12-2007

Towards Automatic and Adaptive Optimizations of MPI Collective Operations

Jelena Pjesivac-Grbovic
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Jelena Pjesivac-Grbovic entitled "Towards Automatic and Adaptive Optimizations of MPI Collective 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 J. Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

George Bosilca, Itamar Elhanany, James Plank

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

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

To the Graduate Council:

I am submitting herewith a dissertation written by Jelena Pješivac-Grbović entitled "Towards automatic and adaptive optimizations of MPI collective 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 J. Dongarra

Major Professor

We have read this dissertation
and recommend its acceptance:

Dr. George Bosilca

Dr. Itamar Elhanany

Dr. James Plank

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and
Dean of the Graduate School

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

Towards
Automatic and Adaptive Optimizations of MPI Collective Operations

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Jelena Pješivac-Grbović

December 2007

Dedication

This dissertation is dedicated to my parents, Divna and Dragan Pješivac, who have always been there for me.

Acknowledgments

Writing this section took longer than I anticipated. Not for the lack of gratitude, but for the lack of proper words to express the gratitude I feel for being given the opportunity to complete the greatest academic challenge I have ever faced.

First, I would like to thank my dissertation advisor, Dr. Jack Dongarra. I remember the first meeting we had back in January 2003 in the Claxton complex, where we discussed MPI and collective operations, specifically how one can do broadcast using a binary tree to achieve better performance than sending the message directly. It sounds funny now, but I was quite amazed with the idea. I went on to use MPI to parallelize the multi-scale avascular tumor model code for my senior project at Ramapo College of New Jersey in Spring 2003. I think that since that January day, I knew that I would dedicate at least some time to collective operations, just because they are "so cool." Throughout my studies at UT, I was constantly impressed with Jack's ability to present, propose, analyze, respond to requests, and manage and befriend people. Being able to attend the Supercomputing Conference, I could see first hand, that I was rather lucky for having him as my advisor. Regardless of where my professional career takes me, I hope that I will have an opportunity to interact with Jack again.

Second, I would like to thank my project leaders, Dr. Graham Fagg, and Dr. George Bosilca for mentoring me on a day to day basis. I am especially thankful to Graham for having patience with me my first semester at UT when I spent the whole semester optimizing the barrier collective using four algorithms. I guess there is a learning curve in this process and at some point, things "click," but from my current perspective, I might not have had the patience necessary to manage a student as clueless in this particular field as I originally was. Additionally, I would like to thank Graham for sharing his ideas with me and persuading me to try the quadtree encoding approach for the problem at hand. George was a fantastic mentor, colleague, boss, and friend. I enjoyed working with and learning from him. From the disucssions in our Claxton conference room about models and algorithms, to attending conferences and meetings, to advice regarding family and children, George was the one person to whom I could easily tell my doubts and fears about the program, work, and career, as well as accept the criticism from, knowing that it is only for my benefit.

Third, I would like to thank my committee members, Dr. James Plank and Dr. Itamar Elhanany for valuable suggestions and for taking the time to participate in this process. Dr. Plank's lectures helped me determine the direction in which I would like to take my career.

The work in this dissertation would be much harder to complete without the funding from the following grants: NSF Grant # ANI-0222945, Rice University Subcontract #

R7B127 on Prime Contract # 12783-001-05 49, and DoE Contracts # DE-FC02-01ER41203 and # DE-FG02-06ER25728.

The Innovative Computing Laboratory and its people provided an incredible environment for me to learn and grow. Our administrative staff members, Teresa, Tracy, Leighanne, Tracy, Scott, Jan, and David, made my life at UT simpler and enjoyable. Scott Wells helped enormously with most of my publications and this dissertation by proofreading the material, sometimes even on the day of the deadline! Current and former ICLers, Aurelien, Edgar, Stan, Felix, Alfredo, Jeff, Erik, Don, Asim, Terry, David, Brett, and many others were great officemates, excellent company at conferences and meetings, and impressive experts on high-performance computing, cell phones, and much more. The four years at ICL were both the most challenging and the most rewarding years of my life so far. Thank you all!

In addition, I would like to mention my fellow graduate students and friends who influenced my UT experience. I want to thank Thara Angskun for his wonderful collaboration, for being an excellent officemate, and, for being a never ending source of information about debugging, automake, character encodings, imaging, etc. Also, I would like to thank Erika Fuentes, Fengguang Song, and Haihang You for useful discussions regarding my dissertation in my last year at UT. Finally, I would like to thank Sam Reynolds and Michael Camfield for organizing the study group for our qualifying exam in Fall 2006. It was a pleasure working and studying with you all.

My life would not be complete without friends both in Tennessee and across the globe. I would like to thank Zoran Dimitrijević, Zorana Dicić, Jasmina Josić, Jovan and Winkie Ilić, Lana and Zoran Živanović, Megan Ayer, Paul Deiana-Molnar, Pedja Klašnja, Carrie Morris, and Jeff Larkin for being there for me at times of joy and times of stress. I am looking forward to seeing you all in new and exciting places. A special thanks goes to Zoran, Jovan, and Lana for giving me important and useful advice regarding my education, dissertation, and career choices.

Finally, I would like to thank my family for providing an enormous support network throughout my life and especially in the last couple of months. My parents, Divna and Dragan Pješivac, were always there for me, guiding me in the right direction and supporting my decisions. Together with my mother-in-law, Nada Filipović, they were instrumental in me finishing the dissertation on time by providing love and care for my son, Milan. My younger brothers, Raško and Vlado Pješivac, were a pain in the beginning, but they grew into wonderful adults who make me proud for being their older sister. The largest thank you goes to my husband, Dragoslav Grbović, who provided me with unconditional love and understanding, and my son, Milan Grbović, whose smile has the power to chase away even the darkest clouds and whose good sleeping habits made this dissertation possible.

Lastly, I would like to thank the Holy Father for presenting me with this challenge and giving me the strength and support to meet it.

Jelena Pješivac-Grbović
September, 2007, Knoxville, TN

Abstract

Message passing is one of the most commonly used paradigms of parallel programming. Message Passing Interface, MPI, is a standard used in scientific and high-performance computing. Collective operations are a subset of MPI standard that deals with processes synchronization, data exchange and computation among a group of processes. The collective operations are commonly used and can be application performance bottleneck. The performance of collective operations depends on many factors, some of which are the input parameters (e.g., communicator and message size); system characteristics (e.g., interconnect type); the application computation and communication pattern; and internal algorithm parameters (e.g., internal segment size). We refer to an algorithm and its internal parameters as a method.

The goal of this dissertation is a performance improvement of MPI collective operations and applications that use them. In our framework, during a collective call, a system-specific decision function is invoked to select the most appropriate method for the particular collective instance. This dissertation focuses on automatic techniques for system-specific decision function generation. Our approach takes the following steps: first, we collect method performance information on the system of interest; second, we analyze this information using parallel communication models, graphical encoding methods, and decision trees; third, based on the previous step, we automatically generate the system-specific decision function to be used at run-time. In situation when a detailed performance measurement is not feasible, method performance models can be used to supplement the measured method performance information.

We build and evaluate parallel communication models of 35 different collective algorithms. These models are built on top of the three commonly used point-to-point communication models, Hockney, LogGP, and PLogP. We use the method performance information on a system to build quadtrees and C4.5 decision trees of variable sizes and accuracies. The collective method selection functions are then generated automatically from these trees. Our experiments show that quadtrees of three or four levels are often enough to approximate experimentally optimal decision with a small mean performance penalty (less than 10%). The C4.5 decision trees are even more accurate (with mean performance penalty of less than 5%). The size and accuracy of C4.5 decision trees can be further improved with use of appropriate composite attributes (such as "total message size", or "even communicator size".) Finally, we apply these techniques to tune the collective operations on the Grig cluster at the University of Tennessee and to improve an application performance on the Cray XT4 system at Oak Ridge National Laboratory. The tuned collective is able to achieve more than 40% mean performance improvement over the native broadcast implementation. Using the platform-specific reduce on Cray XT4 lead to 10% improvement in the overall application performance. Our results show that the methods we explored are both applicable and effective for the system-specific optimizations of collective operations and are a right step toward automatically tunable, adaptive, MPI collectives.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Technology advances in recent years have pushed the limits of single processor systems. Today, even laptops come with multicore CPUs and are equipped with multiple network interfaces. Parallel computing is becoming less of an exotic approach used by domain scientists to solve the fluid dynamics and supernova explosion types of problems, it is also becoming mainstream in multiple areas. On a small scale, many newly produced machines are multicore, and efficient ways of programming and harnessing this new computing power are needed. At the same time, due to cost drops, many smaller institutions (groups in academic environment, small- to intermediate- size companies) are able to afford clusters to support their research, web services, data mining efforts, etc. In this setting, clusters are often used to provide the high-availability of services as well. On a large scale, domain scientists are still trying to solve fluid dynamics, structured mechanics, signal processing, climate modeling, manufacturing, finance, and similar problems. New to the mix are computational biology, visualization, and nanotechnology. But the common thread in all these type of problems is that they are too large to be solved using a single or even couple of CPUs. Finally, geographically distributed computing (e.g., grids, SETI@home) can be a feasible option for the loosely coupled problems or really large data sets.

