MPI_Bcast and MPI_Reduce varies by message size (1K cores, Cori)

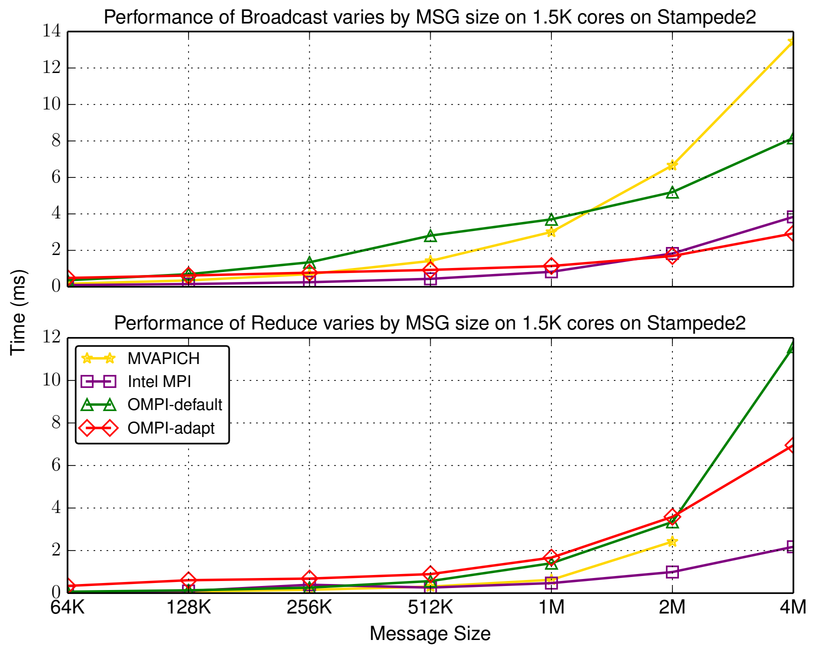


Figure 4.9: Performance of MPI_Bcast and MPI_Reduce varies by message size (1.5K cores, Stampede2)

performance graph, I adopt the Hockney's cost model [33] to model the cost of collective operations. This model assumes that the time to send a message of size m between two nodes is $T = \alpha + \beta m$, where α is the latency (or startup time) per message, independent of message size, and β is the transfer time per byte or reciprocal of network bandwidth. As for MPI_Reduce, I assume that the time spent in computation on data in a message of size m is γm, where γ is computation time per byte. Therefore, the entire time of sending a message of size m between two processes is $T = \alpha + \beta m + \gamma m$.

A perfect pipeline needs to meet two criteria: a large enough segment size and a sufficient number of segments. If the segment size is too small, message latency as α in the Hockney's model becomes dominant, preventing the full utilization of network bandwidth. If there are not enough segments, the pipeline initialization time becomes predominant, and hence the overall performance is affected. It is difficult for small messages to meet both criteria, and this means the ADAPT framework could show lesser improvement over other implementations when the messages are small. On large messages, the benefit of concurrent communications in the ADAPT framework becomes the dominant factor. On Cori, ADAPT provides 10×, 10×, and 1.6× speedup against OMPI-default, Intel MPI, and Cray MPI for MPI_Bcast and 5×, 2× and 1.5× speedup for MPI_Reduce when the message size is 4MB. On the same message sizes on Stampede2, compared with OMPI-default, Intel MPI and MVAPICH, ADAPT achieves 2.8×, 1.3× and 4.6× speedup for MPI_Bcast. ADAPT's MPI_Reduce is slower than Intel MPI's on Stampede 2, and the reason is explained in the previous section.

Scalability is another important factor of MPI libraries. Figure 4.10 shows the performance of strong scaling for MPI_Bcast and MPI_Reduce operations with 4MB messages on Cori. In this experiment, the ADAPT framework uses the chain algorithm as the communication tree for all groups in the topology-aware tree based on [50]. Since the ADAPT framework allows concurrent communications over independent hardware, the cost of MPI_Bcast and MPI_Reduce can be calculated through the longest chain in the communication tree.

Based on the Hockney's Model, the cost of chain is $T = (P + n_s - 2) \times (\alpha + \beta m)$, where P is the number of processes participating in a collective operation, and n_s is the number of segments. If the message size is large enough to ignore the cost of pipeline initialization, the

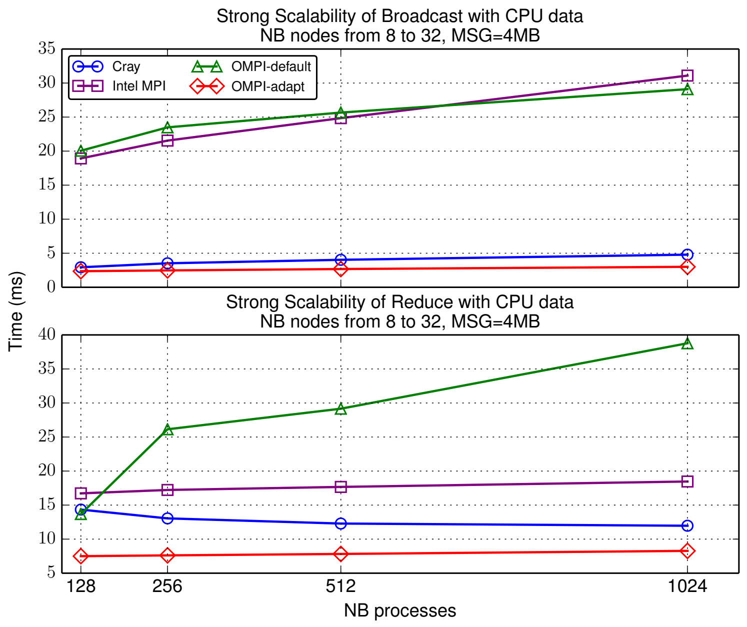


Figure 4.10: Strong scalability of MPI_Bcast and MPI_Reduce varies by number of nodes (4 MB message, Cori)

cost of the chain algorithm can be treated as $T = n_s \times (\alpha + \beta m)$. Thus, theoretically, the performance of the chain algorithm does not depend on the number of processes within the chain. Therefore, as seen in figure 4.10, the time of ADAPT's MPI_Bcast and MPI_Reduce does not increase significantly with the number of processes, and the operations tend to be stable as the number of nodes increases. Compared with other MPI implementations, the ADAPT framework consistently achieves the best strong scalability, thanks to its event-driven design and the topology-aware communication tree.

Chapter 5

Hierarchical Collective Operations - Combine Specialized Modules

5.1 Overview

As introduced in the previous chapter, combining the topology-aware tree and the event-driven collective framework can maximize the communication overlap on different levels. However, this approach fails to fully utilize the hardware capability on each level, since both the topology-aware tree and the event-driven framework are based on MPI point-to-point communication routines even for intra-node communication. Typically, in a point-to-point communication, there are two memory copies, one from a send buffer to a temporary buffer, and another one from the temporary buffer to a receive buffer, however this is not optimal for intra-node communication.

The processes in a node share the same memory space, so if a collective framework can utilize that, then it only needs one memory copy to directly move the data from the source to the destination. Hence, to improve the performance of intra-node MPI collective operations, it is crucial to have specialized collective frameworks that can utilize the shared memory space among participating processes. Besides the specialized modules using the shared memory space in a computing node, the collective operations on other hardware need specialized modules as well. For example, GPU collective operations are very different from the conventional CPU collective operations since GPUs are typically connected via PCI-E or NVLink. Hence, specialized GPU collective modules are required and introduced [61, 47] to achieve decent performance on GPUs. The same goes for the inter-node level, where the communications go through the interconnects. As demonstrated in [54], the collective operations on this level need to leverage the full-duplex mode of the network to fully utilize the bandwidth. Moreover, if the network switch level information is available, collective operations can be further optimized [36, 58] using that information. Once specialized modules on each level of the hardware are implemented, I can combine the collective operations from these specialized modules to perform hierarchical collective operations.

This chapter introduces my second approach to perform hierarchical collective operations. It is divided into two sections. In section 5.2, I introduce some specialized modules in Open MPI; and in section 5.3, I present present "HAN," a new hierarchical autotuned collective framework in Open MPI, which selects the suitable homogeneous collective communication modules as submodules for each hardware level, uses the collective operations from the submodules as tasks, and organizes these tasks to perform efficient hierarchical collective operations. With a task-based design, the tasks with different implementations can be easily swapped to adapt to new networks, which makes the HAN framework not only suitable for current machines but also adapted to the fast-changing HPC systems.

5.2 Specialized Modules in Open MPI

Picking suitable modules for each level is the first part of this implementation of hierarchical collective operations. To utilize the shared memory space among the processes in the same node, my second approach breaks the hierarchical collective operations into two levels (inter-node and intra-node); and hence, I need to find proper modules on these two levels.

5.2.1 Inter-Node Modules

As discussed in the introduction, overlapping the collective communication on different levels is an important factor to the performance of hierarchical collective operations. To attain the overlap of inter- and intra-node communications, my design relies on non-blocking collective operations for inter-node communications, as discussed in section 2.4. In the current Open MPI, there are two existing collective communication modules that support non-blocking collective operations: (1) Libnbc [34], a default legacy module, and (2) ADAPT, the previous introduced framework with an event-driven design. These two modules can be selected to perform inter-node collective operations in this design.

5.2.2 Intra-Node Modules

As for intra-node collective operations, Open MPI provides two modules, SM and SOLO. SM is a module that utilizes shared memory buffers to communicate with other processes, and SOLO is an experimental module, which utilizes MPI one-sided communication.

Figure 5.1 shows the performance of intra-node MPI_Bcast on 20 processes in the same computing node. This experiment is on Saturn, which is a small scale machine at Innovative Computing Laboratory (ICL). The computing node used in this test has two Haswell E5-2650 processors, and each processor has ten cores. The tested message sizes cover a long range, from 512 bytes to 4 MB. The red bars show the performance of the SOLO module, the blue bars show the performance of the SM module, and the orange bars represent the performance of the default module in Open MPI. Both the SM module and the SOLO module have better performance than the default module in Open MPI on all message sizes, which shows the benefits of utilizing the shared memory space. Comparing red bars and blue bars, the SOLO module has worse performance than the SM module on small messages. It needs 500% time of the SM module on 512 bytes messages. However, with the increasing message size, the performance gap of the SOLO module and the SM module decreases. When the message size is 262144 bytes (256 KB), the SOLO module starts to be faster than the SM module; and eventually, the SOLO module only needs 71% time of the SM module to do an intra-node MPI_Bcast on 4 MB message.

Figure 5.2 and figure 5.3 show the performance of intra-node MPI_Reduce and MPI_Allreduce on 20 processes in the same computing node, respectively. The computing node used in this test is the same as the previous experiment of the intra-node MPI_Bcast. The SOLO module has the worst performance on small messages for both MPI_Reduce and MPI_Allreudce. It is even slower than the default module in Open MPI that using point-to-point routines.

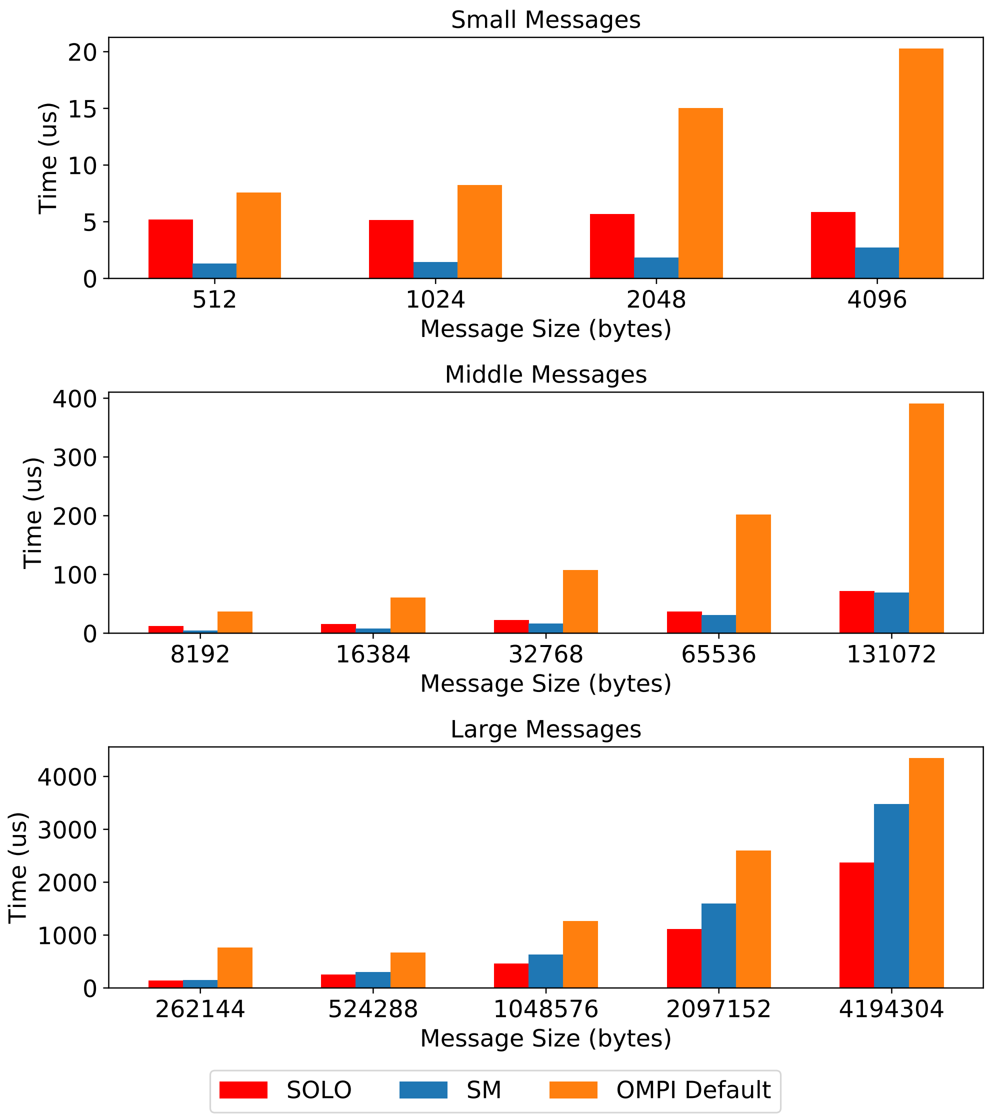


Figure 5.1: Performance of Intra-node MPI_Bcast (20 processes, Saturn)

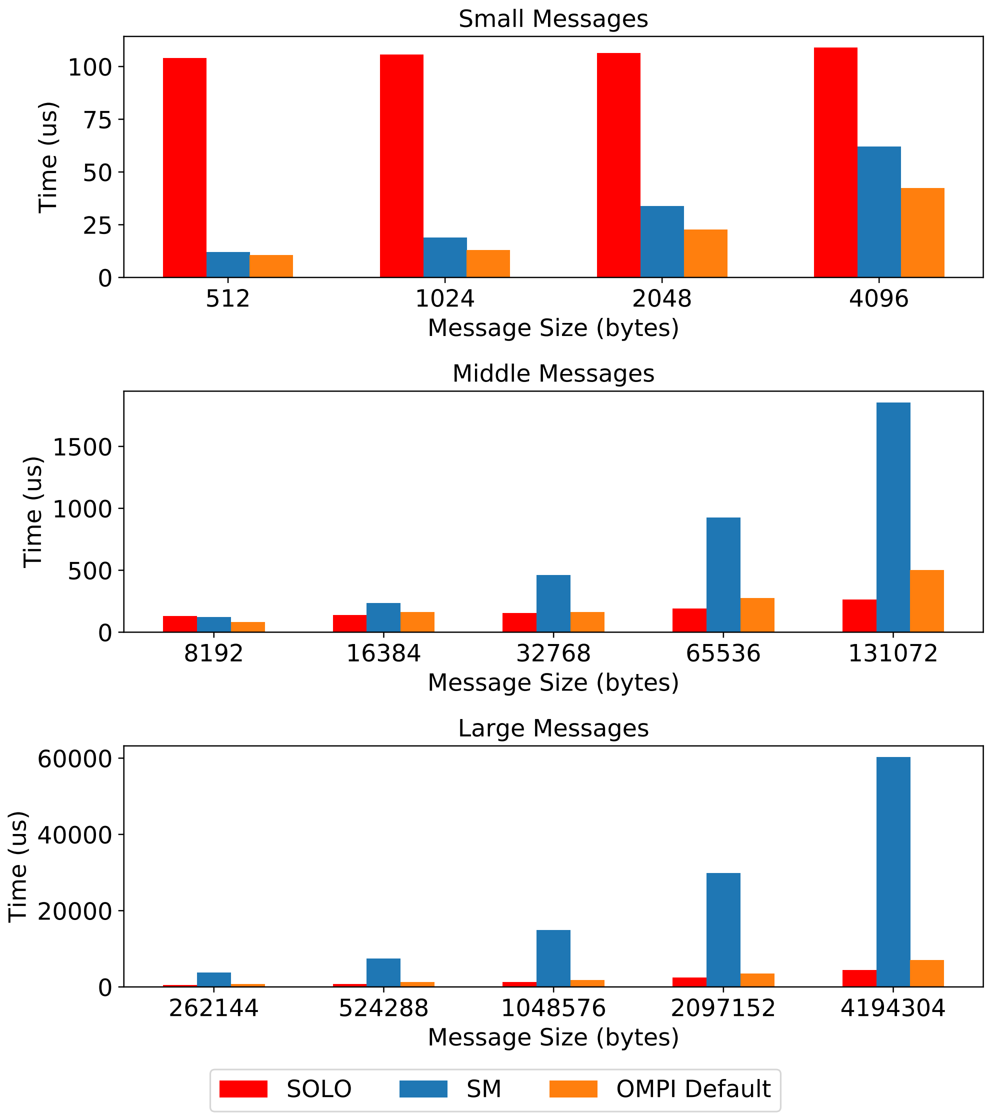


Figure 5.2: Performance of Intra-node MPI_Reduce (20 processes, Saturn)