High-performance computing is an area of computing focused on large-scale systems and applications. The TOP500 project was started in 1993 to provide a reliable basis for tracking and detecting trends in high-performance computing [Top 500, 2007]. The growth of this field in the last 15 years has been remarkable. As of June 2007, the minimum computational power to enter the TOP 500 list is around 4 TFlops, and the theoretical aggregate computing power is at 4.92 PFlops [Top 500, 2007]. Compared to the June 2006 statistics when the entry bar was at 2.026 TFlops and aggregate computational power was at around 2.79 PFlops, we see that the aggregate computational power increased more than 70%. In addition, 63% of systems on the TOP 500 list from June 2007 have more than 1024 processors, compared to 35.6% year earlier. Going back even further, in June 2002, when the Earth Simulator [Earth Simulator, 2002] was introduced in Japan, only 23 systems on the list were achieving more than 1 TFlop. On the average, the bottom 200+ systems on the list are not present on the next edition, six months later. The systems on the TOP 500 list are distributed across different application areas from defense and finance to medicine and pharmaceutics to gaming and research with many more areas in between. High-performance computing has become reality in many fields.

Developing software for these parallel environments (even at small scale) is a challenging task. Some of the challenges include the programmer's view of system memory, the level of parallelism that will be employed (instruction-, thread-, or process-level parallelism), debugging and tracing at scale, I/O management, etc. On large scale systems, these issues are even more pronounced.

Two frequently used parallel programming models differ on the programmer's view of system memory: shared vs. distributed system memory. In the shared memory model, every process is able to access remote data seamlessly (although, if the accessed memory location belongs to a remote process, a performance penalty can occur). The shared memory model is amenable for fine-grain parallelism. Multithreading and directive-based approaches (such as OpenMP) are often used when this capability is present. In the distributed memory model, explicit message passing is the most commonly used programming approach. The message passing is well suited for coarse-grain parallelism. In this approach, each of the processes has local memory, and the access to the remote data is provided via explicit messages in which both the sender and receiver process need to be involved.

Message Passing Interface, *MPI* [Snir et al., 1998], is a standard used in scientific and high-performance computing. MPI is governed by the *Message Passing Forum*, a standardization body consisting of academia, government, and industry members. MPI was designed to replace most of the vendor-specific, message passing libraries that proliferated in late 80s and early 90s. The goal was to help library and application developers to create portable and high-performance code more easily, while allowing system vendors to utilize their specialized hardware features.

Operations used to exchange the information among a group of processes are an important subset of the MPI standard. These operations are referred to as collective operations or collectives, and some common examples are Barrier, Broadcast, Reduce, and Alltoall.

The profiling studies of production codes showed that the collective operations are commonly used and that they could be an application performance bottleneck [Rabenseifner, 1999]. For these reasons, in the last decade, the collective operations have been an active area of research, resulting in a large number of different algorithms with radically different performance characteristics. Often, these algorithms are tuned for a particular usage case, e.g., large message sizes, or a specialized physical network topology. In addition, some of the algorithms utilize internal explicit message segmentation to allow overlap between different communications or between computation and communication. In such case, the performance of an algorithm depends both on system properties and the selected segment size. We refer to the algorithm and its internal parameters, such as segment size, as a *method*. In order to achieve close-to-optimal performance, even on a single system, one must resort to using multiple methods. The existence of a single method that achieves an optimal performance under all usage case scenarios is highly improbable.

The goal of this dissertation is a performance improvement of the MPI collective operations and applications which use them on individual systems. To achieve this we focus on the run-time collective method selection process. We use performance models, graphical encoding, and statistical learning techniques to automatically build adaptable, efficient, and fast run-time decision functions.

1.1 Contributions

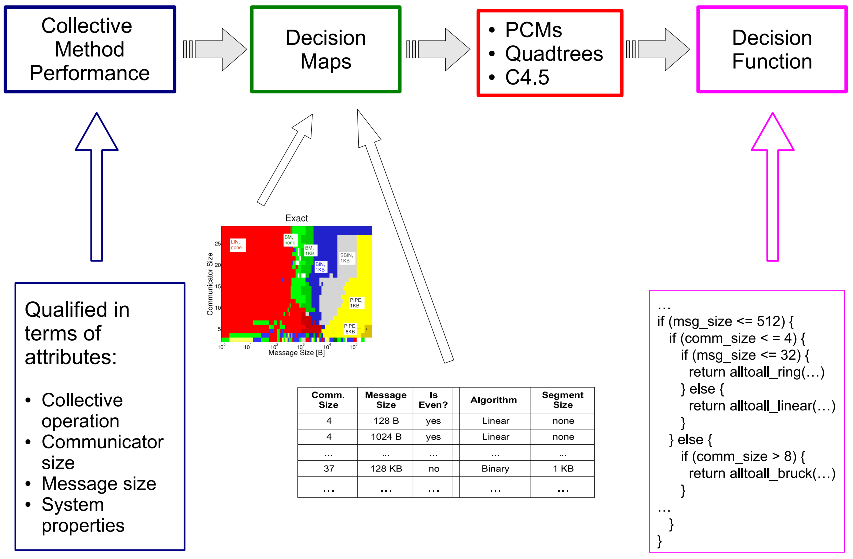
This dissertation extends the current state of the art in collective communication performance tuning via the following: by providing the performance models of many of the MPI collective operation algorithms in terms of multiple point-to-point communication models; introducing new collective communication algorithms and new variants of existing collective communication algorithms; exploring different methods for automatic analysis of the method performance information, and finally, providing the experience-based expert advise for the platform-specific collective operation performance tuning process.

Performance Models of MPI Collective Algorithms: Modeling performance of collective algorithms has been addressed by a number of authors in the past. However, unlike most of them, we develop and verify algorithm performance models in terms of multiple point-to-point communication, such as Hockney, LogP/LogGP, and PLogP. The dissertation contains 105 performance models for 35 different collective algorithms. We study analytical properties of these models to determine optimal segment size of segmented algorithms. We also extend the existing analysis of parallel communication models (*PCMs*) to determine the applicability of models and to determine when they fail to capture experimentally observed behavior.

New Algorithms for MPI Collective Operations: We introduce a number of new or new variations of existing algorithms. We propose a **split-binary** algorithm for broadcast operation and the **pipelined/segmented** and **synchronized** versions of many of the algorithms currently in use. We show that in some cases, the segmented algorithm can give an excellent performance improvement over the basic algorithm and we explain the methods for finding the near-optimal segment size. Synchronized versions of algorithms show performance benefits for regular applications for rooted collectives in large-scale systems where the probability of overloading root process with a large number of messages is high.

Platform-specific Performance Tuning: The main contribution of this work is the development and analysis of different methods for collective communication performance tuning on individual systems. Given the collective algorithm performance information, we employ **parallel communication models**, **graphical encoding techniques**, and **statistical learning methods** to construct an MPI collective method selection function (*decision function*). Similar approaches have been tried in high-performance computing fields in optimizations of matrix multiplication operations and selection of non-linear solvers. However, the constraints of our problem are different; the time to make a decision is critical and needs to be minimized. Moreover, even on dedicated systems with very low levels of system noise, the reproducibility of MPI collective operation performance results can be a challenge. Thus, the methods we develop must produce efficient and resilient decision functions.

Finally, we evaluate our approach on a number of different architectures and interconnects. Experimentally, we covered a large number of different systems: from FastEthernet, GigE, MX, and Infiniband to Cray specific Portals. We also run our tests on systems of



Comm. Size	Message Size	Is Even?	Algorithm	Segment Size
4	128 B	yes	Linear	none
4	1024 B	yes	Linear	none
...	...	...	...	...
37	128 KB	no	Binary	1 KB
…	…	…	…	…

```
…
if (msg_size <= 512) {
   if (comm_size < = 4) {
      if (msg_size <= 32) {
        return alltoall_ring(…)
      } else {
        return alltoall_linear(…)
      }
   } else {
      if (comm_size > 8) {
        return alltoall_bruck(…)
      }
…
   }
}
```

Figure 1.1: Platform-specific tuning process.

different scales: from a 32 node Boba cluster, via 64 node Grig and Frodo, to 4096 node Thunderbird and 11,000 process Jaguar.*

Figure 1.1 depicts the general steps in the adaptable, platform-specific tuning process employed in this dissertation. Starting from the collective method performance information on the particular system, we build experimentally optimal decision maps, which can be represented either visually or using tables. This information is used to construct different quadtrees and C4.5 decision trees, which then can be used, together with parallel communication models, to generate decision functions for the particular collective on that system.

1.2 Document Organization

The remainder of this dissertation presents the details about our research. Chapter 2 provides relevant information about the MPI standard and collective operations as well as available algorithms. Chapter 3 presents a survey of the literature for each contribution. Chapter 4 discusses and evaluates parallel communication models developed as a part of this dissertation. Chapter 5 provides a formal thesis statement and contains the main contributions of this work: methods for automatic algorithm selection function generation. The results from this dissertation research are in Chapter 6. Finally, we conclude with Chapter 7.

*These sistems are described in detail in Chapters 4 and 6.

Chapter 2

Message Passing Interface

2.1 MPI Standard

The first version of the standard, MPI-1, was delivered in 1994 as a joint effort of academia, national laboratories, and industry. The goal of the standard was to select a set of features commonly used by message passing applications, provide a standard interface (*API*) for them, and at the same time, allow enough flexibility for library implementers and vendors to tune their implementations and utilize the special hardware features available on their machines. The MPI-1 standard defined [MPI Forum, 1995]:

- Point-to-point communication
- (Intercommunicator) Collective operations
- Process groups
- Communication contexts (communicators)
- Process topologies
- Bindings for FORTRAN 77 and C
- Environmental Management and inquiry
- Profiling interface

The collective operations in the MPI-1 standard are restricted to processes in a single communicator and are thus referred to as intercommunicator collective operations. The MPI-1 standard decided not to address the following sets of operations [MPI Forum, 1995]:

- One-sided / Explicit shared-memory operations
- Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages
- Program construction tools
- Debugging facilities

- Explicit support for threads
- Support for task management
- I/O functions

The MPI-1 standard was replaced by MPI-1.2 which clarified some of the ambiguities and included errata.

In 1997 the MPI-2 standard was proposed. MPI-2 expanded MPI-1.2 functionality to include [MPI Forum, 1995]:

- One-sided / Explicit shared-memory operations
- I/O functions
- Dynamic process management
- Explicit support for threads
- Bindings for C++

Additionally, MPI-2 introduced intracommunicator collective operations. The intracommunicator collective operations take place over processes in two communicators. The syntax of these collective operations does not differ from intercommunicator ones, but their semantics are quite different. In this study, we will not consider intracommunicator collectives.

2.2 MPI Collective Operations

Collective operations are used for synchronization and data exchange among a group of processes. Most of the intercommunicator collective operations were defined by the MPI-1 standard:

Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

Synchronization routine. Blocks the caller process until all processes in the specified communicator have reached this synchronization point.

Broadcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype ddt, int root,
            MPI_Comm comm)
```

Broadcasts a message from a root process to all processes in the communicator.

Scatter(v)

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
                 MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 int root, MPI_Comm comm)
```

Scatters the buffer specified at root process among all processes in the communicator in order: i^{th} block of data is sent to i^{th} process in the communicator. The vector version of this function allows individual block sizes to be different and block locations to be out of order. This is an inverse operation of MPI_Gather(v).

Gather(v)

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int *recvcounts, int *displs,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Gathers specified data ordered by the process rank to the root process. Each of the ranks in the communicator sends its buffer to the root process (including the root itself) which stores received data in order by the rank of the sending process. Vector version of this collective allows for variable size of send buffers and out-of-order placement of incoming buffers.

Allgather(v)

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                  void* recvbuf, int recvcount, MPI_Datatype recvtype,
                  MPI_Comm comm)
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                   void* recvbuf, int *recvcounts, int *displs,
                   MPI_Datatype recvtype, MPI_Comm comm)
```

Same as Gather(v), except the result of the operation is available on all processes in the communicator. If P is the number of nodes in the communicator, this function is equivalent to P calls to MPI_Gather(v) operation with different root each time. Alternatively, it is equivalent to a MPI_Gather(v) operation followed by MPI_Bcast operation with a properly defined datatype.

Alltoall(v)

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                 void* recvbuf, int recvcount, MPI_Datatype recvtype,
                 MPI_Comm comm)
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls,
                  MPI_Datatype sendtype,
                  void* recvbuf, int *recvcounts, int *rdispls,
                  MPI_Datatype recvtype, MPI_Comm comm)
```

The alltoall(v) function is used to implement total exchange among the processes in the communicator. Each process sends an individual message to all other processes and receives an individual message from all of them at the same time. In Alltoall, a process r sends block j to a process j and the process j receives that block as the block r. Alltoallv allows for non-uniform data block sizes and out-of-order placement of received blocks.

Reduce

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```

The reduce operation is used to combine the elements in the send buffer of all processes in the communicator using a specified operation. The result of the operation is stored in recvbuf at the root process. MPI defines predefined operations, such as MPI_SUM, MPI_MIN, MPI_LAND, etc., which work with predefined datatypes. The user has the freedom to define operations that work with user defined datatypes as well. All MPI operations must be associative. Additionally, the predefined operations are considered commutative. The standard defines the order of execution as local_buffer = received_data *op* local_buffer .

Reduce_scatter

```
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts,
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Equivalent to a Reduce operation on data buffer of size $\Sigma_i recvcounts(i)$ followed by Scatterv operation on all processes in the communicator. Receive counts for Scatterv operation are specified in *recvcounts* array. Reduce_scatter is a reduce operation whose result is scattered among all processes in the communicator. The individual data block sizes do not need to be uniform. Equivalent to a Reduce followed by a Scatterv operation.

Allreduce

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

Reduce operation whose result is available on all processes in the communicator. Equivalent to a Reduce operation followed by a Broadcast.

Scan

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm )
```

Scan performs prefix reduction across the processes in the communicator. After the operation, the receive buffer at process k holds reduction of the values in send buffers of processes with ranks $0...k$.

Additionally, MPI-2 standard defined:

Exscan

```
int MPI_Exscan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, MPI_Comm comm)
```

Exclusive scan operation is very similar to scan: after the operation, the receive buffer at process k holds reduction of the values in send buffers of processes with ranks $0...k-1$.

Alltoallw

```
int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
                  MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
                  int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)
```

Generalized Alltoall function; each of the processes sends individual message to each of the other processes in the communicator. The message sizes can vary from process to process and, in addition, the datatype of the message can be different.

Figure 2.1 illustrates the functionality of non-vector versions of MPI collective operations. Vector versions of operations perform the same type of operation except the individual block sizes can vary.

Algorithms for MPI collective operations are discussed in detail in Section 4.1.

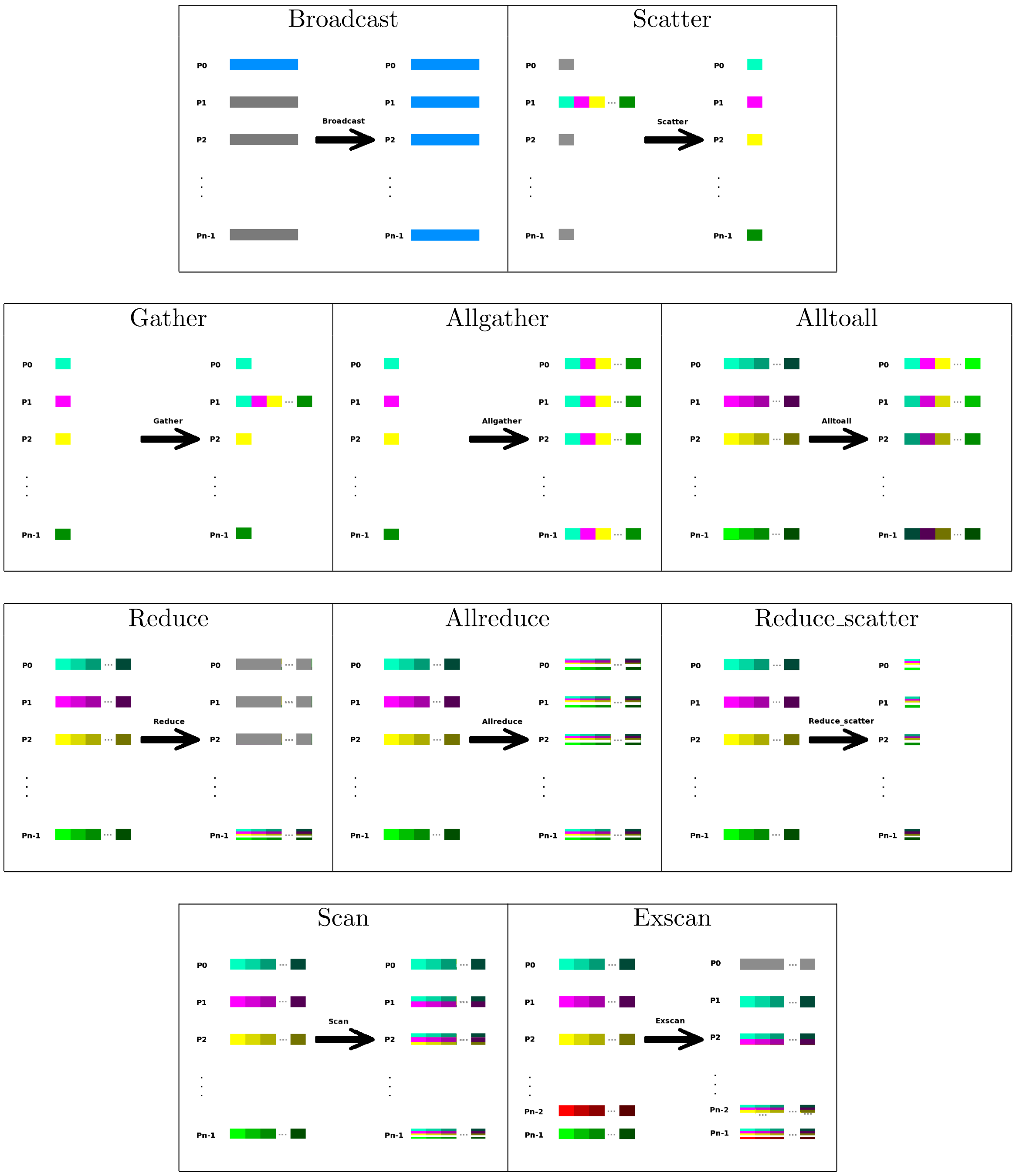


Figure 2.1: MPI collective operations (non-vector versions)

Chapter 3

Literature Review

The related work falls into the following major categories: MPI implementations and their handling of intercommunicator collectives; models of collective operation performance; and algorithm selection and automatic tuning problem. In this chapter, we review some of the most notable related efforts in these three fields.