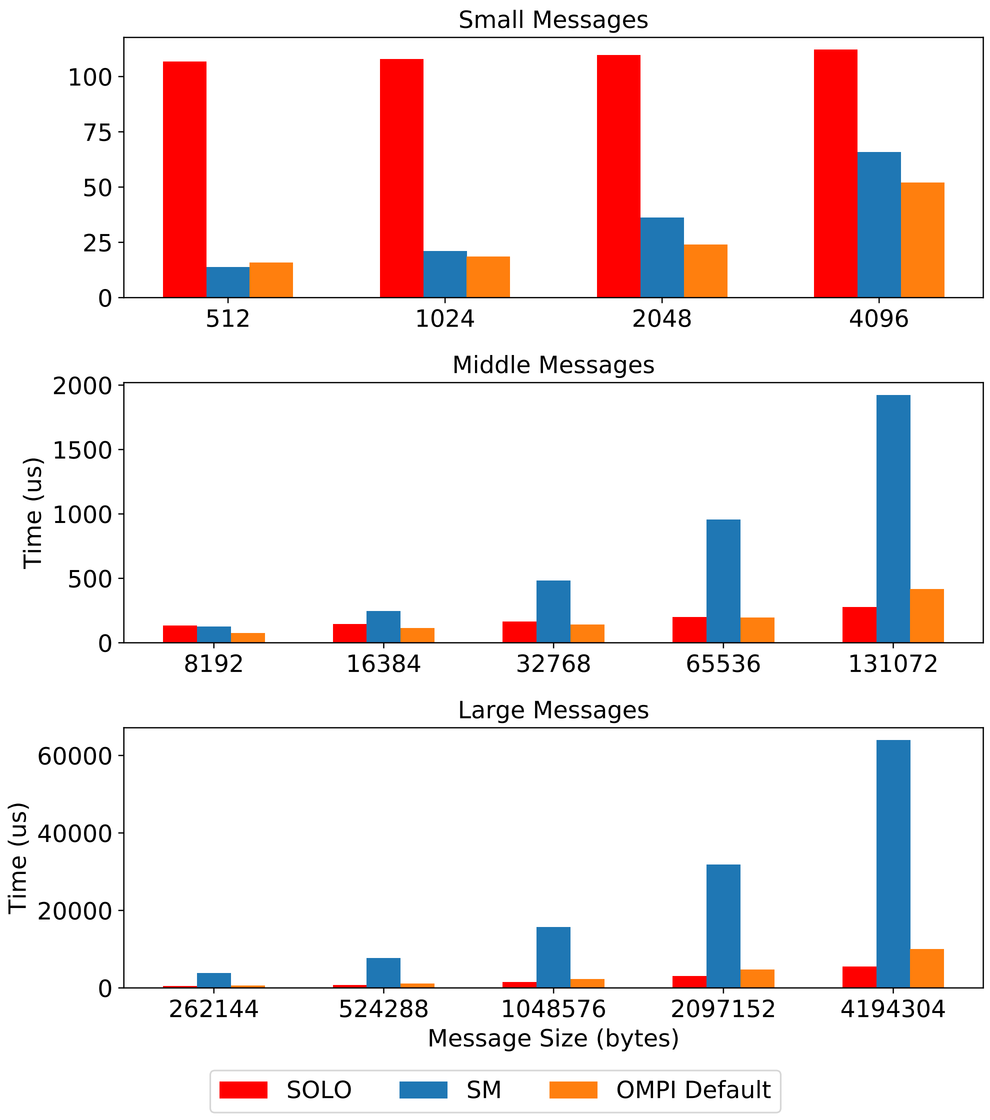


Figure 5.3: Performance of Intra-node MPI_Allreduce (20 processes, Saturn)

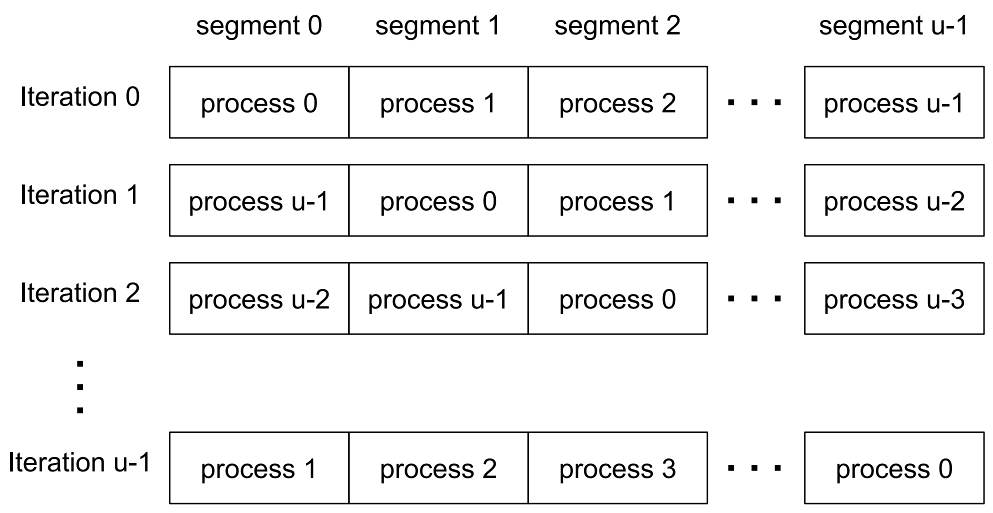


Figure 5.4: The reduction phase of MPI_Reduce and MPI_Allreduce in the SOLO module

With the increase of the message size, the performance of the SOLO module increases as well, resulting in the SOLO module becomes the fastest module among all the three tested modules when the message size is between 16384 bytes (16 KB) and 4191304 bytes (4MB). The reason for such performance is the SOLO module always segmentizes the message, and this is not optimal for small message.

The implementations of MPI_Reduce and MPI_Allreduce in the SOLO module have two phases: reduction phase and distribution phase. In the reduction phase, the processes copy the data to a shared memory buffer, which can be accessed for all the processes in the same node and perform reduction operation there concurrently. When the reduction phase is finished, the final result of the MPI_Reduce and MPI_Allreduce is located at the shared memory buffer. Then in the distribution phase, this final result is copied back to the root process in MPI_Reduce or to all the processes in MPI_Allreduce. Among these two phases, the reduction phase is the most time-consuming part.

Figure 5.4 shows the algorithm of the reduction phase with u processes. The reduction phase needs u iterations, and the message is divided into u segments. At each iteration, each segment is accessed by one process on the shared memory buffer. At iteration 0, process 0

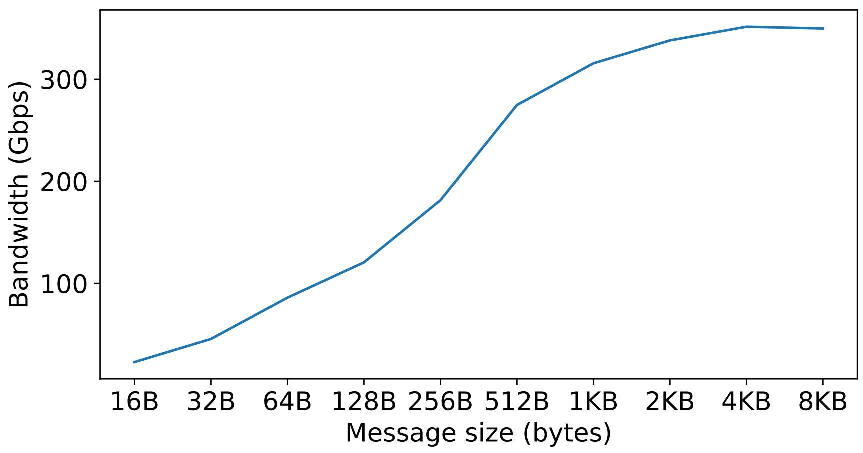


Figure 5.5: Memcpy bandwidth on Saturn

accesses the segment 0 on the shared memory buffer, and it copies its segment 0 from the local buffer to the shared memory buffer. At that time, the segment 0 on the shared memory buffer contains the data of process 0, which is copied at the previous iteration; hence, process $u-1$ performs a reduction operation on segment 1 between the shared memory buffer and its local buffer. In the following iterations, each process accesses segment 0, in turn, to reduce its data to the shared memory buffer, and after $u-1$ iterations, the segment 0 on the shared memory buffer contains the first part of the final result. The same algorithm applies for other segments, as shown in figure 5.4, and at the end of the reduction phase, each segment on shared memory buffer contains a part of the final result.

By using the pipelining technique, this method allows all the processes to access the shared memory space concurrently to fully utilize the memory bandwidth. At each iteration, all processes participate in the collective operation, and no process is idle. For example, at iteration 0, all the processes copy its segment i to the shared memory buffer, as shown in the first row in figure 5.4. However, using the pipelining technique has some limitations. One of them is that the segment size cannot be too small.

Figure 5.5 shows the memcpy bandwidth on a computing node of Saturn using netpipe [56], and it suggests the memcpy bandwidth grows with the increasing message size. It also shows that the messages need to be big enough to reach high memcpy bandwidth, and for small messages, the memcpy bandwidth is very low. As shown in figure 5.5, when the message size is 16 bytes, the memcpy bandwidth is 23 Gbps, which is only about 6% of the memcpy bandwidth of 8 KB messages. So if the segment size in the SOLO module is too small, the cost of transferring these segments becomes too high, which hurts the performance of the collective operations, and this could explain the performance issue of the SOLO module on small messages.

Both the SOLO module and the SM module take advantage of the shared memory space. However, from the above analysis, due to the differences in algorithms and implementations, the SM module exhibits better performance with smaller messages while the SOLO module performs significantly better for larger ones.

5.3 Combine Specialized Modules - HAN: Task-Based Design

After selecting proper modules, the next step is to combine them, which requires a new hierarchical collective operation framework. Two crucial factors are considered when implementing the new hierarchical collective operations framework.

First, communications on one level need to overlap with the ones on other levels, especially for large messages. From the hardware perspective, the data transfer on different levels can be mostly independent of each other naturally since they mainly occupy different hardware devices. However, in software implementations, some problems such as lacking enough segmentation or sharing software resources would limit the communication overlap; thus, those problems need to be avoided.

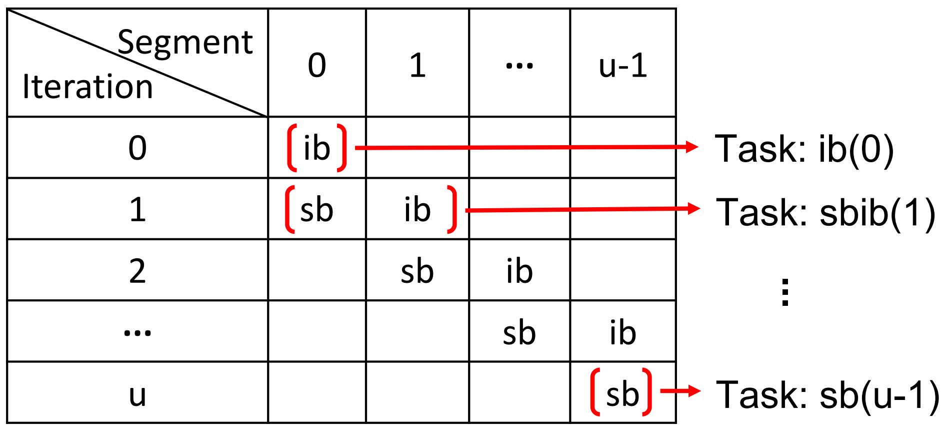
Second, to meet the requirement of the fast-changing hardware, a hierarchical collective framework needs to be flexible enough to adapt to new architecture easily. In the inter-node level, various interconnects have been introduced with different network topologies such as hypercube [4], polymorphic-torus [43], fat-tree [41], dragonfly [40]. Inside a node, with adopting co-processors such as GPUs, how these computing units are connected varies drastically. For example, the Summit machine at ORNL utilizes the NVLink as the interconnects between CPU/GPUs and GPUs, which is totally different from conventional PCI-Express in terms of topology, latency, and bandwidth.

To address these problems, I present “HAN” (Hierarchical AutotuNed), a flexible hierarchical collective framework in Open MPI, which uses the existing homogeneous collective modules of Open MPI as its submodules, and combines them to perform hierarchical collective operations. It provides three features to address the factors discussed before. First, it picks proper collective frameworks as submodules to utilize the hardware specification of each level. Second, it adopts the pipelining technique to provide the capability of overlapping the communications of different levels. Last, it can easily switch out submodules to adapt to hardware updates with its task-based design.

The HAN framework groups processes based on their physical locations, and hence divides collective operations into two levels: inter- and intra-node. In the inter-node level, each node selects one process (**node leader**) to exchange messages across nodes, while in the intra-node level, all the processes, including node leaders and the other processes, participate in their local communications. It is noticed that the design of HAN is applicable to multiple topology levels. If there are N levels, processes are divided into N groups. However, limited to the topology information obtained, I use two levels to demonstrate the design of HAN.

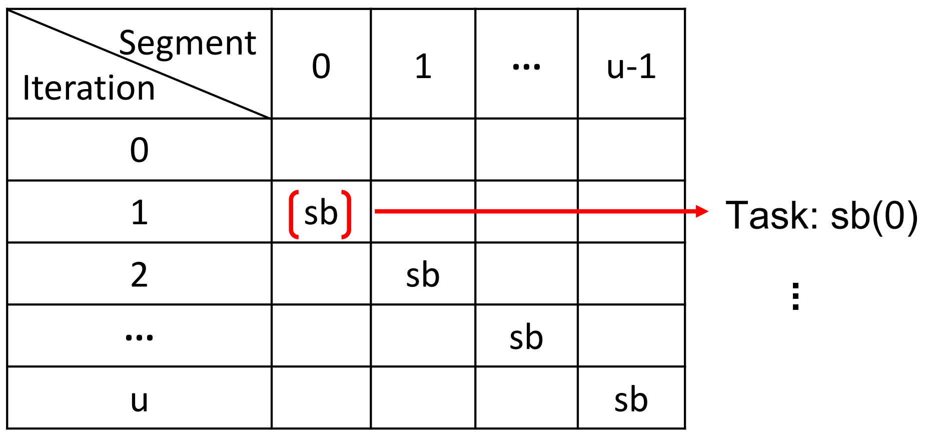
The HAN framework uses a task-based interface to organize and overlap the communications of different levels. It utilizes the pipelining technique [29], breaking the message down into smaller segments and sending them out in order, to increase the opportunity of overlapping communications. In the HAN framework, communications of segments are performed via tasks. To perform a hierarchical collective operation, each task contains one or more collective operations from different submodules on different levels. With the task-based design, the underlying submodules used for collective operations are interchangeable, allowing the HAN framework to easily adopt new submodules.

In the following sections, I use MPI_Bcast and MPI_Allreduce as examples to demonstrate how I design the MPI one-to-all and all-to-all collective operations in the HAN framework.



Iteration \ Segment	0	1	···	u-1
0	ib			
1	sb	ib		
2		sb	ib	
···			sb	ib
u				sb

(a) Node Leader Processes



Iteration \ Segment	0	1	···	u-1
0				
1	sb			
2		sb		
···			sb	
u				sb

(b) The Other Processes

Figure 5.6: Design of MPI_Bcast in HAN

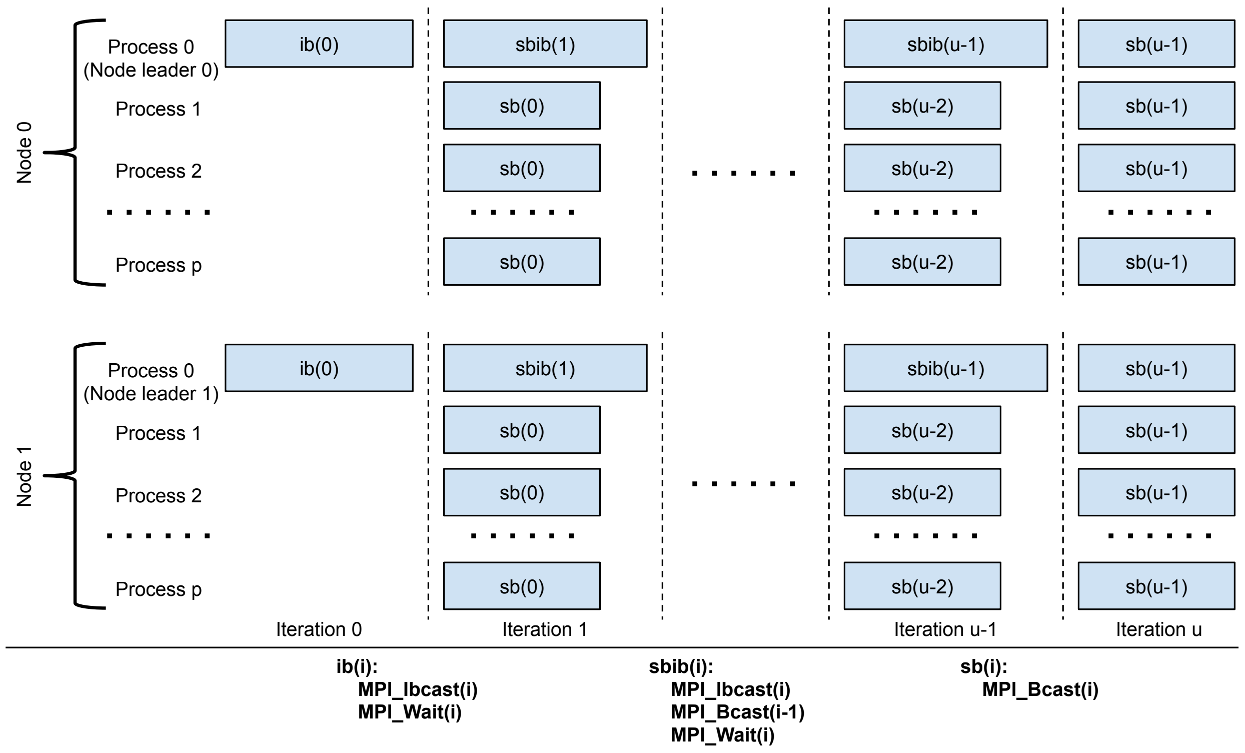


Figure 5.7: An Example of MPI_Bcast on two nodes

5.3.1 MPI_Bcast

MPI_Bcast is a widely used one-to-all collective operation, which propagates data from the root process to other processes within an MPI communicator. Figure 5.6 shows the implementation of MPI_Bcast in the HAN framework. From the root, each segment firstly goes through an inter-node broadcast (*ib*) to reach the node leaders; then, each node leader issues an intra-node broadcast (*sb*, *s* stands for shared memory) to distribute the segment to the other local processes. Since the *ib* and *sb* mainly occupy different hardware, these two broadcasts have the potential to overlap. To maximize this overlap, three types of tasks are defined in our implementation:

- Task $ib(i)$ represents an inter-node broadcast of segment i.
- Task $sbib(i)$ includes an intra-node broadcast of segment $i-1$ received in the previous iteration and an inter-node broadcast of segment i.
- Task $sb(i)$ represents an intra-node broadcast of segment i.

Assuming there are u segments in total. With the task-based design, to perform a hierarchical MPI_Bcast, node leaders execute $ib(0)$, $sbib(1)$, $sbib(2)$, ... $sbib(u-1)$ and $sb(u-1)$, and the other processes execute $sb(0)$, ... $sb(u-1)$, as in figure 5.6. This design is also shown in figure 5.7, which gives an example of MPI_Bcast on two nodes and p processes per node. At iteration 0, only node leaders participate in MPI_Bcast, and each of them executes a $ib(0)$. At iteration i, node leaders execute $sbib(i)$ and the other processes execute $sb(i-1)$. Finally, at iteration u, all the processes execute $sb(u-1)$ to transfer the last segment.

The Cost Model

To find the best configuration of MPI_Bcast in the HAN framework, it is crucial to have an accurate cost model. In the cost model, I consider the cost of a collective operation to be the longest time among all processes since the time for each process to finish the collective operation may be different depends on the implementations. This definition has been used by multiple cost models [12, 50], and it is the maximum value reported by Intel MPI Benchmark (IMB) [1] and OSU Benchmark [2].

The cost of MPI_Bcast is computed by aggregating the cost of each timestep in figure 5.6, so the time spent in node leader processes is:

$$\max_i(T_i(ib(0)) + T_i(sbib(1)) + ... + T_i(sbib(u-1)) + T_i(sb(u-1))) \tag{5.1}$$

and the time spent in non-node leader processes is:

$$\max_i(T_i(ib(0)) + T_i(sb(1)) + ... + T_i(sb(u-1)), \tag{5.2}$$

where u is the total number of segments. $T(t)$ is used to represent the cost of task t on each process, so $T_i(ib(0))$ means the cost of task $ib(0)$ on the process i. Usually, the cost of $sbib(x)$ is larger than $sb(x)$ since it has one more inter-node MPI_Ibcast to do; therefore, comparing equation 5.1 and 5.2, I use the cost of node leader process (equation 5.1) as the cost of MPI_Bcast.

Since $ib(0)$ is the first task, I assume each process issues it simultaneously. Hence, its cost can easily be measured by a simple benchmark shown in algorithm 9. The blue bars in figure 5.8 and figure 5.9 show the performance of $ib(0)$ on each node leader with rank 0 as the root process, when transferring 64 KB and 2 MB segments on 6 nodes with different configurations. From it, two following things can be observed:

1. different submodules and algorithms behave differently;
2. every node leader finishes ib at a different time.

Algorithm 9: Benchmark for ib(0)/sb(0)

```
  Output: result
1 result = 0;
2 for i = 0 to num_iters-1 do
3     MPI_Barrier();
4     t0 = MPI_Wtime();
5     ib(0)/sb(0);
6     t1 = MPI_Wtime();
7     result = result + t1 - t0;
8 result = result / num_iters;
```

The last task $sb(u-1)$ only contains an intra-node broadcast, which is independent of the processes on the other nodes. Since segment size is the same among all the segments, I use $T_i(sb(0))$ to represent $T_i(sb(u-1))$. The cost of $sb(0)$ can be measured as same as $ib(0)$, and the results are shown as the orange bars in figure 5.8 and figure 5.9.

$T_i(sbib(1)) + ... + T_i(sbib(u-1))$ contributes to the major cost of MPI_Bcast when the number of segments (u) is big enough. To get an accurate estimation of this part, there are two essential factors to be taken into consideration.

1. The first factor is the overlap of ib and sb. The ib mainly operates on the interconnects between nodes, while the sb communicates over the memory bus, which means these two broadcasts can overlap to some degree. In this way, $T_i(sbib(x))$ should be less than $T_i(ib(x)) + T_i(sb(x))$ for any process i. Previous research [8, 17] assumes the overlap

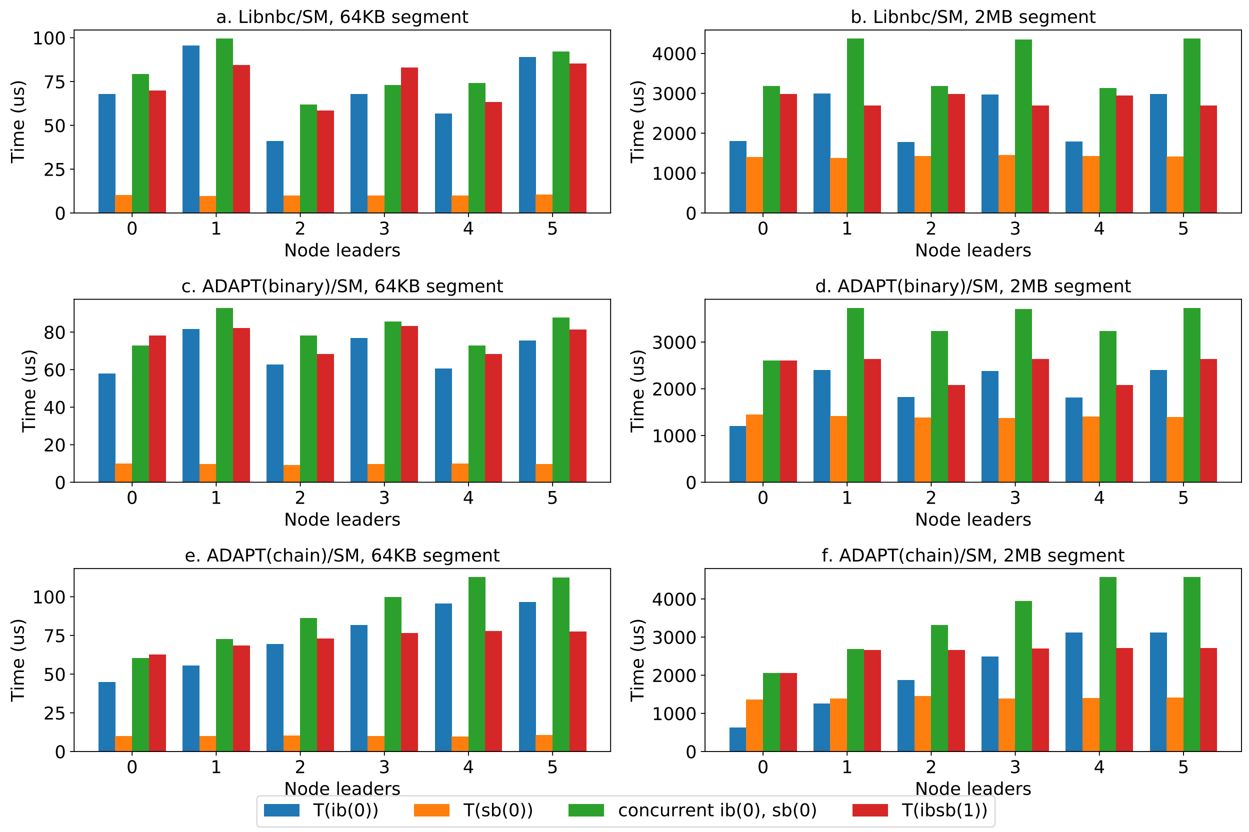


Figure 5.8: Cost of tasks (ib, sb, sbib) on NaCl (0 is the root), SM module

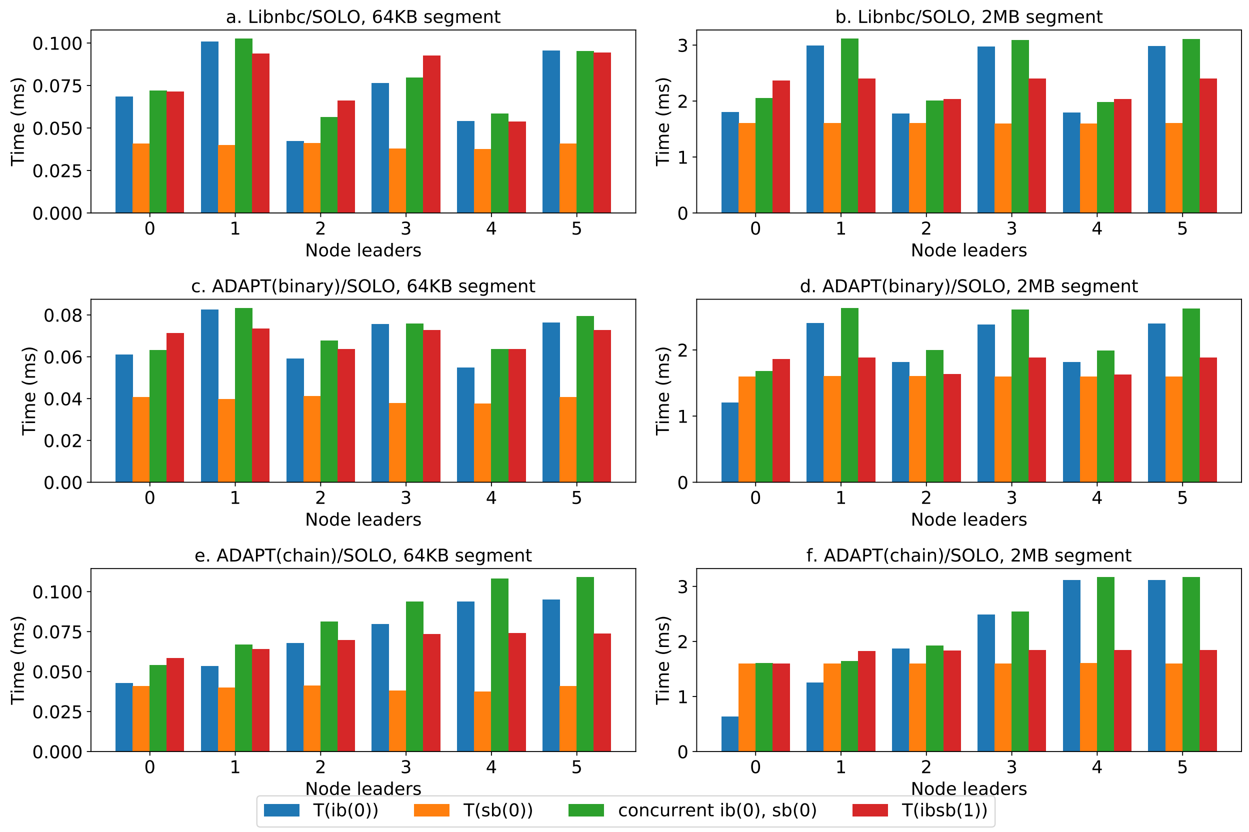


Figure 5.9: Cost of tasks (ib, sb, sbib) on NaCl (0 is the root), SOLO module

Algorithm 10: Benchmark for sbib(1)

```
    Output: result
1   result = 0;
2   for $i$ = 0 to num_iters-1 do
3       MPI_Barrier();
4       ib(0);
5       t0 = MPI_Wtime();
6       sbib(1);
7       t1 = MPI_Wtime();
8       result = result + t1 - t0;
9   result = result / num_iters;
```

of the communications on different levels is perfect, which suggests $T_i(sbib(x)) = max(T(ib(x)), T(sb(x)))$. However, it is not always the case. The overlap might not be perfect because of the following two reasons:

(a) the *ib* needs to push the data back to memory which could compete with the *sb* for the memory bus;

(b) from the implementation perspective, in single-threaded MPI, these two broadcasts share the same CPU resource to progress, which could affect the performance of both when they are running simultaneously.

The blue, orange and green bars in figure 5.8 and figure 5.9 show the cost of task $ib(0)$, $sb(0)$ and concurrent $sb(0)$ with $ib(0)$ (issue a non-blocking *ib* with an *sb* simultaneously and wait for them to complete), respectively. These two figures suggest that the overlap rate also depends on submodules. In figure 5.8, the submodule used to do *sb* is the SM module. We can observe that the green bar of each node leader is close to the sum of its blue bar and orange bar, which suggests the intra-node broadcast in the SM module does not overlap well with inter-node collective operations. While in figure 5.9, I use the SOLO module to perform the *sb*. When using the SOLO module, the overlap between *ib* and *sb* is much more significant, but still not perfect.