3.1 MPI implementations and collective operations

The MPI collective algorithm selection problem has been addressed in many MPI implementations. Most of the current implementations select an appropriate method at run-time based on input parameters such as communicator and message sizes, operation type or root if applicable.

3.1.1 MPICH

MPICH [MPICH, 1994] and MPICH2 [MPICH2, 2002] are implementations of MPI-1 and MPI-2 standards, respectively, developed at Argonne National Laboratory. These software packages are some of the most frequently used MPI implementations. A number of both open source and vendor projects use MPICH and MPICH2 as their basis. For example, MPICH-V [Bouteiller et al., 2006, MPICH-V, 2007] (a fault tolerant MPI implementation), MPICH-GM [MPICH-GM, 2007] and MPICH-MX [MPICH-MX, 2007], MPI implementations on top of Myricom GM and MX interconnects are based on MPICH. Similarly, MVAPICH2 [MVAPICH, 2007], a high-performance MPI implementation for Infiniband, IBM's MPI for BlueGene/L, and Microsoft MPI, are based on MPICH2. The MPICH2 was developed to replace MPICH, and most of the current development happens in the MPICH2 project. However, MPICH is maintained as well and it features the same collective communication module as MPICH2.

In MPICH2, every collective operation is implemented using one or more algorithms. The most important collective operations (allreduce, reduce, allgather, all-to-all, and broadcast [Rabenseifner, 1999]) have the most available implementations. These algorithms do not support explicit, internal message segmentation. The following algorithms are available for these collectives (See Section 4.1 for description of these algorithms) [Thakur et al., 2005]:

- allreduce: recursive doubling, Rabenseifner's algorithm, and reduce + broadcast.
- reduce: binomial tree and Rabenseifner's algorithm.
- allgather: bruck, recursive doubling, and ring algorithms.
- alltoall: bruck, linear, and two versions of pairwise exchange (for power-of-two and non-power-of-two process case).
- broadcast: binomial tree, scatter + allgather (implemented in number of ways).
- barrier: bruck (dissemination algorithm).

The algorithm selection is based primarily on the message size used in the collective. For small message sizes, the latency of the operation is minimized, while for large message sizes, the bandwidth utilization is the determining factor. Communicator size, and whether or not the number of processes is an exact power of two, can affect the algorithm selection as well. Fixed decision functions are predetermined by implementers based on SKaMPI benchmark results on two systems: Linux cluster with MX interconnect and Cray T3E system [Thakur et al., 2005]. The value of a "small" and "large" message size is system dependent; so while the decision is "fixed" (e.g., `if (message_size less_than the_small_message_size) use algorithm_1`) the actual switching point can differ from system to system.

3.1.2 FT-MPI

Fault Tolerant MPI (FT-MPI) [FT-MPI, 2003] is an implementation of the 1.2 MPI standard, developed at the University of Tennessee, Knoxville as a part of the Harness project [Harness, 1999]. FT-MPI provides a process-level fault tolerance at the MPI API level. FT-MPI does not provide transparent fault tolerance for the user, who is responsible for recovering the application [Fagg et al., 2004]. In the case of failure, FT-MPI can abort the job (non-FT behavior), respawn the dead process, shrink/resize the application to remove the missing processes, or leave the application as is, and create holes in the `MPI_COMM_WORLD` communicator. If a failure happens during a communication call, FT-MPI ensures that the in-flight messages will be either canceled or received. Similarly, collective operations can be declared atomic or non-atomic as far as the fault tolerance is concerned.

Currently, only the linear algorithms for collectives are designed to support the fault tolerance requirements of FT-MPI. Thus, all collective communication optimizations are applicable to the non-fault-tolerant case. The following collectives have more than the basic implementation available in FT-MPI in non-fault-tolerant mode:

- barrier: fan-in-fan-out, recursive doubling, bruck, double ring.
- bcast: linear, binomial tree with segmentation, generalized tree with fixed number of children with segmentation, k-chain algorithm with segmentation, split-binary with segmentation.
- scatter: linear, binomial tree.
- allgather: linear gather + broadcast (with different combinations of topologies, no segmentation).

- allgatherv: linear gatherv + broadcast (with different combinations of topologies, no segmentation).
- reduce: linear, binomial tree with segmentation, generalized tree with segmentation, k-chain algorithm with segmentation.
- allreduce: reduce + broadcast (with different combinations of topologies and segment sizes).
- alltoall: bruck, linear.
- reduce_scatter: reduce (with different topologies) + linear scatterv.

Section 4.1 provides descriptions of the algorithms listed above.

In FT-MPI, a decision function is called to select a method at run-time. The decision function is generated manually using a visual inspection method augmented with Matlab scripts used for analysis of the experimentally collected performance data. This approach results in precise, albeit complex, decision functions. The latest version of FT-MPI contains decision functions tuned for FastEthernet network. Thus, the performance of FT-MPI collectives can be suboptimal on fast interconnects, i.e., MX, Infiniband, or even GigE.

The FT-MPI project is no longer actively maintained, and most of the development efforts is now focused on the Open MPI project.

3.1.3 Open MPI

Open MPI [Open MPI, 2005] is an open source, peer reviewed, high-performance, production-quality MPI implementation. The Open MPI project started as a collaborative effort of developers from four different MPI implementations: FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI [FT-MPI, 2003,LA-MPI, 2002,LAM/MPI, 2002,PACX-MPI, 2003]. Currently, the Open MPI project has 18 members and contributors from academia, government institutions (national laboratories), and industry.

Open MPI implementation is composed of the following parts: the MPI layer (OMPI), the run-time environment (ORTE), and the portability layer (OPAL). The OMPI layer implements MPI semantics, while the ORTE provides a resource manager, global data store, messaging layer, and a peer discovery system for parallel job start up. OPAL provides a number of useful functions and data structures implemented in a portable manner, such as high-resolution timers, fast atomic memory operations, and portable I/O functionality [Gabriel et al., 2004].

All three major parts in Open MPI (i.e., OMPI, ORTE, and OPAL) are implemented using a modular component architecture (MCA) (Figure 3.1). MCA allows Open MPI to be easily configured at run-time to utilize a subset of available components. For example, for a particular program execution, a user may select only Myricom's MX interconnect on their cluster via command line parameters. At the later time, the user can select GigE or even both interconnects, without any need for recompilation of the source code or the Open MPI library.

The collective framework in Open MPI implements collective communication using a number of different components. The following components are currently in use and/or being developed:

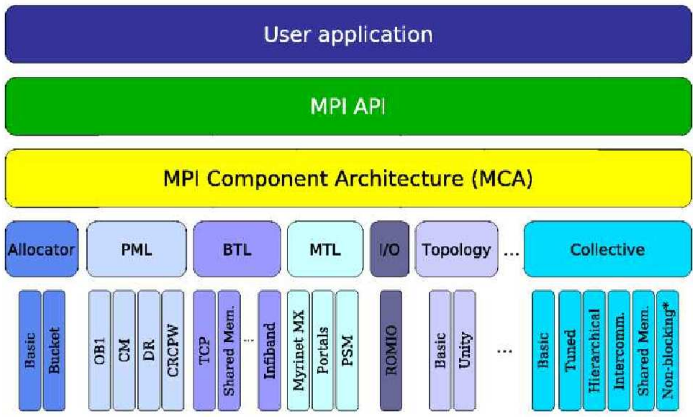


Figure 3.1: Modular Component Architecture in Open MPI.

- **basic**, which provides basic implementation for for all inter- and intra-communicator collectives. Most of the collectives are implemented using a single (often linear) algorithm, but some utilize both linear and logarithmic (binomial) algorithms without segmentation. This component was carried over from LAM/MPI.
- **self**, which provides a special implementation of intercommunicator collectives for a single process and `MPI_COMM_SELF` communicator.
- **tuned**, which provides multiple algorithms for most intercommunicator collectives. The algorithms are implemented on top of point-to-point communication routines. The algorithm selection in this component can be done in either of the following three ways: via compiled, predefined decision functions tuned for low-latency high-bandwidth interconnects; via user-specified command line flags; or using a rule-based run-length encoding scheme that can be tuned for a particular system.