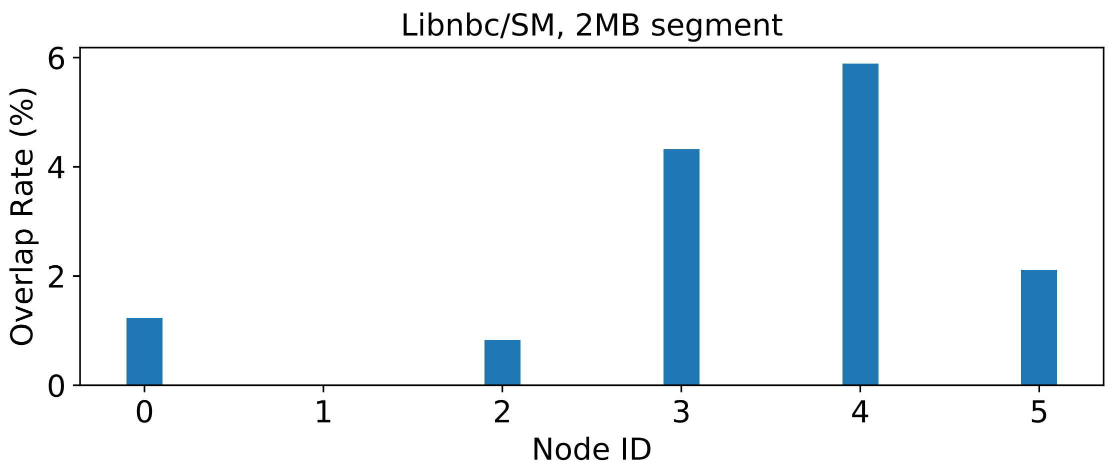


Figure 5.10: Overlap Rate of Concurrent Intra-node and Inter-node Collective Operations (6 nodes, 12 ppn, SM module)

To quantify the overlap of *ib* and *sb*, I define the overlap rate of an intra-node collective operation and an inter-node collective operation as listed below:

$$\max(0, \min(1, (t\_inter + t\_intra - t\_total)/ \min(t\_inter, t\_intra))) \tag{5.3}$$

In the above equation, t_inter is the cost of an inter-node collective operation, t_intra is the cost of an intra-node collective operation, and t_total is the overall time of these two operations when they are issued simultaneously.

Figure 5.10 shows the overlap rate of *ib* and *sb* on each node leader, with *ib* using Libnbc and *sb* using the SM module. The figure shows that the overlap rates are very low on the node leaders. The minimal overlap rate is on node leader 1 (0%), and the maximum is on node leader 4 (6%). The overlap rates are much higher when using the SOLO module, as presented in figure 5.11, all node leaders have more than 80% overlap. Even though the overlap rates of the SOLO module are very high, it can not reach 100%. Thus, neither $max(T_i(ib(x), T_i(sb(x)))$ nor $T_i(ib(x)) + T_i(sb(x))$ can be used to represent $T_i(sbib(x))$ in an accurate cost model.

2. The second factor is the operations happened before each *sbib* need to be considered to obtain an accurate cost of *sbib* since each *sbib* is issued at a different timestamp

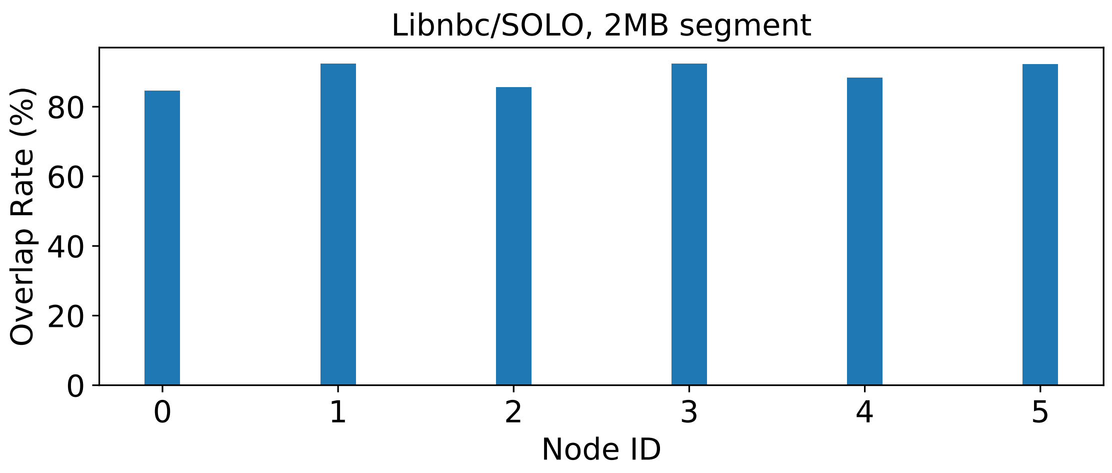


Figure 5.11: Overlap Rate of Concurrent Intra-node and Inter-node Collective Operations (6 nodes, 12 ppn, SOLO module)

on each process. The blue bars in figure 5.8 and figure 5.9 show the cost of $ib(0)$, the operation before $sbib(1)$, on different node leaders. As seen in the figure, no matter what submodules are used, the node leader processes finish the task $ib(0)$ at different timestamps. Hence, to simulate the different starting time of $sbib(1)$, an $ib(0)$ is issued before it using the benchmark in algorithm 10, and the results of this benchmark is shown as the red bars in figure 5.8 and figure 5.9. The performance differences between red bars and blue bars in the figures prove the importance of the previous tasks since the only difference between the two is whether there is an $ib(0)$ before timing the task $sbib(1)$. Therefore, to get the accurate cost of $sbib(1)$, task $ib(0)$ needs to be executed before the timing, and to get the accurate cost of $sbib(2)$, task $ib(0)$ and $sbib(1)$ need to be performed. In this way, to calculate $T_i(sbib(1)) + ... + T_i(sbib(u-1))$, we need to know the cost of $sbib(i)$ where $1 \leq i \leq u-1$, and to get the cost of each $sbib(i)$, all the previous tasks from $ib(0)$, $sbib(1)$ to $sbib(i-1)$ need to be executed. This procedure is highly expensive and contains a lot of redundant tasks.

Figure 5.12 shows the cost of $sbib(i)$ where $1 \leq i \leq 8$ with different algorithms and submodules on node leader 0. All the sub-figures in the figure show a similar trend that after the first few tasks, the cost of $sbib$ is stabilized. This is because when executing the first few $sbib$s, the pipeline of $sbib$ is not constructed, which could leads to some

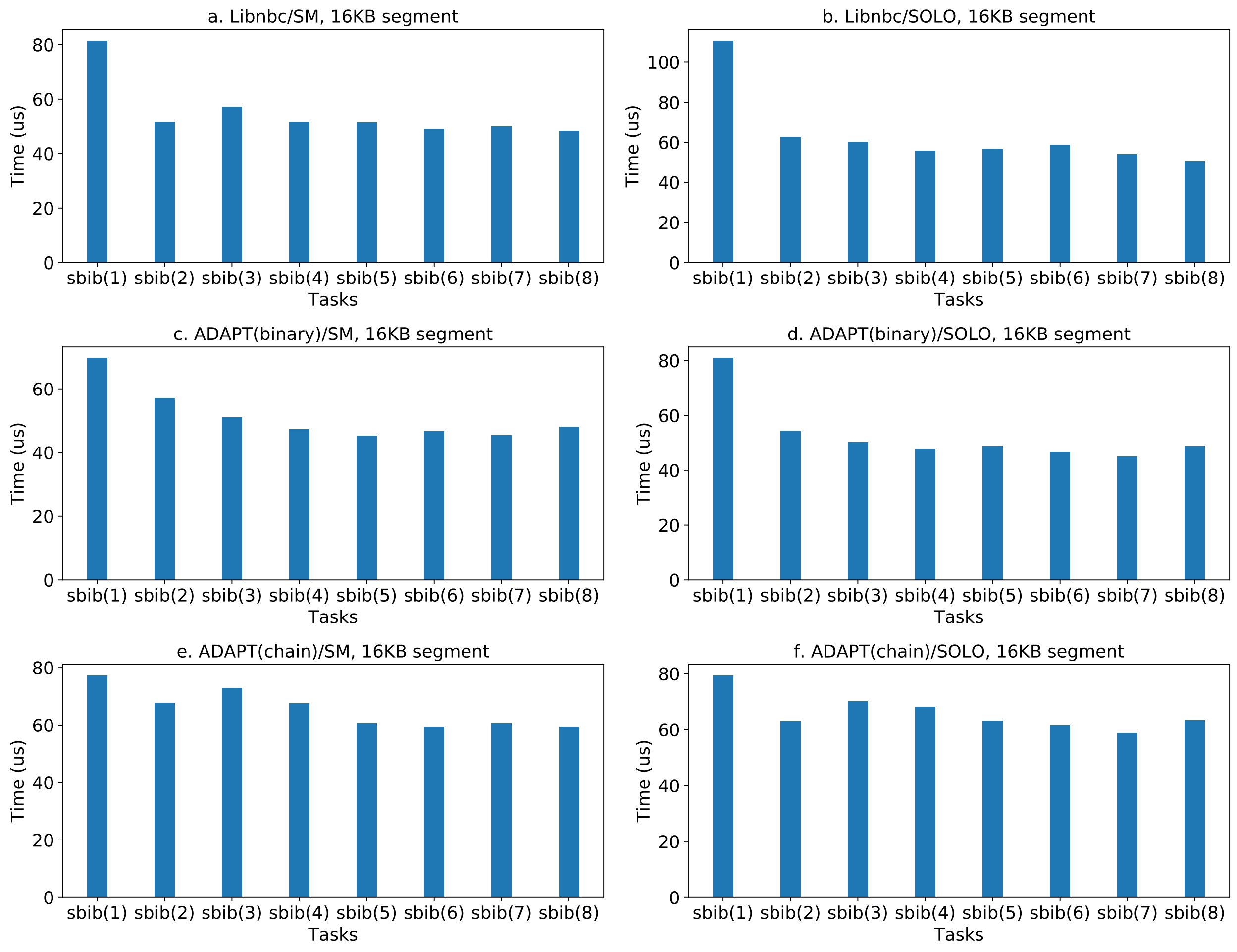


Figure 5.12: Cost of tasks on one node leader on NaCl (64 nodes, 12 ppn)

delays. Once the pipeline is fully constructed, the cost of *sbib* is stabilized. To save the time of the benchmarking and avoid repeatedly executing tasks, I use the stabilized cost ($sbib(s)$) times $u-1$ to estimate the time of $T_i(sbib(1)) + ... + T_i(sbib(u-1))$. Therefore, equation 5.1 can turn into:

$$\max_i(T_i(ib(0)) + (u-1)T_i(sbib(s)) + T_i(sb(u-1))) \tag{5.4}$$

Model Validation

Figure 5.13 shows the comparison of the estimated time mathematically calculated from the cost model and the actual time of doing a 4MB MPI_Bcast on 64 nodes with 12 processes per node with different combinations of submodules, algorithms, and segment sizes. The experiment is conducted on NaCl, which has 66 compute nodes connected by Infiniband QDR, and each node has two 2.8GHz Intel Xeon X5660 and 12 GB memory. As seen in the figure, the cost model is accurate in most cases. Even though in the cases where the prediction is not that accurate, such as when segment size is 16KB in figure 5.13.e and figure 5.13.f, the trends of the estimated and actual time still match well, and that can help to find the best configuration of MPI_Bcast. When comparing the cost of MPI_Bcast of all configurations, the best configuration estimated by the cost model and the actual best configuration are the same, which is to use 128KB segment size with the binary algorithm in ADAPT for inter-node communication and the SOLO submodule for intra-node communication. It shows that the cost model can be used for the autotuning component of the HAN framework, and I discuss the details of autotuning in the next chapter.

5.3.2 MPI_Allreduce

In this section, as an example of all-to-all collective operations, I introduce the implementation of MPI_Allreduce in the HAN framework. As seen in figure 5.14, a hierarchical MPI_Allreduce requires four steps to transfer one segment:

1. intra-node reduction (sr),

2. inter-node reduction (ir),

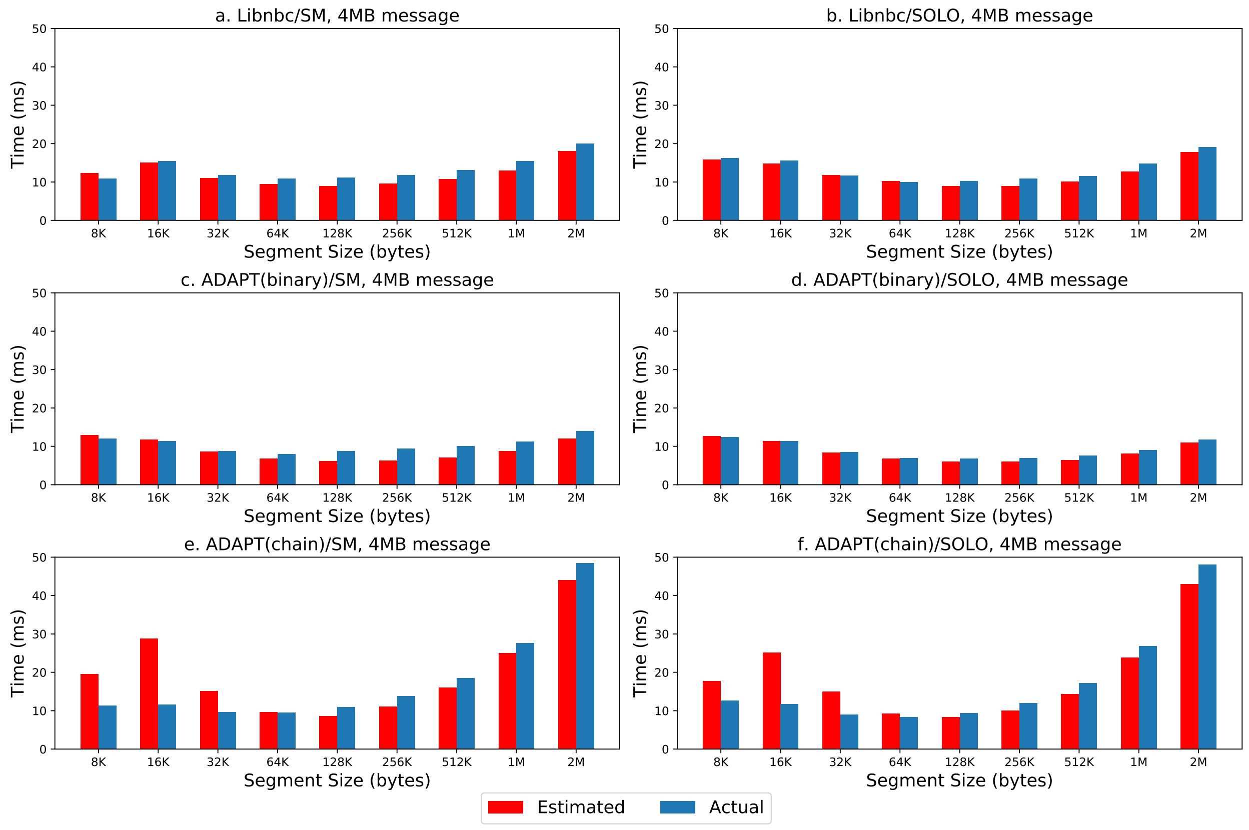
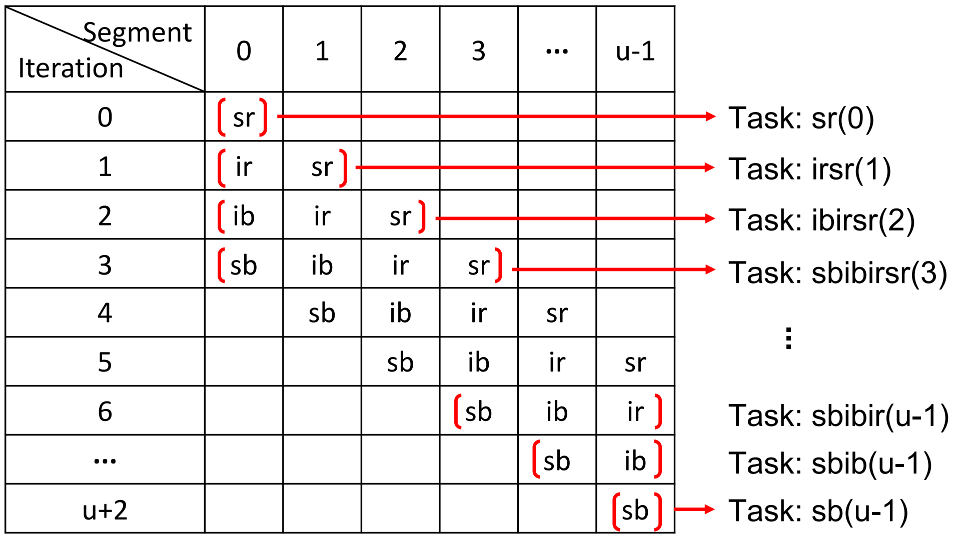
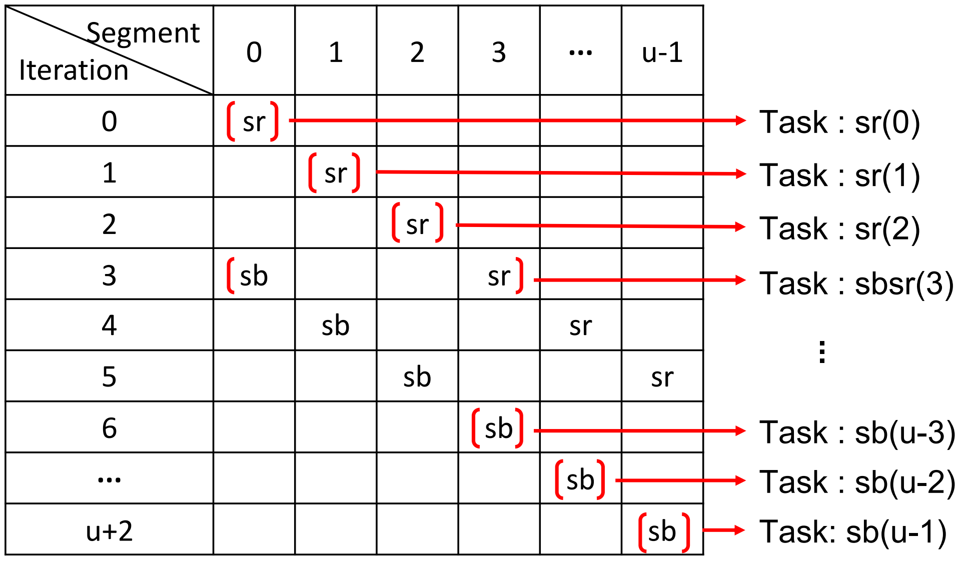


Figure 5.13: Performance of MPI_Bcast on 64 nodes with the combinations of different submodules on NaCl



Iteration \ Segment	0	1	2	3	…	u-1
0	sr					
1	ir	sr				
2	ib	ir	sr			
3	sb	ib	ir	sr		
4		sb	ib	ir	sr	
5			sb	ib	ir	sr
6				sb	ib	ir
…					sb	ib
u+2						sb

(a) Node Leader Processes



Iteration \ Segment	0	1	2	3	…	u-1
0	sr					
1		sr				
2			sr			
3	sb			sr		
4		sb			sr	
5			sb			sr
6				sb		
…					sb	
u+2						sb

(b) The Other Processes

Figure 5.14: Design of MPI_Allreduce in HAN

3. inter-node broadcast (*ib*),
4. intra-node broadcast (*sb*).

In the implementation of MPI_Bcast, I have discussed how my design overlaps the collective operations of different levels (i.e. *ib* with *sb*); additionally, if the interconnects are full-duplex, a collective operation can overlap with another one within the same level. For example, *ir* and *ib* can be overlapped if they are on different directions of the same inter-node network. The comparison of the performance of *ib*, *ir*, and concurrent *ib* and *ir* of different submodules and algorithms is shown in figure 5.15, and it suggests a high degree of overlap. To maximize the opportunity of such overlap, when it is possible to specify the algorithm, the HAN framework selects the same algorithm with the same root to perform *ir* and *ib*; if not, the overlap opportunity is up to the inter-node submodule. Considering both kinds of overlap, I define the following tasks in the MPI_Allreduce:

- Tasks $sr(i)$ and $sb(i)$ represent an *sr* and an *sb* of segment i, respectively.
- Task $irsr(i)$ includes an *ir* of segment $i-1$ and an *sr* of segment i.
- Task $ibirsr(i)$ contains an *ib* of segment $i-2$, an *ir* of segment $i-1$ and an *sr* of segment i.
- Task $sbibirsr(i)$ is consisted of an *sb* of segment $i-3$, an *ib* of segment $i-2$, an *ir* of segment $i-1$ and an *sr* of segment i.
- Task $sbibir(i)$ includes an *sb* of segment $i-2$, an *ib* of segment $i-1$ and an *ir* of segment i.
- Task $sbib(i)$ contains an *sb* of segment $i-1$ and an *ib* of segment i.
- Task $sbsr(i)$ is only executed in non-node leader processes. It receives reduced segment $i-3$ from its leader process via an *sb*, and then reduce segment i to its leader process with an *sr*.

An example of the MPI_Allreduce on two nodes is shown in figure 5.16, with p processes per node. At iteration 0, all the processes execute $sr(0)$; at iteration 1, node leaders execute

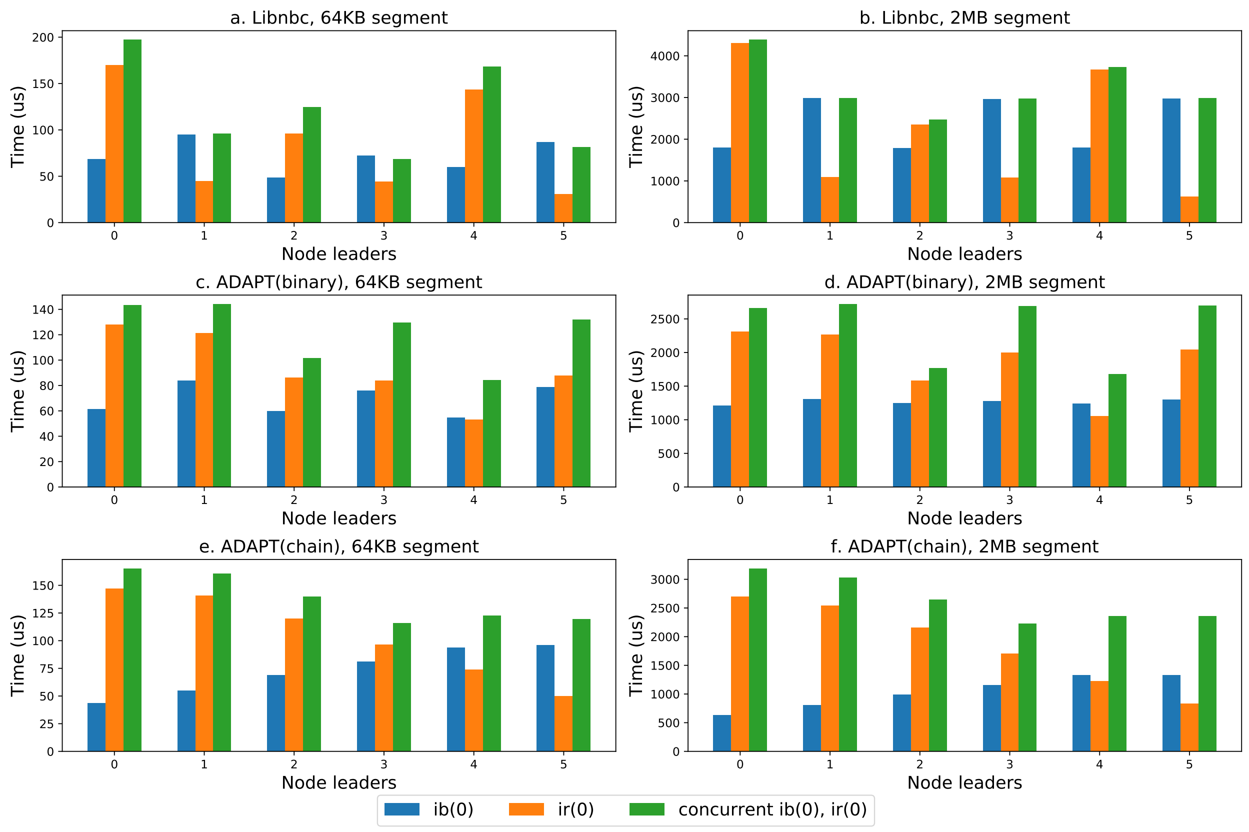
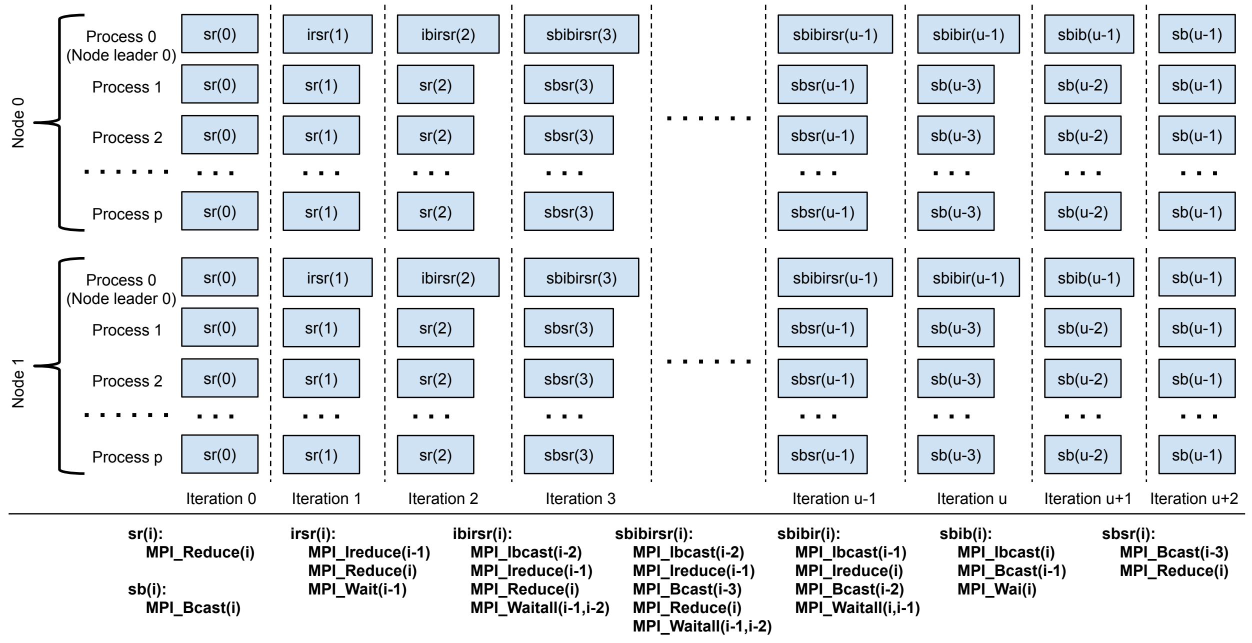


Figure 5.15: The overlap between *ib* and *ir* (0 is the root)



```
sr(i):
    MPI_Reduce(i)

sb(i):
    MPI_Bcast(i)

irsr(i):
    MPI_Ireduce(i-1)
    MPI_Reduce(i)
    MPI_Wait(i-1)

ibirsr(i):
    MPI_Ibcast(i-2)
    MPI_Ireduce(i-1)
    MPI_Reduce(i)
    MPI_Waitall(i-1,i-2)

sbibirsr(i):
    MPI_Ibcast(i-2)
    MPI_Ireduce(i-1)
    MPI_Bcast(i-3)
    MPI_Reduce(i)
    MPI_Waitall(i-1,i-2)

sbibir(i):
    MPI_Ibcast(i-1)
    MPI_Ireduce(i)
    MPI_Bcast(i-2)
    MPI_Waitall(i,i-1)

sbib(i):
    MPI_Ibcast(i)
    MPI_Bcast(i-1)
    MPI_Wai(i)

sbsr(i):
    MPI_Bcast(i-3)
    MPI_Reduce(i)
```

Figure 5.16: An Example of MPI_Allreduce on two nodes

$irsr(1)$, and the other processes execute $sr(1)$; and at iteration 2, node leaders execute $ibirsr(2)$, and the other processes execute $sr(2)$. From iteration 3 to iteration $u-1$, node leaders execute $sbibirsr(i)$ and the other processes execute $sbsr(i)$, where $3 \leq i \leq u-1$. Then at the last three iterations, node leaders execute $sbibir(u-1)$, $sbib(u-1)$, and $sb(u-1)$, while the other processes execute $sb(u-3)$, $sb(u-2)$, and $sb(u-1)$.

The Cost Model

Similar to MPI_Bcast, I estimate $sbibirsr(3) + sbibirsr(4) + ... + sbibirsr(u-1)$ with the stabilized cost of $sbibirsr$ ($T_i(sbibirsr(s))$). In this way, the cost of MPI_Allreduce is:

$$\begin{aligned}&\max_i(T_i(sr(0)) + T_i(irsr(1)) + T_i(ibirsr(2)) + (u-3) * T_i(sbibirsr(s))\\&+T_i(sbibir(u-1)) + T_i(sbib(u-1))) + T_i(sb(u-1))\end{aligned} \tag{5.5}$$

Then I use similar benchmarks as in algorithm 9 and algorithm 10 to measure the cost of the tasks. Task $ibirsb(u-1)$ and $sbib(u-1)$ are issued after $u-3$ times of task $sbibirsr$; however, when benchmarking them, it is not realistic to repeat $sbibirsr$ $u-3$ times. Instead, I modify algorithm 10 to keep running $sbibirsr$ until every process reaches a relatively stable state before measuring $ibirsb(u-1)$ and $sbib(u-1)$.

Model Validation

Figure 5.17 compares the time of MPI_Allreduce estimated by the cost model against the actual time. Similar to MPI_Bcast, with the estimated time, I can estimate the optimal configuration for MPI_Allreduce on a 4MB message is to use 1MB segments with the binary algorithm in ADAPT for inter-node communications and use the SOLO framework as submodule for intra-node communications. It exactly matches the best configuration of the actual time.

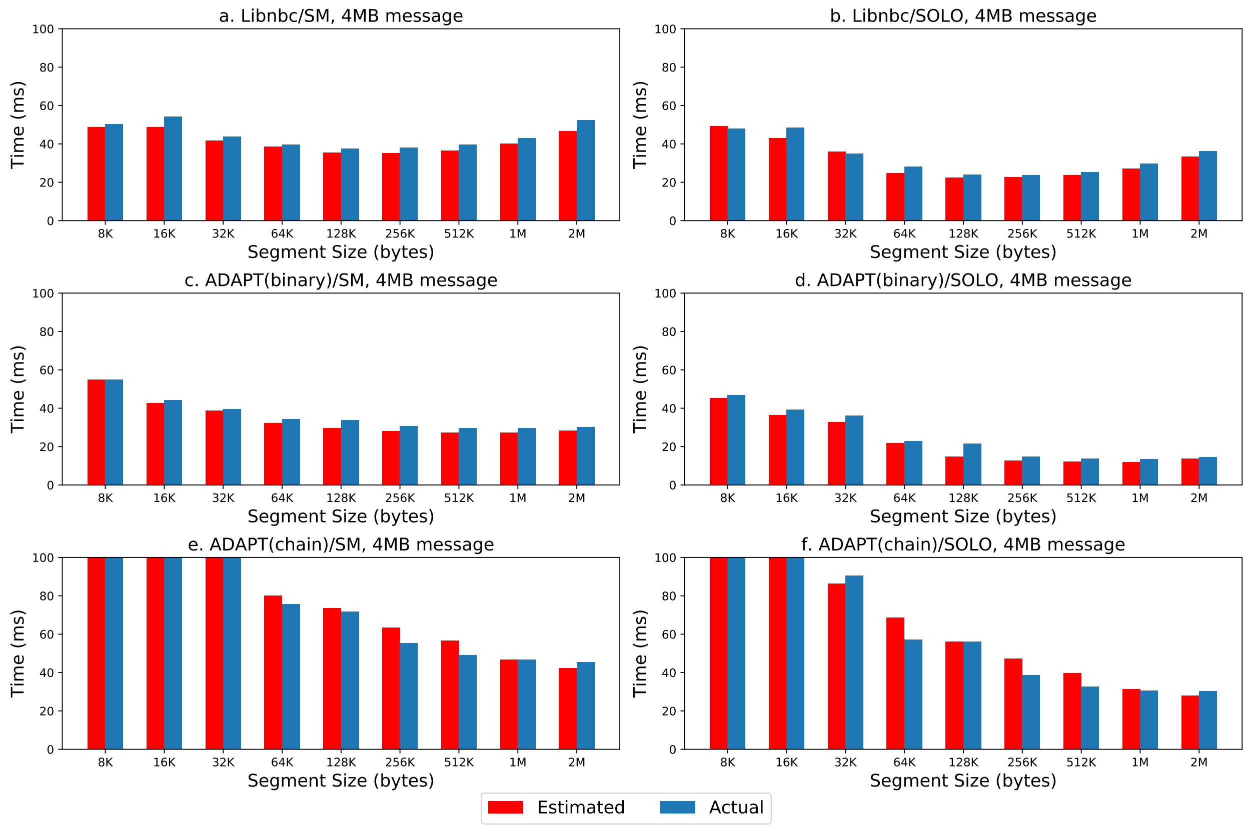


Figure 5.17: MPI_Allreduce on 64 nodes with combination of different submodules on NaCl

Chapter 6

Autotuning of Collective Communication Operations

6.1 Overview

Some submodules in the HAN framework, such as ADAPT, provide different algorithms for each type of collective operation. For example, MPI_Ibcast in ADAPT contains multiple algorithms, such as chain, binary tree, and binomial tree. For each algorithm, the underlying configurations, such as segment size, can also affect the performance of the operations. As demonstrated in previous research [59, 19], the performance penalty is huge for collective operations choosing incorrect configurations. Therefore, to achieve decent performance, an autotuning component is needed in the HAN framework to search for the optimal configuration.