  Our group at the Innovative Computing Laboratory at the University of Tennessee is the primary developer of this component and the component is described in more detail in Section A.1.2.
- **hierarchical**, which features collective implementations for collectives in hierarchical systems (such as system of clusters, or cluster of large SMPs). This component can use the locality information to try to utilize the most efficient algorithm for the particular subset of processes in the communicator. For example, consider a cluster of large SMPs connected by the MX interface. If we were to execute a broadcast operation, we would most likely want to utilize MX only for communication between different nodes and to use shared memory in between. A hierarchical implementation of broadcast could be optimized for the particular usage case.
- **shared memory**, which provides implementation of different collective operations on shared memory systems.

- **non-blocking**, which provide a high-performance, prototype implementation of non-blocking collective operations. The MPI standard does not define non-blocking collective operations. However, given the appropriate hardware, some application could greatly benefit from communication/computation overlap. This component is a test-bed for the proposal for standard extension.

3.1.4 Collective operations in hardware

Some of the specialized, high-performance networks, e.g., Myrinet's MX, Open Fabrics (Infiniband), and Quadrics include hardware-level support for MPI. Network Interface Cards (NICs) for these networks have the processing power to offload protocol processing from the host CPU. They can bypass the operating system and interact directly with MPI processes. This reduces overall latency of communication and can increase bandwidth as the amount of local buffering can be reduced by allowing the NIC to write directly to user memory. Often, point-to-point communication requests can be matched on the NIC itself as well.

In addition to point-to-point communication support, Quadrics and Blue Gene/L interconnects provide NIC-based collective operations [Almasi et al., 2005, Petrini et al., 2001]. The performance gain by using hardware-based collectives can be orders of magnitude higher than using the most advanced software implementations based on point-to-point communication. In addition, some reported data suggests that the performance of hardware-based operations is more consistent than the software-based approach [Moody et al., 2003]. A possible explanation lies in the fact that once the operation is delegated to the NIC it is no longer subject to CPU scheduling and will progress independently.

The limitations of hardware-based collectives are that they often implement only a small subset of MPI collectives (such as broadcast, reduce, and barrier) and are sometimes limited in their applicability (e.g., they can be used only on full partition of the system, or integer datatype). Moreover, the processing capabilities of the NIC are often far worse than the host CPU's, so in reduction operations, they are useful only for rather small message sizes [Moody et al., 2003]. In these cases, to achieve good performance one still needs to properly select either software or hardware-based implementation when applicable.

In Quadrics, NIC-based barrier and broadcast operations are implemented on top of hardware multicast operation [Petrini et al., 2003]. In addition, NIC-based reduce was developed in [Moody et al., 2003]. Elan network cards do not provide native support for floating point operations, so the floating point operations were implemented in software. The reduce implementation in [Moody et al., 2003] supports three algorithms: serial (linear), *f-nomial tree*, and *f-nomial tree with split*. F-nominal trees are an extension of binomial trees in which root has f children instead of just 2. The *split* version of the algorithm breaks the message into a specified number of blocks, such that each group is responsible for performing computation in the particular block. This is similar to the segmentation discussed in 4.1, but in this case, the number of segments is specified.

The IBM's Blue Gene/L systems come with a number of different networks: torus network that is used for point-to-point communication, collective network that is used for optimized collectives and communication with I/O nodes, and global interrupt network [Almasi et al., 2005]. On a Blue Gene/L system, the global interrupt network can be used for implementation of an efficient hardware barrier. Broadcast and Alltoall(v) operations are implemented using both torus and collective networks. The broadcast implementation is based on a mesh algorithm described in [Chan et al., 2006b] and utilizes the multicast capabilities of torus network. The implementation is pipelined at the packet level. The alltoall implementation utilizes a linear algorithm with a twist: the packet injection rate is controlled at low levels to prevent overflooding and congestion. The major limitation of hardware-based collectives on the Blue Gene/L system is that they can be used only on a full system partition (64 continuous nodes).

3.2 Parallel communication models

An important aspect of collective algorithm optimization is understanding the performance of an algorithm in terms of different parallel communication models. This section surveys the current state of collective communication analysis using parallel communication models. Some of the most commonly used parallel communication models, Hockney, LogP/LogGP, and PLogP, will be described in detail in Chapter 4.

Grama et al. in [Grama et al., 2003] introduce basic collective communication operations, such as one-to-all and all-to-all broadcast, reduction, and personalized communication. They develop and analyze algorithms for linear, mesh, and hypercube network topologies. First, they provide an optimal algorithm for a linear/ring network topology and then they extend it to higher-dimension topologies by repeating the linear algorithm in each of the additional dimensions appropriately. This method generates ring and recursive doubling types of algorithms. Second, they consider message splitting to reduce bandwidth requirements for broadcast, reduce, and allreduce operations. Message splitting is not equivalent to message segmentation. Message splitting divides the original message, M, into the specified number of blocks, n, of approximately same size, $\frac{M}{n}$. Message segmentation specifies the size of the individual block m_s resulting in $n_s = \lceil \frac{M}{m_s} \rceil$ blocks. In message splitting, the number of blocks is often equal to the number of nodes in the dimension of interest, while the message segment size usually depends on the network characteristics.

In [Thakur and Gropp, 2003,Thakur et al., 2005], Thakur et al. use the Hockney model to assess the performance of allgather, broadcast, all-to-all, reduce-scatter, reduce, and allreduce collectives and determine whether a particular algorithm would perform better for small or large message sizes. Using this analysis coupled with extensive testing, they determine switching points between algorithms based on message size and whether the number of involved processors is an exact power of two or not. The Hockney model was used by Rabenseifner et al. in [Rabenseifner and Träff, 2004,Thakur et al., 2005] to estimate the performance of a tree-based reduce algorithm optimized for large messages.

Similarly to [Grama et al., 2003], Chan et al. [Chan et al., 2004] use the Hockney model to evaluate the performance of different collective algorithms on a $c \times r$ mesh topology. In [Chan et al., 2006b], models of N-dimensional minimum spanning tree and bucket algorithms are applied to optimize broadcast and scatter on an IBM Blue Gene/L supercomputer. The same authors expand the basic Hockney model to include contention in [Chan et al., 2006a]. Using these models, a heuristic for building tuned collectives on linear, mesh, and hyper-cube network topologies is developed. They expand the message splitting concept from [Grama et al., 2003] to treat basic collective algorithms as building blocks, and the collective operation is implemented by recursively splitting the communicator and message size using available algorithms.

Kielmann et al. [Kielmann et al., 1999, Kielmann et al., 2001] use the PLogP model [Kielmann et al., 2000] to find an optimal algorithm and parameters for topology-aware collective operations incorporated in the MagPIe library. The MagPIe library provides collective communication operations optimized for wide area systems. Across high-latency, wide-area links MagPIe selects segmented linear algorithms for collectives, while various tree-based algorithms are used in a low-latency environment.

Barchet-Estefanel et al. [Barchet-Estefanel and Mounié, 2004] use the PLogP model to evaluate performance of broadcast and scatter operations on intra-cluster communication. In [Barchet-Steffenel and Mounié, 2005], the authors extend the PLogP model to include "contention" and "supplemental" factors. The values of the additional factors cannot be directly measured and are determined by back-fitting models to the experimental data.

Bell et al. [Bell et al., 2003] use extensions of LogP and LogGP models to evaluate the performance of small and large messages on contemporary super-computing networks. Similarly to PLogP, their extension of the LogP/LogGP model accounts for the end-to-end latency instead of the transport latency. Additionally, they evaluate the potential for overlapping communication and computation on their systems. Bernaschi et al. [Bernaschi et al., 2003] analyze the efficiency of a reduce-scatter collective using the LogGP model.

Vadhiyar et al. [Vadhiyar et al., 2004] use a modified LogP model that takes into account the number of pending requests that have been queued and the types of non-blocking send operations (blocking, eager/immediate, and rendezvous). They use this model to find an optimal segment size for a particular instance of broadcast operation.

Martinasso and Méhaut in [Martinasso and Méhaut, 2006] develop a model for concurrent MPI communication over SMP clusters. This parallel communication model is based on Log(G)P and PLogP models, but it is extended to include MPI-level flow control: eager and rendez-vous protocols for point-to-point messages. They analyze the different communication patterns on SMP clusters and include resource sharing conflicts that occur due to the particular pattern.

3.3 Algorithm selection and automatic tuning

In our context, the tuning process is equivalent to selecting the most appropriate algorithm for a particular problem instance. Empowering a user with the capability to specify the most appropriate algorithm for his/her application, as in [Open MPI, 2005, Fagg et al., 2006] is useful. However, the real benefit to the user is when the process of selecting the best available algorithm is automatic.

Currently, most MPI implementations approach the algorithm selection problem using performance models combined with exhaustive testing [Thakur et al., 2005, Fagg et al., 2003, Kielmann et al., 1999, Rabenseifner and Träff, 2004, Chan et al., 2004]. The individual approaches were described in Section 3.1.