This chapter introduces the autotuning component in the HAN framework, which merges two existing autotuning methods by using a novel cost model based on the costs of tasks introduced in chapter 5. The experiments in this chapter show that the autotuning component is able to significantly reduce the search space and provide accurate estimations. In section 6.2, I present the design of the autotuning component and prove its effectiveness; and in section 6.3, I evaluate the autotuned HAN framework on two clusters.

Table 6.1: Inputs of Automatic Tuning

Symbol	Description
n	Number of Nodes
p	Number of Processes per Node
m	Message Size
t	Collective Operation Type (Bcast, Reduce, ...)

6.2 Task-Based Autotuning

Because of the high cost of exhaustive search and the low accuracy of the conventional cost models introduced in chapter 1, I create a task-based autotuning approach using new cost models based on the empirical costs of tasks. Costs of tasks are obtained by benchmarking submodules. Since submodules are tightly coupled in the HAN framework, testing the performance of an individual submodule is not sufficient to represent the overall performance. To accurately estimate the performance, I benchmark the submodules when they are working together and use these results in the cost model, introduced in chapter 5, to estimate the cost of a collective operation and perform autotuning.

The goal of autotuning is to find the best configuration (submodules, algorithms, segment sizes, etc.) for any given input. Generally, there are two steps in autotuning:

1. The first step is to find the optimal configuration for some inputs to generate a lookup table. As shown in table 6.1, an input contains four entries: number of nodes n, number of processes per node p, message size m, and the collective operation type t. The output entries of the lookup table are shown in table 6.2. Usually, m is continuous, but it is impractical to test every message size; thus, most approaches use discrete message sizes such as 4B, 8B, 16B, 32B, ... to sample the continuous value and form a search space. The same sampling method can be applied to other entries such as n and p.

2. The second step is to use the lookup table from the previous step to generate decisions for any inputs (n, p, m and t). As message sizes in the lookup table are not continuous, if the input message size falls between two message sizes in the lookup table, the autotuning process needs to find the optimal configuration for it.

Table 6.2: Available configurations in the HAN module (MPI_Bcast and MPI_Allreduce, *iralg* and *irs* are only used for MPI_Allreduce)

Symbol	Description
fs	Segment Size in the HAN module
imod	submodule used for inter-node
smod	submodule used for intra-node
ibalg	Inter-node Bcast Algorithm if supported
iralg	Inter-node Reduce Algorithm if supported
ibs	Inter-node Bcast Segment Size if supported
irs	Inter-node Reduce Segment Size if supported

Some previous research focuses on the second step, where many methods such as quadtree encoding [52], decision trees [51] have been studied to improve its accuracy. However, the first step, which takes a major part of the time and is the foundation to ensure the accuracy of the second step, has not been well studied. Hence, in this dissertation, I focus on the first step to minimize the cost of autotuning and improve its accuracy.

The most straightforward approach to do the first step is to perform an exhaustive search for each input in table 6.1, which tests every possible configuration and then find the fastest one. Take MPI_Bcast for an example. Assume the sizes of the search spaces of messages, segment sizes, nodes, processes per node are, M, S, N, and P, respectively, and the total selections of algorithms are A (including submodules $\times$ algorithms per submodule). Exhaustive search needs to run all possible combinations of S and A for each input in M, N, and P, pick the fastest result and record the segment size and algorithm in the lookup table for that input. Therefore the size of the whole search space is $M \times S \times N \times P \times A$. Since an MPI_Bcast has to be run for every search, the total searching needs to run MPI_Bcast $M \times S \times N \times P \times A$ times, which is a highly expensive process. The orange bar in figure 6.1 shows the cost of autotuning on NaCl with exhaustive search, which takes around 48 hours. The purple, brown, and orange bars in figure 6.2 show the median, average, and best cost of MPI_Bcast and MPI_Allreduce of all possible configurations, using exhaustive search. All autotuning experiments are conducted on NaCl other than larger-scale machines used in section 6.3, as the exhaustive search cannot complete within a job limitation on those

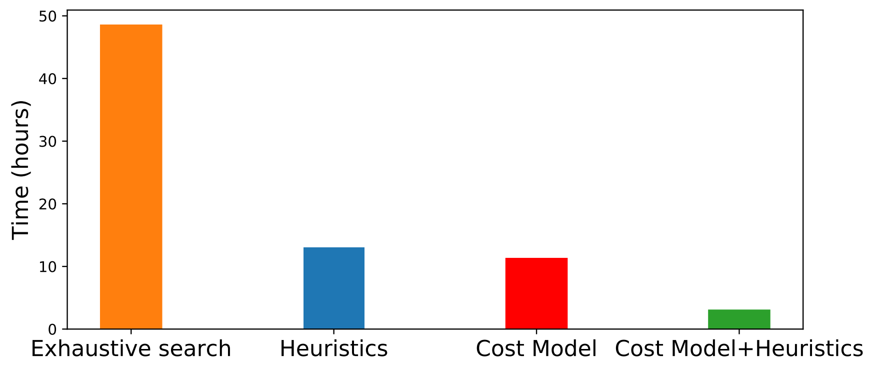


Figure 6.1: Time of total searches of MPI_Bcast and MPI_Allreduce on Nacl

machines. As seen in the figure, both the median and average time are much higher than the best, which indicates the importance of finding the optimal configuration.

With the cost model introduced in section 5.3, the task-based autotuning approach only needs to measure the cost of each task and use its cost to calculate the overall cost of a collective operation. As every task operates on one segment, the search space needed for one task are S, N, P, and A. Suppose there are T types of tasks (3 for MPI_Bcast and 7 for MPI_Allreduce); therefore, the size of the whole search space of my approach becomes $T \times S \times N \times P \times A$. For different message size M, the cost of tasks are reused; hence, compared to the previous approach, the task-based autotuning is able to reduce message size M, which is one of the largest search spaces, to a constant T. Besides the smaller search space, the cost of performing each search is also much shorter since tasks are just a part of the whole MPI collective operations. Also, the cost of tasks can be reused for different types of collective operations, such as the task *sbib* is in both implementations of MPI_Bcast and MPI_Allreduce. With these improvements, the cost of autotuning is significantly reduced. As the red bar in the figure 6.1, the task-based autotuning, reduces the cost of search by 77% as compared to the exhaustive search. Even though with much fewer searches, my approach can still estimate the optimal configuration accurately. The red bars in figure 6.2 show the best performance of MPI_Bcast and MPI_Allreduce with the configurations estimated by the task-based cost model. The results indicated that the task-based autotuning approach produces a similar configuration to the actual best configuration obtained by an exhaustive search (orange bars).

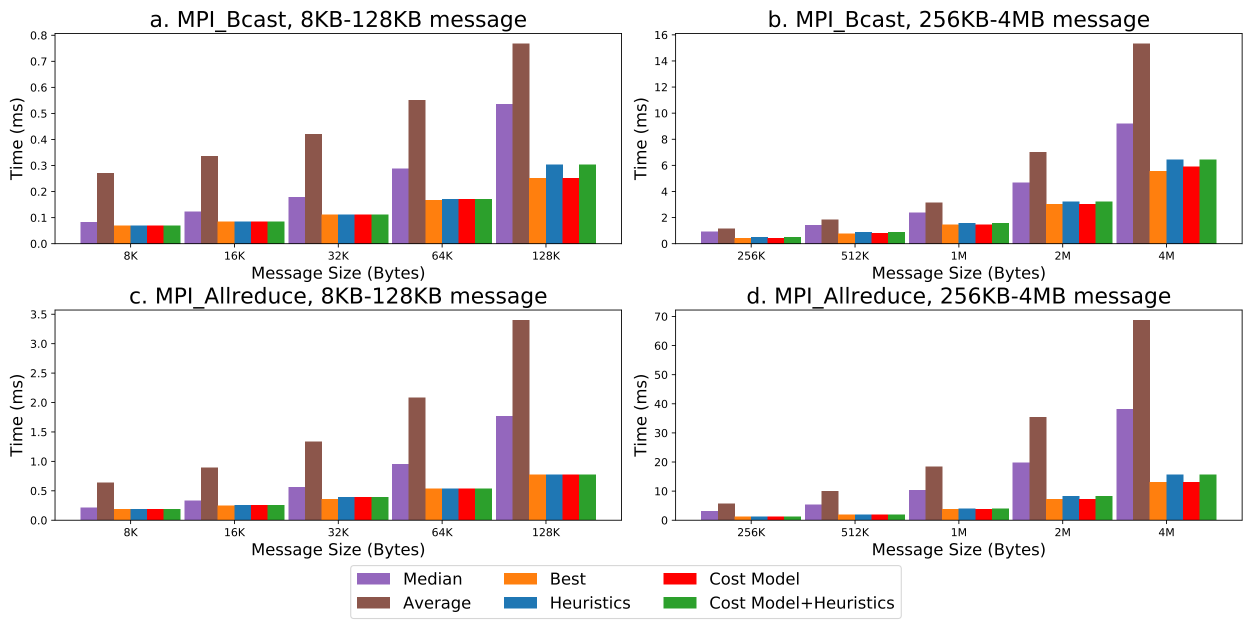


Figure 6.2: Performance of MPI_Bcast and MPI_Allreduce on 64 nodes with different configurations of sub-modules on NaCl

As previous research [50, 59] suggests, using heuristics is an effective way to reduce the search space. In the HAN framework, with my understanding of the submodules and the algorithms, I create multiple heuristic strategies. For example, one of the heuristics is to only use the SOLO submodule when the segment size is big than 512 KB since I know the SM submodule generally has better performance than the SOLO for small messages. Besides limiting the selections of submodules, the algorithm selections are also limited heuristically. An example is the chain algorithm in ADAPT can perform well only when there are enough segments for pipelining, so the chain algorithm is only used when there are more than eight segments. The blue bars in figure 6.1 shows the searching time on NaCl of just using the heuristics, which takes 26.8% time of the exhaustive search. Heuristics can be combined with cost model by using heuristic methods on the benchmarking of tasks, to further reduce the search space. The time is shown as the green bars in figure 6.1, which only takes 4.3% time of the exhaustive search. However, by using heuristic approaches, additional assumptions are made to narrow down the search space, which might result in lower accuracy of selecting the best configurations. The blue and green bars in figure 6.2 show the results of adding heuristics, suggesting that heuristics produce less accurate results compared to the original approach in the HAN framework. To balance the search speed and accuracy, the HAN framework provides an option for users to enable or disable the heuristics based on their needs.

In conclusion, while can select the good configurations, with the help of the cost model, the autotuning component in the HAN framework can greatly reduce the time spends on searching the optimal configurations.

6.3 Evaluation

6.3.1 Experimental Setup

In this section, I evaluate the autotuned HAN framework on two high-performance computers: Shaheen II and Stampede2, and compare it with other state-of-the-art MPI implementations using benchmarks and applications.

Shaheen II is a Cray XC40 system equipped with dual-socket Intel Haswell 16 cores CPUs running at 2.3GHz and 128GB 2300MHz DDR4 memory. The interconnects is Cray Aries with Dragonfly topology. On Stampede2, I use the Intel Skylake compute nodes; each node has 48 cores with two sockets, and 192GB 2.67GHz DDR4 memory. The nodes are connected via the Intel Omni-Path network.

The HAN framework is implemented on top of Open MPI 4.0.0. Hence, for fair comparisons, it is compared with the default module of Open MPI 4.0.0 on both machines. It is noticed that the default module is tuned with conventional methods [19], and the HAN framework is tuned with the task-based autotuning approach. Additionally, the HAN framework is compared with the system built-in Cray MPI 7.7.0 on Shaheen II, and Intel MPI 18.0.2 and MVAPICH2 2.3.1 on Stampede2.

6.3.2 Benchmark Results

IMB [1] is used to compare the HAN framework against other MPI implementations using MPI_Bcast and MPI_Allreduce, representing two widely used communication patterns: one-to-all and all-to-all. I experiment on two types of message sizes: small and large. The range of small message experiments is up to 128K, which represents message sizes in most scientific applications. Experiments on large messages start from 128KB to 128MB, which represents message sizes in applications processing a large amount of data such as machine learning and data analytics.

MPI_Bcast

Figure 6.3 presents the time of MPI_Bcast with 4096 processes on Shaheen II. Even though both HAN and the default framework in Open MPI are tuned, HAN still outperforms the default framework in Open MPI significantly: up to 4.72x and 7.35x speed up on small and large messages, respectively, thanks to the task-based hierarchical implementation and accurate cost models.

However, the HAN framework is slightly slower than Cray MPI on small messages. To better understand the performance gap, I measure the point-to-point communication performances of both Open MPI and Cray MPI using Netpipe [56]. In most MPI implementations, collective communications rely on the underlying point-to-point routines to transfer data between processes; therefore, the performance of point-to-point communications directly impacts the speed of collective communications. As seen in figure 6.4, when the message size is between 512B and 2MB, Open MPI has less bandwidth comparing to Cray MPI especially for the messages range from 16KB to 512KB, which could explain the performance differences for the small message on figure 6.3. On larger message sizes, Open MPI and Cray MPI both reach the same peak point-to-point communication performance; and in these cases, the HAN framework outperforms Cray MPI up to 2.32x thanks to the communication overlap of different hardware levels.

Figure 6.5 exhibits the performance of MPI_Bcast with 1536 processes on Stampede2. On this machine, the HAN framework gives the best performance on both small and large messages. It achieves up to 1.15X, 2.28X, 5.35X speedup on small messages, and up to 1.39X, 3.83X, 1.73X speed up on large messages against Intel MPI, MVAPICH2, and default Open MPI, respectively.