In [Chan et al., 2006a] the authors provide a strategy for building a tuned broadcast implementation. They consider a group of "basic" algorithms for broadcast, scatter, all-gather, etc., based on the minimum spanning tree and bucket approach. These algorithms are considered building blocks for a "hybrid" broadcast implementation. Hybrid algorithms use predefined strategies to split communicator and message size recursively into small and large groups such that the appropriate basic algorithm (possibly for different collectives) can be called. For example, broadcast for intermediate message size and large communicator size can be implemented as a sequence of scatter and allgather collectives. Thus, the notion of explicit "switching" points in the algorithm selection process is removed. However, the strategy for building an appropriate collective must include explicit decisions on what to select and what constitutes large- or small-enough message/communicator size.

In [Hartmann et al., 2006], the authors describe experimental results of their orthogonal tuning process.[*] Their approach works for a fixed communicator size. In the first step, they measure performance of all orthogonal variations of an algorithm. In the next step, the information about the best algorithm is stored in a lookup table indexed by message size. This lookup table is used at run time to select the optimal algorithm. [†]

Vadhiyar et al. [Vadhiyar et al., 2000] implement automatically tuned MPI collective operations. They use performance models coupled with modified hill-descent heuristics to reduce the search space for tuning of the selected subset of MPI collective operations. In addition, the authors experiment with dynamic topology reordering in order to minimize the delays that occur when an intermediate node does not enter the collective on time. In a dynamic reordering strategy, all processes notify the root process that they are ready to enter the collective by sending a zero-byte message. Root dynamically constructs the topology based on the rank ordering and actually starts sending the data to children processes immediately. The actual benefit of this approach is not evaluated in the paper. The authors conclude that the noise and randomness present in the experimental data is often high enough to prevent a purely mathematical modeling approach to achieve optimal performance.

Faraj et al. [Faraj et al., 2006] implement the STAR-MPI system which utilizes the Automatic Empirical Optimization of Software (AEOS [Whaley et al., 2001]) technique at run-time to dynamically select the best performing algorithm for the application on the platform. At the run-time, STAR-MPI monitors the performance of MPI collective operations and, when the performance degradation is detected, the algorithm in use is replaced with an algorithm with better expected performance.

In addition, the algorithm selection problem is addressed in many different fields and is approached using various techniques, from models to statistical learning methods.

Vuduc et al. construct statistical learning models to build system-specific decision functions for a matrix-matrix multiplication algorithm selection [Vuduc et al., 2004]. In their work, they consider three methods for decision function construction: parametric modeling; parametric geometry modeling; and non-parametric geometry modeling. The non-parametric geometry modeling uses statistical learning methods to construct implicit models of the boundaries/switching points between the algorithms based on the actual experimental data. To achieve this, Vuduc et al. use the support vector method [Vapnik, 1998].

Whaley et al. [Whaley et al., 2001] in the ATLAS project, use empirical search-based tuning to generate dense linear algebra matrix kernels optimized for the particular machine. The whole analysis needs to be performed only once on a particular system.

[*]Authors refer to mesh-based algorithms as "orthogonal."

[†]The approach presented by these authors is analogous to the part of the work done by this dissertation using run-length encoding. However, we consider a wider input parameter space (including at least communicator and message sizes, as well as additional information such as total data per process, etc.) for which lookup tables would not be sufficient.

The SALSA [Eijkhout et al., 2005, Bhowmick et al., 2007] project searches for suitable linear and nonlinear system solvers for sparse matrices. The authors use machine learning algorithms to generate functions that map linear systems to suitable solvers. The process consists of two steps: feature extraction, which identifies important properties of the problem, and classifier, which finds an appropriate solver for the problem. SALSA utilizes a multivariate Bayesian decision rule approach for this purpose. At the heart of the prediction system is a database that contains information about the sets of linear systems and performance of different solvers on those problems. The information in the database is divided into training and testing sets and is used to create appropriate classifiers. Finally, based on the features of the problem, the most appropriate solver is predicted and used to solve the problem. The performance results from previous runs are fed back to the database, so the system adjusts the heuristic decision making process to improve the solver's performance over time.

Lagoudakis and Littman, in [Lagoudakis and Littman, 2000] apply reinforcement learning techniques to the algorithm selection problem for recursive algorithms, such as the sorting and order statistics selection problem. Both of these problems are fully characterized by the input array size (n). The authors treat the algorithm selection problem like the Markov Decision Process (*MDP*) in terms of states (current instantiation of the problem), actions (different algorithms we can select), total cost (real execution time), and policy (mapping from problem instances to algorithms), such that the total execution time is minimized. In addition, logarithmic compression is used to reduce the size of the input parameter space. For recursive algorithms, the authors argue that the running time of an algorithm can be significantly different for small input sizes, but the difference diminishes as the input size grows. Their results show that this approach is feasible, especially in situations when the cost function can be precomputed and then used at runtime.

The SPIRAL project [Püschel et al., 2005] is a code generation framework for linear digital signal processing (*DSP*) transforms on individual platforms. The framework produces tuned C or FORTRAN source code for a particular DSP transform instance (e.g., DFT of size n) on the specified system. The framework consists of the following modules/levels:

1. The *algorithm level* provides the (recursive) mathematical formulation of DSP transforms. The generated algorithms can be optimized using domain-specific rules and the rules are applied in recursive fashion generating a rule-tree. The rule-tree preserves the information on how the transform was expanded, and thus defines the algorithm used to compute the transform. The final output of this module is the fully expanded formula in matrix representation, $y = A \cdot x$, where x and y are input and output vectors of appropriate size.

2. The *implementation level* converts the fully expanded formula to the domain-specific *Signal Processing Language (SPL)*. In the process, a number of mathematical optimizations are applied. This module is further capable of optimizing the SPL representation of the formulas to include platform specific optimizations, such as vector operations, fused multiply add operations, the amount of loop unrolling, etc. The output of this module is C or FORTRAN source code for the particular problem instance.

3. The *evaluation level* includes compilation and performance evaluation. In the compilation step, a program instance is created and performance of the particular implementation is measured. The most commonly used performance measure is the running time, although the system can utilize alternative measures (e.g., the number of L1 cache misses, number of fused add operations). The performance evaluation step feeds the performance information to the search/learning modules.

4. The *search/leaning* module is used to reduce the number of candidate implementations that would be generated and evaluated by the system. The search block controls the enumeration of algorithms and implementations at code generation time. The following search methods are supported in SPIRAL: exhaustive, random, dynamic programming, evolutionary, and hill climbing. The learning block is responsible for selecting the best rules to be applied at formula generation and implementation time. This block models the performance of individual ruletree nodes as a function of a small set of node features (size and stride of node, node's parent, children, grandchildren, and common parent). Using these models, the regression tree learner RT4.0 [Torgo, 1999] is applied to predict the running times for all nodes.

The current limitations of SPIRAL project are: its applicability to a narrow set of operations (discrete linear signal transforms); the fact that some of the optimizations (such as vectorization) can be used only on limited subset of supported transforms; and the fact that the resulting code is an out-of-place implementation of an algorithm created for a particular problem instance (e.g., DFT of size n), and not general DSP transform.

Chapter 4

Parallel Communication Models

This chapter discusses parallel communication models and their application to MPI collective algorithms. Section 4.1 provides descriptions of different algorithms for MPI collective operations. In addition to describing the existing algorithms, we introduce a number of new algorithms, such as split-binary broadcast, linear gather and alltoall with synchronization, ring allreduce algorithm with segmentation, and linear scan algorithm with synchronization. Section 4.2 provides details about some of the most commonly used point-to-point communication models: Hockney, LogP/LogGP, and PLogP. Section 4.3 contains the main contribution of this chapter: detailed performance models of all of the discussed algorithms built on top of point-to-point communication models. Finally, in Section 4.4 we evaluate and analyze the applicability of developed models.

4.1 Algorithms for MPI Collective Operations

This section discusses different algorithms for MPI collective operation.

4.1.1 Virtual topologies

MPI collective operations can be classified as either one-to-many / many-to-one or many-to-many operations. For example, broadcast and scatter follow the one-to-many communication pattern, reduce and gather are examples of many-to-one, and barrier, alltoall, allreduce, and allgather employ the many-to-many communication pattern.

A generalized version of the one-to-many / many-to-one type of collective operations can be expressed as *i)* receive data from preceding node(s), *ii)* process data, if required, *iii)* send data to succeeding node(s). The data flow for this type of algorithm is unidirectional. Virtual topologies can be used to determine the preceding and succeeding nodes in the algorithm, and a single algorithm can be implemented to support the data flow. Moreover, segmentation of data in these algorithms is relatively easy to implement. Figures 4.1, 4.2, and 4.3 display the topologies we investigated.

The process ranks in virtual topologies we consider can be changed. In general, we aim to grow balanced trees - such that in the case of binary tree for example, the number of nodes in the left and right subtree can differ for at most one. However, less balanced trees with appropriate rank ordering can be beneficial to different collective operations.

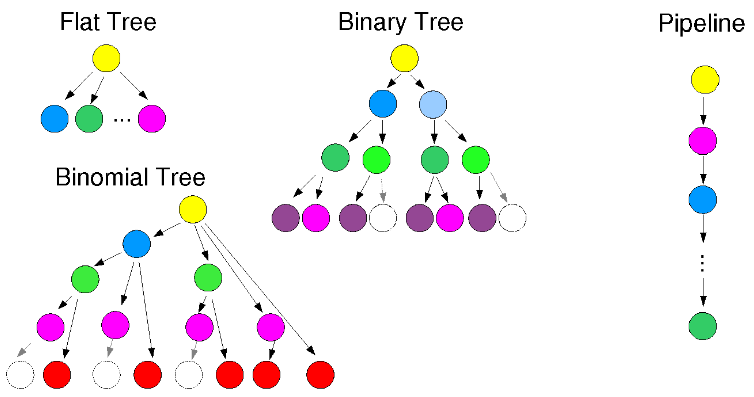


Figure 4.1: Virtual topologies

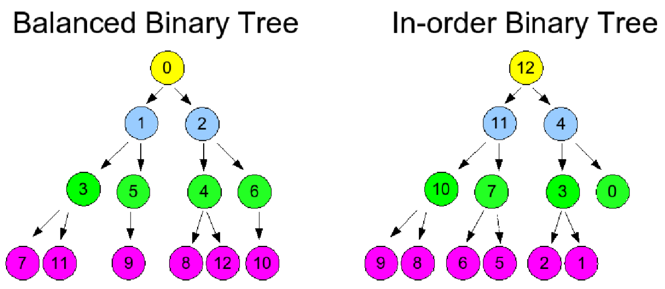


Figure 4.2: Balanced vs. In-order binary tree topologies

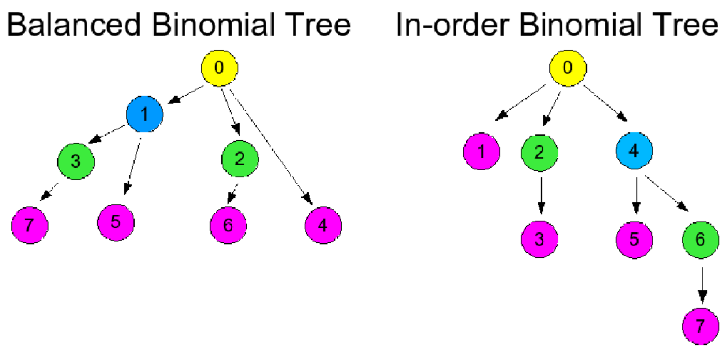


Figure 4.3: Balanced vs. In-order binomial tree topologies. In order to minimize latency for the in-order binomial tree, we need to send messages to the children in reverse order.

Figures 4.2 and 4.3 show two variants of binary and binomial trees that can be used by the algorithms we consider. In these figures, the same color of a node denotes that these nodes receive/send messages at the same algorithm time-step.

4.1.2 Collective Algorithms

This section discusses different algorithms for MPI collective operations. For consistency reasons, we describe both existing and newly proposed algorithms in the same manner. However, we explicitly state when the algorithm we describe has not been previously published.

The following are the algorithms which are original contributions of this dissertation:

- Split-binary broadcast algorithm with and without segmentation,
- Linear gather algorithm with synchronization,
- Linear alltoall algorithm with synchronization,
- Ring allreduce algorithm with segmentation, and
- Linear scan algorithm with synchronization.

Throughout the text, these algorithms are denoted with a "*".

Barrier

The barrier operation is often implemented using one of the following algorithms:

- **Double Ring**
  The processes exchange a token in a ring from left to right when the token reaches a process for the second time, the process forwards it to its right neighbor, and exits the barrier. This algorithm requires $2 \cdot P$ steps, where P is the number of processes in the communicator.

- **Fan-in-fan-out**
  A process is deterministically selected to act as a root of the operation (usually process 0 or (communicator size $-$ 1)). Non-root processes report by sending a zero-byte message to the root process. Once all non-root processes report, root sends back a zero-byte exit message to all other processes. Once the exit message is received, a process can leave the barrier.

  In the linear version of this algorithm, non-root processes send messages directly to the root and vice versa. The total number of exchanged messages is still $2 \cdot P$, but the running time performance of this algorithm is quite different from the Double Ring algorithm.

  Alternatively, this algorithm can be viewed as a zero-byte gather operation followed by a zero-byte broadcast operation. In this case, it is possible to reduce the total number of messages and communications from $O(P)$ to $O(log_2(P))$.

- **Recursive doubling**
  This algorithm requires $O(log_2(P))$ steps to complete. At step k, rank r exchanges a zero-byte message with rank (r XOR 2^k). At the end of $log_2(P)$ steps, the algorithm guarantees that all processes have entered the barrier, and thus everyone is allowed to leave. In the case when the communicator size is not an exact power of two, the algorithm requires extra steps:

  1. In the initial step, the largest power of two less than the communicator size, adj_size, is determined. All processes whose rank, r, is greater or equal to adj_size send a zero-byte message to process $r - adj\_size$.
  2. Processes 0 through adj_size execute the basic step described above.
  3. In the final step, ranks $r \in \{0, ...P - adj\_size\}$ send the exit message to processes $(r + adj\_size)$.

- **Bruck / Dissemination algorithm**
  This algorithm requires $O(log_2(P)) = \lceil log_2(P) \rceil$ steps, regardless of P. At the step k, process r sends a message to rank $(r + 2^k)$ and receives message from rank $(r - 2^k)$ with wrap around. At the end of $\lceil log_2(P) \rceil$ steps, the algorithm guarantees that all processes entered the barrier, so the operation completes.

Broadcast

We consider two types of broadcast algorithms: generalized segmented broadcast with virtual topologies and split-binary broadcast. Both implementations support message segmentation.

- **Generalized broadcast with virtual topologies**
  This algorithm implements a broadcast operation as a communication pipeline: For all message segments, process r receives the segment s from the parent $parent(r)$, and forwards it to all of its children $children(r)$. In the actual implementation, messages are preposted using non-blocking communication such that the arrival of the next segment from the $parent(r)$ is overlapped with the forwarding of the current segment s.

  We have considered **pipeline**, **flat tree**, **binary tree**, and **binomial tree** virtual topologies for this algorithm (See Figure 4.1).

- **Split-binary tree algorithm ***
  We introduced the split-binary tree algorithm as an optimization of the regular binary tree broadcast algorithm. The algorithm consists of two phases. In the first phase, the original message is split in half at root, and the left and right halves are sent down the left and right subtrees, respectively, using the same pipeline as in the generalized broadcast case. At the end of this phase, all ranks in the left subtree contain the left half of the original message, and the ranks in the right subtree contain the right half of the original message. In the second phase, each of the ranks in the tree finds its pair from the opposite subtree and exchanges its respective half of the message. If the tree had an even number of processes, the last leaf does not have a pair in the right subtree. To compensate for this, root sends the right half of the message to it.

Scatter(v)

The scatter operation is usually implemented in one of the following two ways:

- **Linear algorithm**
  This algorithm follows the definition of the scatter operation given in the MPI standard [Snir et al., 1998]. With communicator of size P, the root process issues $(P-1)$ (non-)blocking send operations containing appropriate buffers to the remaining processes. In case the operation was not specified as an "in-place," the root also performs a local copy of its own data.

- **Binomial algorithm**
  This algorithm utilizes a in-order binomial trees (Figure 4.3) to decrease the total number of steps necessary to complete this operation. If the root of the operation is not rank zero, a local shift operation needs to be performed to ensure that all messages contain only contiguous data. The root process sends a message to each of its children containing data for all offsprings of the particular child. The reduction of the total number of steps is achieved by increasing the total bandwidth requirement of the algorithm and allocating temporary buffer space on non-leaf nodes.

  This algorithm is hard to generalize for any virtual topology, because only properly ordered topologies allow for contiguous data to be distributed.

The scatterv operation can be implemented using the **linear algorithm** in the same way the scatter operation is implemented. However, additional optimizations of this algorithm are hard to achieve since send counts and displacements values are only known at the root process. Thus, to utilize tree-based algorithms, one must broadcast the send counts array first. The tree-based algorithms are primarily used to reduce the total latency of an operation and, due to increased bandwidth requirements are not particularly beneficial for large message sizes. Thus, one can expect that broadcast operation preceding the tree-based scatterv operation would annul savings achieved by reducing the total number of steps.

Gather(v)

The gather operation can be implemented as an inverse of the scatter operation. Thus, both linear and binomial algorithms can be utilized.

- **Linear algorithm without synchronization**
  In this algorithm, non-root nodes send their messages to root, who posts (non-blocking) receives from everyone. If the operation was not denoted as "in-place," the root must perform a local copy of its own data.

- **Linear algorithm with synchronization ***
  A potential problem with the linear gather algorithm without synchronization is overloading the root node with unexpected messages when the communicator or the message size is large.

  We introduce an alternative version of this algorithm that utilizes the explicit synchronization to prevent overloading of the root process. The incoming message on non-root processes is split into two segments. The root process gives an explicit green-light (a zero-byte message) to a child process to send its first segment only after the first segment from the previous child has arrived. By doing this, the second segment from the previous child is overlapped with the first segment of the next child, and the number of unexpected messages at the root process is reduced. All the non-blocking requests for the second segment are waited for at the end of the communication loop. This prevents the strict synchronization but it still limits the number of non-root processes that send data at the same time to the root process.