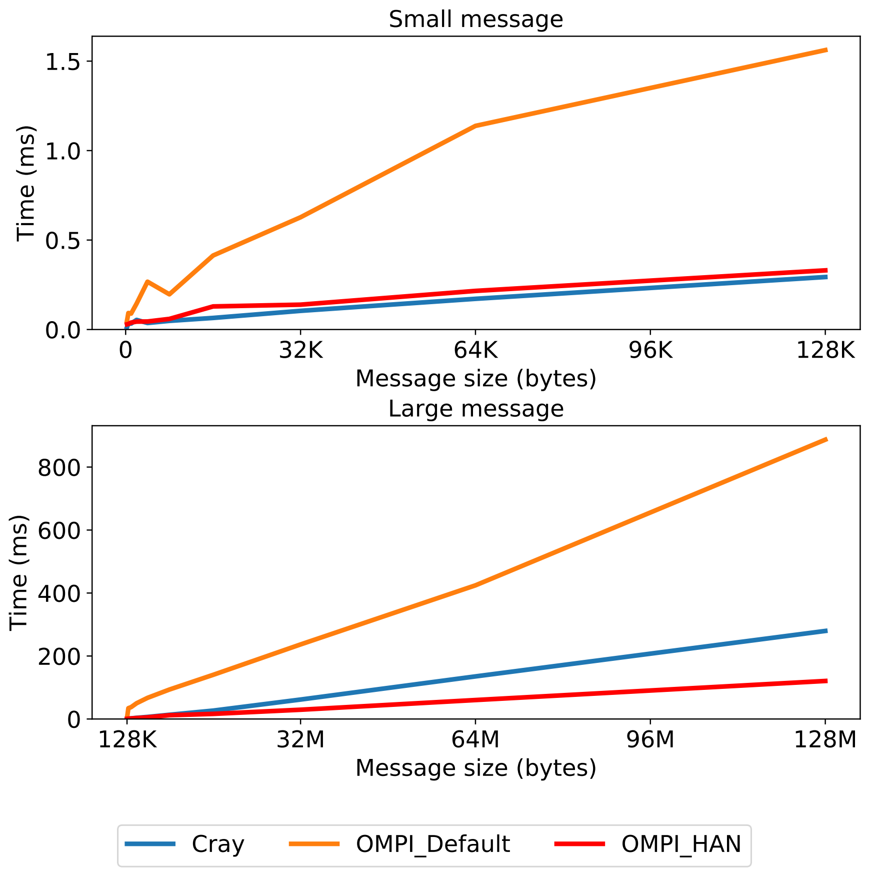


Figure 6.3: Performance MPI_Bcast (Shaheen II, 4096 processes)

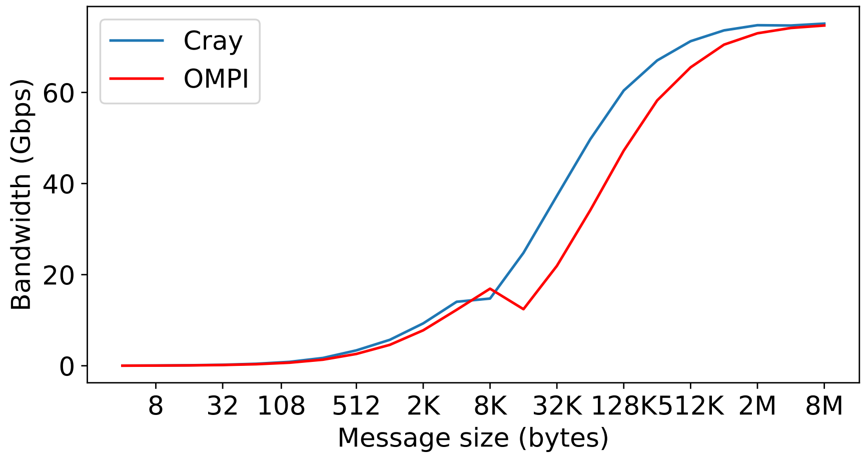


Figure 6.4: Bandwidth of MPI point-to-point communication routines (Shaheen II, 4096 processes)

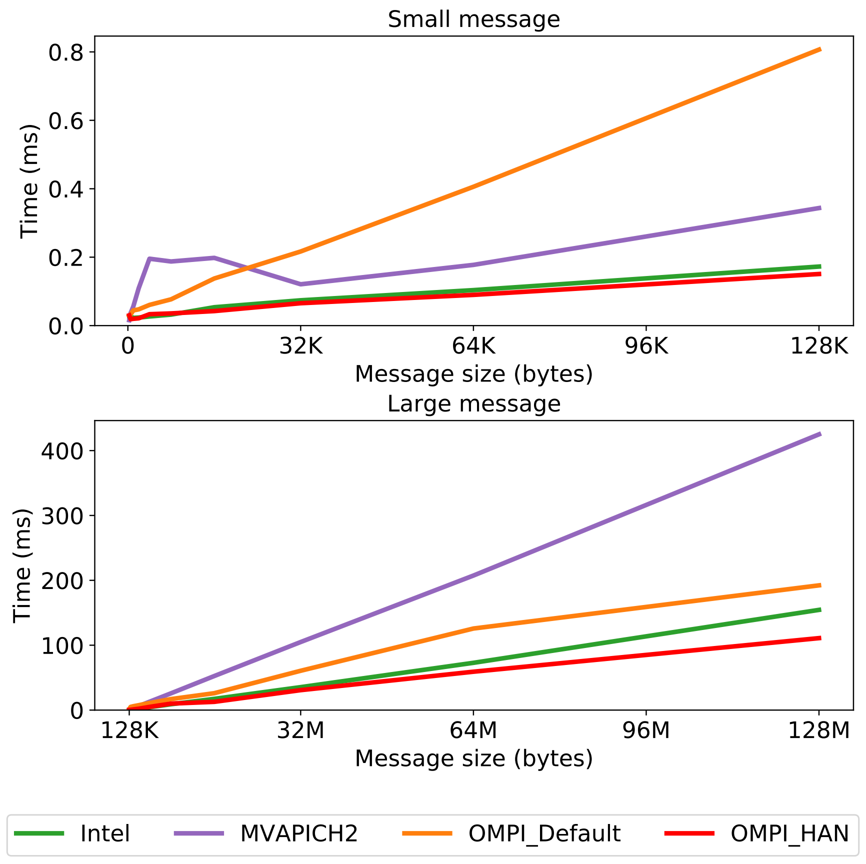


Figure 6.5: Performance of MPI_Bcast (Stampede2, 1536 processes)

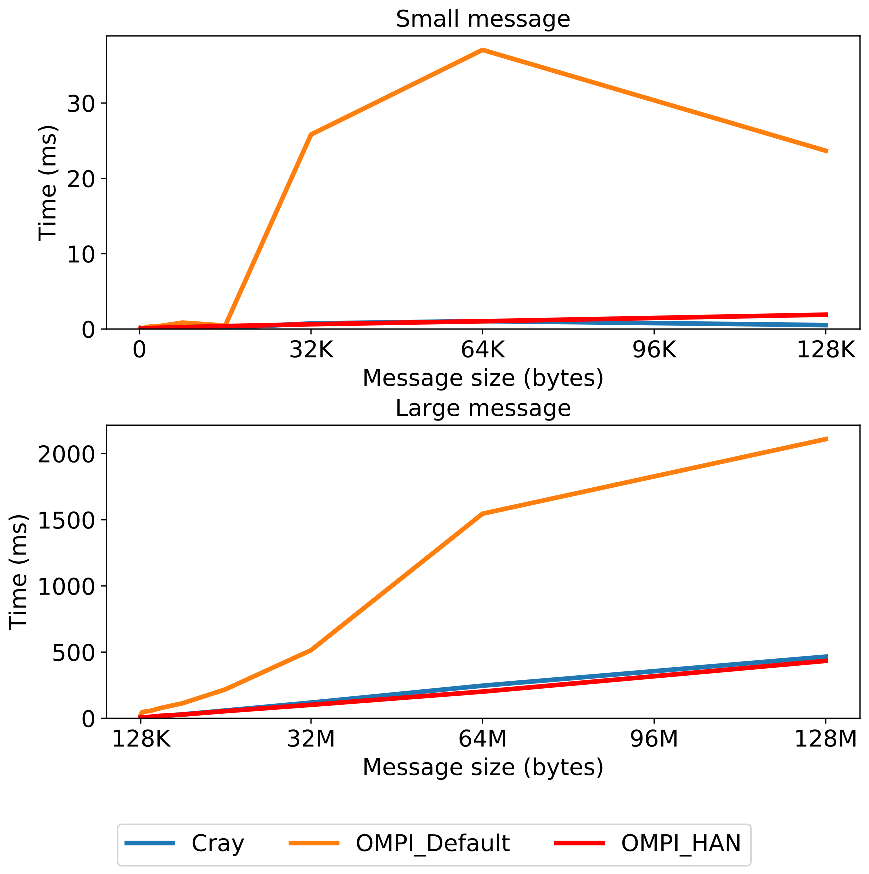


Figure 6.6: Performance of MPI_Allreduce (Shaheen II, 4096 processes)

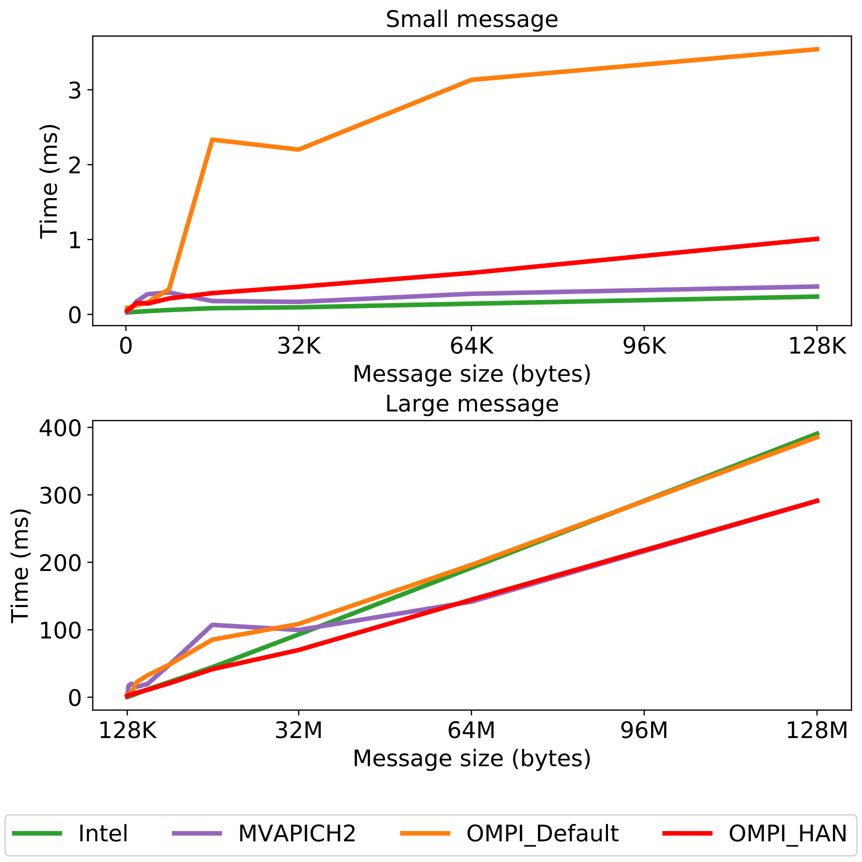


Figure 6.7: Performance of MPI_Allreduce (Stampede2, 1536 processes)

MPI_Allreduce

Figure 6.6 and figure 6.7 present the cost of MPI_Allreduce with 4096 processes on Shaheen II and 1536 processes on Stampede2, respectively. Compared with the default Open MPI, my design shows significant improvements in all cases. Compared with other state-of-the-art MPI implementations, the HAN framework shows some improvements with larger size messages. On Shaheen II, the HAN framework shows better performance than Cray MPI after the message size is larger than 2MB and eventually achieves up to 1.12X speedup. On Stampede2, my design is the fastest when message size is between 4MB and 64 MB, and afterward, it gives similar performance as MVAPICH2, but still faster than others.

Besides the point-to-point communication performance discussed in the previous section, another important factor that affects the performance of MPI_Allreduce is the cost of reduction operations. Among the four submodules currently used in the HAN framework, only SOLO and ADAPT utilize AVX instructions [25] to boost the performance of reduction operations. However, the designs of these two submodules lead to high overhead on small messages. Hence, HAN selects Libnbc and SM to perform MPI_Allreduce on small messages; unfortunately, neither of them supports AVX instruction, leading to sub-optimal performance as compared to other MPI implementations. It shows the limitation of my design as the HAN framework relies on its submodules' performance. That being said, the HAN framework can adopt new submodules very easily. Hence, in future work, I plan to provide more optimized submodules to further improve the overall performance.

6.3.3 Application Results

In this section, I evaluate the autotuned HAN framework with two applications on Stampede2.

ASP

ASP [53] solves the all-pairs-shortest-path problem with a parallel implementation of the Floyd-Warshall algorithm. Processes take turns to act as the root, and broadcast a row of

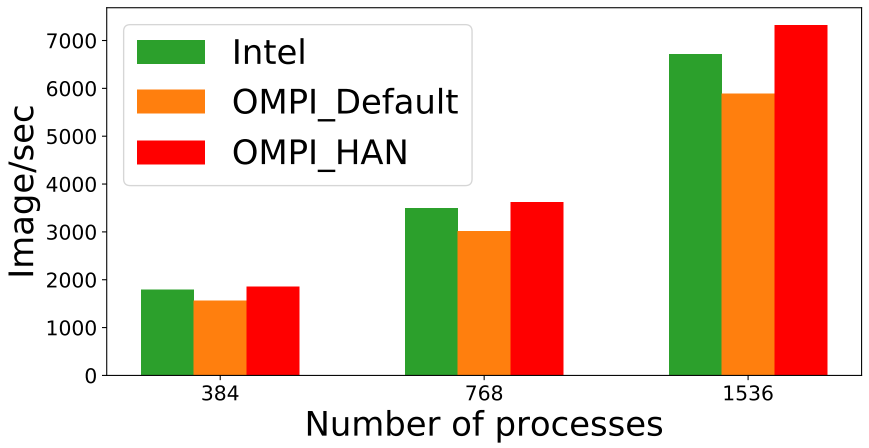


Figure 6.8: Performance of Horovod on Stampede2

the weight matrix to others, followed by computation, which causes MPI_Bcast to be the most time-consuming part of ASP.

Table 6.3 presents the time of the first 1536 iterations in ASP on 1536 processes when the matrix size is 1M. I choose the first 1536 iterations to minimize the testing time but still cover all the possible cases by making sure each process acts as the root process once. The HAN framework achieves 1.16X, 2.68X, and 4.28X speed up against Intel MPI, MVAPICH2, and the default Open MPI for the communication cost in ASP and reduces the communication percentage from 50.24% (Intel MPI), 69.29% (MVAPICH2), 81.77% (default Open MPI) to 46.41%.

Horovod

Horovod [55] is a distributed training framework that uses MPI_Allreduce to average gradients. I use tf_cnn_benchmarks [3] with synthetic datasets to train AlexNet on Stampede2, and the performance is shown in figure 6.8. Due to a configuration problem,

Table 6.3: Performance of ASP (1536 processes, 1M Matrix)

	Intel	MVAPICH2	Default OMPI	HAN
Comm (s)	10.44	24.12	38.52	8.99
Total (s)	20.78	34.81	47.11	19.37

I only manage to run Intel MPI 17.0.3, default Open MPI 4.0.0 and the HAN framework. Compared to other MPI implementations, my design gains more improvement with a higher number of processes. It eventually becomes 24.30% faster than default Open MPI, and 9.05% faster than Intel MPI on 1536 processes.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The scientific computing community's increasing computational need demands more powerful HPC systems, and the increasing scale and complexity of these HPC systems bring new challenges to the design of MPI libraries, especially with regard to implementations of MPI collective operations.

The first challenge is propagation of noise. Initially, noise only means operating system noise such as the delay caused by page faults, interrupts, and the scheduler. Later, its definition is extended to the delay from other sources like fault tolerance, in-situ analytics, and power management. The slowdown caused by the noise on one process typically is very small, but the noise can be propagated and even amplified with MPI collective operations to cause a huge slowdown. Moreover, this slowdown grows with the number of processes in the MPI collective operations and becomes a major bottleneck in large scale applications as suggested in [22, 60]. Thus, with the increasing scale of HPC systems, how to prevent the propagation of noise is a great challenge in implementations of MPI collective operations.

The second challenge is the complex hardware hierarchy on modern HPC systems. For instance, a modern HPC system could consist of thousands of computing nodes connected via high speed interconnects, such as Infiniband. Each computer node could be equipped with several multi-core CPUs or GPUs, while CPUs are connected with each other via memory bus or inter-socket networks such as Intel QPI and AMD HyperTransport, and

GPUs are connected as peripheral devices via PCI-Express or NVLink. The communications on these hardware have significant different performances, and hence, in implementations of MPI collective operations, it is crucial to find a way to minimize the slow communication, maximize the fast communication and overlap the communications on different hardware.

The third challenge is autotuning of MPI collective operations. In Open MPI, many MPI collective operation frameworks have been implemented to support different functionalities and optimize collective operations from different angles, and each framework provides a set of configurations, such as algorithms and segment sizes, to tune its performance. As suggested in [59, 19], the performance penalty of choosing the incorrect modules and configurations could be significant, and hence, autotuning is introduced to select the optimal configurations for a given collective operation automatically. The most challenging part of autotuning is to get or estimate the cost of collective operations with possible configurations, which can be done using either empirical benchmarks or cost models. However, using benchmarks could take too much time on large clusters, and the traditional cost models are not accurate on the modern heterogeneous HPC systems.

To address these three challenges, in this dissertation, I design and implement two collective operations frameworks and an autotuning component in Open MPI.

First, I theoretically analyzed the causes of noise propagation in existing MPI libraries, which are unnecessary synchronizations. To minimize synchronizations, I present "ADAPT," a collective communication framework based on the event-driven design. Through events and callbacks, the ADAPT framework is able to relax the synchronization dependencies and maintain the minimal data dependencies, which is able to provide more tolerance to system noise. Also, I extend the ADAPT framework with topology-aware trees to exploit the parallelism of heterogeneous architectures. I demonstrate experimentally that the ADAPT framework is less affected by noise as compared to other state-of-the-art MPI libraries and it outperforms most state-of-the-art MPI libraries on heterogeneous architectures, especially for large messages.

Second, to further improve the performance of hierarchical collective operations and provide a flexible design to meet the fast-changing hardware, I present "HAN," a task-based hierarchical autotuned collective communication framework in Open MPI, which divides any hierarchical collective communication into a set of tasks. The main benefits of the task-based design are:

1. it selects suitable submodules to utilize hardware features;
2. it provides more opportunities to overlap communications;
3. it minimizes the effort of updating submodules to adapt to new hardware.

I also develop an autotuning component with a novel cost model in the HAN framework, which can significantly reduce the tuning cost and provide an improved accuracy compared to previous approaches. The experiments in two large scale HPC systems demonstrate the HAN framework outperforms other state-of-the-art MPI implementations in most cases in both benchmarks and applications, providing highly efficient and portable collective communication operations.

7.2 Future Work

To further improve the performance of MPI collective operations, I am considering following two possible directions.

- The first direction is to move the execution of collective operations in NIC. The ADAPT framework I presented in chapter 3 and chapter 4 utilizes an event-driven design to relax the synchronizations in existing implementations of collective operations. The events and callbacks in the ADAPT framework are tracked by the progress engine in Open MPI, which occupies a CPU. To reduce the CPU usage in the communication, I am looking at methods to move the progress engine from CPU to NIC, which could result in lower overhead and better overlap rate.
- The second direction is to improve the submodules in the HAN framework. As suggested in section 6.3, the performance of the HAN framework is limited by its submodules' performance. Therefore, to boost the upper bound of the HAN framework, I need to improve the submodules. In the submodules used for inter-node communications, I plan to integrate the network switch level information to minimize

the inter-switch communication and improve their performance. As for intra-node communication, HPC systems are increasingly equip with accelerators, such as GPUs, to have more parallelism with lower power consumption. Hence, I plan to extend the HAN framework with specialized GPU collective operation submodules to support hierarchical collective operations on GPU clusters.

With the increasing scale of HPC systems and the fast-growing computing power, communication becomes the major bottlenecks in many HPC applications. The knowledge I gained during this study and the concepts used in this dissertation are not limited to the MPI community. Thus, the outcome of this dissertation, including theories and implementations, can be extended to optimize communications for other programming models of the HPC community. One potential target is the communication in task-based runtime systems. Task-based runtime systems adopt a data-flow programming model and divide an algorithm into computation (tasks) and communication (dependencies among tasks). Since these systems are able to hind the complex hardware hierarchy from users and provide efficient resource management, task-based runtime systems become more and more popular for developing applications on modern HPC systems. In task-based runtime systems, tasks need to send/receive data to/from other tasks on local or remote compute nodes. Some of these communications follow the same pattern as MPI collective communications, and hence, the optimization methods in the dissertation can be significantly beneficial for communication as well as the overall performance of task-based runtime systems.

Bibliography

[1] (2018). *Intel MPI Benchmarks User Guide*. https://software.intel.com/en-us/imb-user-guide. 34, 65, 86

[2] (2019a). *OSU Micro-Benchmarks*. http://mvapich.cse.ohio-state.edu/benchmarks/. 65

[3] (2019b). *TensorFlow benchmarks*. https://github.com/tensorflow/benchmarks. 93

[4] Agrawal, D. and Bhuyan, L. (1984). Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on Computers*, 33(04):323–333. 61

[5] Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. (1995). Loggp: Incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation. Technical report, Santa Barbara, CA, USA. 4

[6] Aulwes, R. T., Daniel, D. J., Desai, N. N., Graham, R. L., Risinger, L. D., Taylor, M. A., Woodall, T. S., and Sukalski, M. W. (2004). Architecture of la-mpi, a network-fault-tolerant mpi. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 15–. 12

[7] Bayatpour, M., Chakraborty, S., Subramoni, H., Lu, X., and Panda, D. K. D. (2017). Scalable reduction collectives with data partitioning-based multi-leader design. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 64:1–64:11, New York, NY, USA. ACM. 4, 14

[8] Bayatpour, M., Hashmi, J. M., Chakraborty, S., Subramoni, H., Kousha, P., and Panda, D. K. (2018). Salar: Scalable and adaptive designs for large message reduction collectives. *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 12–23. 2, 4, 14, 16, 66

[9] Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. (2006). The influence of operating systems on the performance of collective operations at extreme scale. In *2006 IEEE International Conference on Cluster Computing*, pages 1–12. 2, 13, 36

[10] Bhattacharya, S., Pratt, S., Pulavarty, B., and Morgan, J. (2003). Asynchronous i/o support in linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386. 28

[11] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. 43

[12] Chan, E., Heimlich, M., Purkayastha, A., and van de Geijn, R. (2007). Collective communication: Theory, practice, and experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(13):1749–1783. 65

[13] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. (1993). Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12. ACM. 4

[14] Cutsem, T. V., Mostinckx, S., Boix, E. G., Dedecker, J., and Meuter, W. D. (2007). Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC '07.*, pages 3–12. 26

[15] Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., and Morris, R. (2002). Event-driven programming for robust software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 186–189. ACM. 28

[16] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. (2006). Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 29–42. ACM. 26

[17] Eller, P. R., Hoefler, T., and Gropp, W. (2019). Using performance models to understand scalable krylov solver performance at scale for structured grid problems. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 138–149, New York, NY, USA. Association for Computing Machinery. 4, 16, 66

[18] Fagg, G. E. and Dongarra, J. J. (2000). Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In Dongarra, J., Kacsuk, P., and Podhorszki, N., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, Berlin, Heidelberg. Springer Berlin Heidelberg. 12

[19] Fagg, G. E., Pjesivac-grbovic, J., Bosilca, G., Dongarra, J. J., and Jeannot, E. (2006). Flexible collective communication tuning architecture applied to open mpi. In *In 2006 Euro PVM/MPI*. 15, 16, 81, 86, 96

[20] Faraj, A. and Yuan, X. (2005). Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 393–402, New York, NY, USA. ACM. 4

[21] Faraj, A., Yuan, X., and Lowenthal, D. (2006). Star-mpi: Self tuned adaptive routines for mpi collective operations. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 199–208, New York, NY, USA. ACM. 16

[22] Ferreira, K. B., Bridges, P., and Brightwell, R. (2008). Characterizing application sensitivity to os interference using kernel-level noise injection. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. 2, 13, 36, 95

[23] Ferreira, K. B., Bridges, P. G., Brightwell, R., and Pedretti, K. T. (2010). The impact of system design parameters on application noise sensitivity. In *IEEE Cluster 2010*, pages 146–155. 2

[24] Ferreira, K. B., Widener, P., Levy, S., Arnold, D., and Hoefler, T. (2014). Understanding the effects of communication and coordination on checkpointing at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 883–894. 2

[25] Firasta, N., Buxton, M., Jinbo, P., Nasri, K., and Kuo, S. (2008). Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 19(20). 92

[26] Forum, M. P. I. (2012). MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/. 1

[27] Freeh, V. W., Pan, F., Kappiah, N., Lowenthal, D. K., and Springer, R. (2005). Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 4a–4a. 2

[28] Gabriel, E., Resch, M., and Rühle, R. (1999). Implementing mpi with optimized algorithms for metacomputing. 12

[29] Goglin, B. and Moreaud, S. (2013). KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing*, 73(2):176–188. 14, 62

[30] Graham, R., Venkata, M. G., Ladd, J., Shamis, P., Rabinovitz, I., Filipov, V., and Shainer, G. (2011). Cheetah: A Framework for Scalable Hierarchical Collective Operations. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 73–83. 3, 14, 15

[31] Graham, R. L., Woodall, T. S., and Squyres, J. M. (2006). Open mpi: A flexible high performance mpi. In Wyrzykowski, R., Dongarra, J., Meyer, N., and Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg. Springer Berlin Heidelberg. 11

[32] Gropp, W., Olson, L. N., and Samfass, P. (2016). Modeling mpi communication performance on smp nodes: Is it time to retire the ping pong test. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 41–50. ACM. 4

[33] Hockney, R. W. (1994). The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Comput.*, 20(3):389–398. 4, 51

[34] Hoefler, T., Lumsdaine, A., and Rehm, W. (2007). Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM. 55

[35] Hoefler, T., Schneider, T., and Lumsdaine, A. (2010). Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. 2, 18, 19, 24, 36

[36] Kandalla, K., Subramoni, H., Vishnu, A., and Panda, D. K. (2010). Designing topology-aware collective communication algorithms for large scale InfiniBand clusters: Case studies with Scatter and Gather. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. 3, 14, 54

[37] Karonis, N. T., de Supinski, B. R., Foster, I., Gropp, W., Lusk, E., and Bresnahan, J. (2000). Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *IPDPS 2000*, pages 377–384. 14

[38] Kielmann, T., Bal, H. E., and Verstoep, K. (2000). Fast measurement of logp parameters for message passing platforms. In Rolim, J., editor, *Parallel and Distributed Processing*, pages 1176–1183, Berlin, Heidelberg. Springer Berlin Heidelberg. 4

[39] Kielmann, T., Hofman, R. F. H., Bal, H. E., Plaat, A., and Bhoedjang, R. A. F. (1999). MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. PPoPP '99, pages 131–140. 14

[40] Kim, J., Dally, W. J., Scott, S., and Abts, D. (2008). Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. 61

[41] Leiserson, C. E. (1985). Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901. 61

[42] Levy, S., Topp, B., Ferreira, K. B., Arnold, D., Hoefler, T., and Widener, P. (2014). *Using Simulation to Evaluate the Performance of Resilience Strategies at Scale*, pages 91–114. Springer International Publishing, Cham. 2

[43] Li, H. and Maresca, M. (1989). Polymorphic-torus network. *IEEE Transactions on Computers*, 38(9):1345–1351. 61

[44] Ma, T., Bosilca, G., Bouteiller, A., and Dongarra, J. (2012a). HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 970–982. 3

[45] Ma, T., Bosilca, G., Bouteiller, A., and Dongarra, J. (2012b). HierKNEM: An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 970–982. 14

[46] Mondragon, O. H., Bridges, P. G., Levy, S., Ferreira, K. B., and Widener, P. (2016). Scheduling in-situ analytics in next-generation applications. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 102–105. 2

[47] NVIDIA (2016). NCCL. `https://github.com/NVIDIA/nccl`. 54

[48] Pai, V. S., Druschel, P., and Zwaenepoel, W. (1999). Flash: An efficient and portable web server. 26

[49] Parsons, B. S. and Pai, V. S. (2015). Exploiting process imbalance to improve mpi collective operations in hierarchical systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 57–66, New York, NY, USA. ACM. 14

[50] Pješivac-Grbović, J., Angskun, T., Bosilca, G., Fagg, G. E., Gabriel, E., and Dongarra, J. J. (2007a). Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143. 2, 4, 16, 51, 65, 85

[51] Pješivac-Grbović, J., Bosilca, G., Fagg, G. E., Angskun, T., and Dongarra, J. J. (2007b). Decision trees and mpi collective algorithm selection problem. In Kermarrec, A.-M., Bougé, L., and Priol, T., editors, *Euro-Par 2007 Parallel Processing*, pages 107–117, Berlin, Heidelberg. Springer Berlin Heidelberg. 83

[52] Pješivac-Grbović, J., Bosilca, G., Fagg, G. E., Angskun, T., and Dongarra, J. J. (2007). Mpi collective algorithm selection and quadtree encoding. *Parallel Comput.*, 33(9):613–623. 83

[53] Plaat, A., Bal, H. E., and Hofman, R. F. H. (2001). Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Gener. Comput. Syst.*, 17(6):769–782. 92

[54] Sanders, P., Speck, J., and Träff, J. L. (2009). Two-tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Comput.*, 35(12):581–594. 33, 54

[55] Sergeev, A. and Balso, M. D. (2018). Horovod: fast and easy distributed deep learning in tensorflow. 93

[56] Snell, Q. O., Mikler, A. R., and Gustafson, J. L. (1996). Netpipe: A network protocol independent performance evaluator. In *in IASTED International Conference on Intelligent Information Management and Systems*. 60, 87

[57] Squyres, J. M. and Lumsdaine, A. (2003). A component architecture for lam/mpi. In Dongarra, J., Laforenza, D., and Orlando, S., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 379–387, Berlin, Heidelberg. Springer Berlin Heidelberg. 12

[58] Subramoni, H., Potluri, S., Kandalla, K., Barth, B., Vienne, J., Keasler, J., Tomko, K., Schulz, K., Moody, A., and Panda, D. K. (2012). Design of a Scalable InfiniBand Topology Service to Enable Network-topology-aware Placement of Processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 70:1–70:12. 14, 54

[59] Vadhiyar, S. S., Fagg, G. E., and Dongarra, J. (2000). Automatically tuned collective communications. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 3–3. 4, 15, 81, 85, 96

[60] Vishnoi, N. K. (2007). *The Impact of Noise on the Scaling of Collectives: The Nearest Neighbor Model*, pages 476–487. Springer. 13, 95

[61] Wang, H., Potluri, S., Luo, M., Singh, A. K., Sur, S., and Panda, D. K. (2011). MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science - Research and Development*, 26(3):257. 54

[62] Wang, L., Wu, W., Xu, Z., Xiao, J., and Yang, Y. (2016). Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. 3

[63] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T. (2018). Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 41–53, New York, NY, USA. ACM. 3

[64] Widener, P. M., Levy, S., Ferreira, K. B., and Hoefler, T. (2016). On Noise and the Performance Benefit of Nonblocking Collectives. *Int. J. High Perform. Comput. Appl.*, 30(1):121–133. 13

[65] Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., and Dongarra, J. (2015). Hierarchical dag scheduling for hybrid distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165. 3

[66] Yoshii, K., Iskra, K., Naik, H., Beckmanm, P., and Broekema, P. C. (2009). Characterizing the Performance of Big Memory on Blue Gene Linux. In *2009 ICPP Workshops*, pages 65–72. 13

[67] Zheng, F., Yu, H., Hantas, C., Wolf, M., Eisenhauer, G., Schwan, K., Abbasi, H., and Klasky, S. (2013). Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12. 13

[68] Zhu, H., Goodell, D., Gropp, W., and Thakur, R. (2009). *Hierarchical Collectives in MPICH2*, pages 325–326. Springer Berlin Heidelberg, Berlin, Heidelberg. 14

Vita

Xi Luo was born in Nanjing, Jiangsu, China, in December 22, 1988. After completing his education at Jinling High School in 2007, he entered Sichuan University and received his Bachelor's degree in Computer Science in 2011. In 2014, after finishing his Master's degree in Computer Science at Stevens Institute of Technology, he was enrolled in the Ph.D. program in Computer Science at the University of Tennessee, Knoxville. During his studies, he worked as a graduate research assistant at the Innovative Computing Laboratory (ICL) under the supervision of Dr. Jack Dongarra and Dr. George Bosilca. His research interests are focused on high-performance computing, with a concentration on MPI collective operations. While pursuing his doctoral degree, Xi completed an internship at Oak Ridge National Lab in the summer of 2017 under the guidance of Dr. Dali Wang. Xi Luo is expected to receive his Doctor of Philosophy degree in Computer Science in May 2020.