The following paragraph provides a pseudo-code description of the algorithm:

```
– Non-root process
    1. Wait for the zero-byte message to arrive from the root
    2. Send the first segment of the data to the root using blocking send
    3. Send the second segment of the data to the root process.
– Root process
    1. For every non-root process, $r_i$:
        (a) post non-blocking receive for the first segment of the message from $r_i$
        (b) send zero byte message to the $r_i$
        (c) post non-blocking receive for the second segment of the message from $r_i$
        (d) wait for the first segment from $r_i$ to arrive
    2. perform local copy if needed
    3. wait for all the requests for the second segment to complete.
```

- **Binomial algorithm**
  The binomial algorithm for gather utilizes in-order binomial tree topology (Figure 4.3). Leaf nodes send their data to their parent processes immediately. Intermediate nodes in the tree wait to receive the data from all children before forwarding the complete buffer up the tree. Once the root node receives all data, a local data shift operation may be necessary to put the data in correct place.

  As in the case of binomial algorithm for scatter operation, this algorithm is hard to generalize to other virtual topologies.

The gatherv operation is implemented using the **linear algorithm** in the same way the linear gather algorithm is implemented. Similarly to scatterv, the additional optimizations of the gatherv algorithm would require a broadcast of receive counts and displacements. Thus, we decided only to consider the basic gatherv implementation.

Allgather(v)

The allgather operation can be implemented using numerous algorithms.

- **Gather + Broadcast**
  The basic implementation of allgahter calls a gather operation to a preselected root, followed by a broadcast operation. The performance of this operation depends on the performance of the actual gather and broadcast algorithms used for this purpose. In case we utilized logarithmic, tree-based algorithms, the latency of this implementation

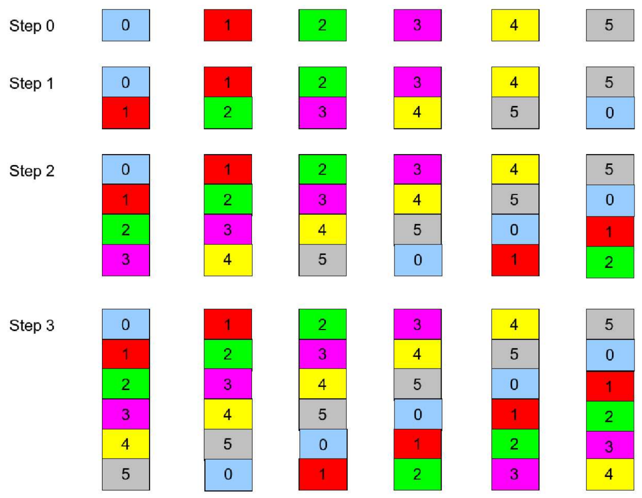


Figure 4.4: Communication loop of the bruck algorithm for allgather on six processes. For space reasons, we only show parts of the receive buffers that contain actual data.

will be $2 \cdot O(log_2(P))$. Since in the allgather operation, every process contains data for another process, more optimal algorithms exist.

- **Bruck**
This is a modification of the all-to-all algorithm described in [Bruck et al., 1997]. The algorithm takes a total of $\lceil log_2(P) \rceil$ steps. To ensure that the contiguous data is being exchanged, each of the processes performs a local shift operation at the beginning of the operation such that its local buffer becomes the first block locally. In the main communication loop, at step k, rank r sends a message to the rank $r - 2^k$ and receives a message from the rank $r + 2^k$. Figure 4.4 shows the communication loop for this algorithm on six processes. At the end of the loop, another local shift operation needs to be performed to ensure that the received blocks are on correct locations. The local shift operations require the allocation of temporary buffer space.

- **Recursive doubling**
The recursive doubling algorithm for allgather is a variation of the bruck allgather algorithm when the communicator size is an exact power of two. In this case, at step k rank r exchanges the message with rank r XOR 2^k. Figure 4.5 shows the communication loop for the four process case. An added benefit of this algorithm is that we do not need to perform local shift operations.

  This algorithm can be adapted to work with the non-power of two number of processes, but in that case the total number of steps is doubled. We only consider the version of the algorithm with the power of two processes.

- **Ring**
The ring algorithm for allgather requires $P - 1$ steps and utilizes only the nearest neighbor communication. At every step k, rank r receives a message from rank $(r - 1)$

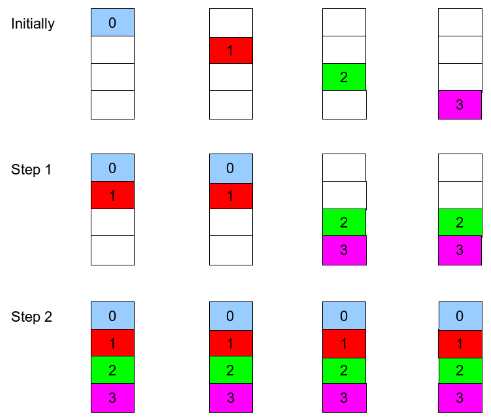


Figure 4.5: Recursive doubling algorithm for allgather on four processes.

containing data from rank $(r-k-1)$ and sends a message to rank $(r+1)$ containing data from rank $(r-k)$ with wraparound. This algorithm does not require any additional temporary buffer space.

- **Neighbor exchange**
  The neighbor exchange algorithm was introduced by Chen et al. [Chen et al., 2005] as an optimization of allgather ring algorithm for even communicator sizes. The algorithm takes $P/2$ steps. In the first step, rank r exchanges its data with $rank$ XOR 1. In subsequent steps, even and odd nodes will exchange the latest data they received with their nearest neighbors, but the neighbor will switch every step. Figure 4.6 shows the communication loop for six processes.

Allgatherv receive counts are known to all processes in the communicator. Thus, essentially all allgather algorithms can be modified for the allgahterv operation. Since displacements may not be contiguous, the allgatherv version of the algorithms may need to resort to using uncontiguous datatypes.

Alltoall(v/w)

We consider the following alltoall implementations.

- **Linear without synchronization**
  The linear algorithm provides the most basic alltoall implementation. Each process posts $(P-1)$ non-blocking receive requests, followed by $(P-1)$ non-blocking send requests, performs a local copy of data, and then waits for the requests to complete. In order to avoid overwhelming a single node, each of the processes posts its send requests starting with the next node $(rank+1)$ with wraparound. The linear algorithm does not require any additional buffering.

- **Linear with synchronization ***
  The basic implementation of the linear algorithm can have scalability problems when

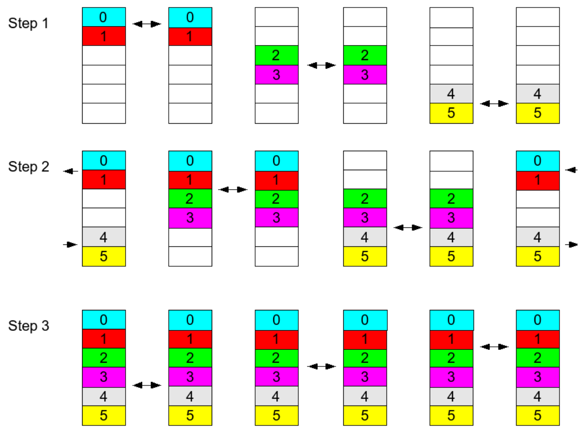


Figure 4.6: Neighbor exchange algorithm for allgather on six processes.

the communicator size is large. For example, one can expect that posting 10,000 non-blocking requests would incur a significant overhead. Thus, we provide a version of the linear algorithm that at any given time has at most *max_reqs* outstanding requests. As requests complete, new requests are posted. As in the case of the basic linear algorithm, the order in which requests are posted should minimize the request matching times.

- **Pairwise exchange**
  The pairwise exchange algorithm implements a fully synchronized version of the alltoall algorithm. The algorithm takes $(P-1)$ steps and does not require additional buffering. At step k, rank r sends the data block $(r+k)$ to rank $(r+k)$ and receives the data block $(r-k)$ from rank $(r-k)$ with wrap around. The synchronous nature of this algorithm ensures that, in a fully connected system, no process will be overwhelmed by the amount of data it receives.

- **Bruck algorithm**
  The bruck algorithm for the alltoall operation was described in [Bruck et al., 1997] and referred to as the index algorithm. The algorithm takes $\lceil log_2(P) \rceil$ steps to complete. Initially, all the processes perform a local shift operation such that the local data ends up as the frist block of the receive buffer. In the communication loop, at step k, blocks whose index in binary representation has k^{th} bit set to one are sent to the rank $(r+2^{(k-1)})$ and received from the rank $(r-2^{(k-1)})$. In addition, at every step, newly received data needs to be copied to the active sending buffer. Figure 4.7 shows the communication loop for this algorithm on six processes. After the communication loop, all the blocks are on their respective destination processes, but another local shift operation needs to be performed to place them in correct order. The reduction in the number of steps of this algorithm is achieved by utilizing the temporary buffering and extra bandwidth requirements.

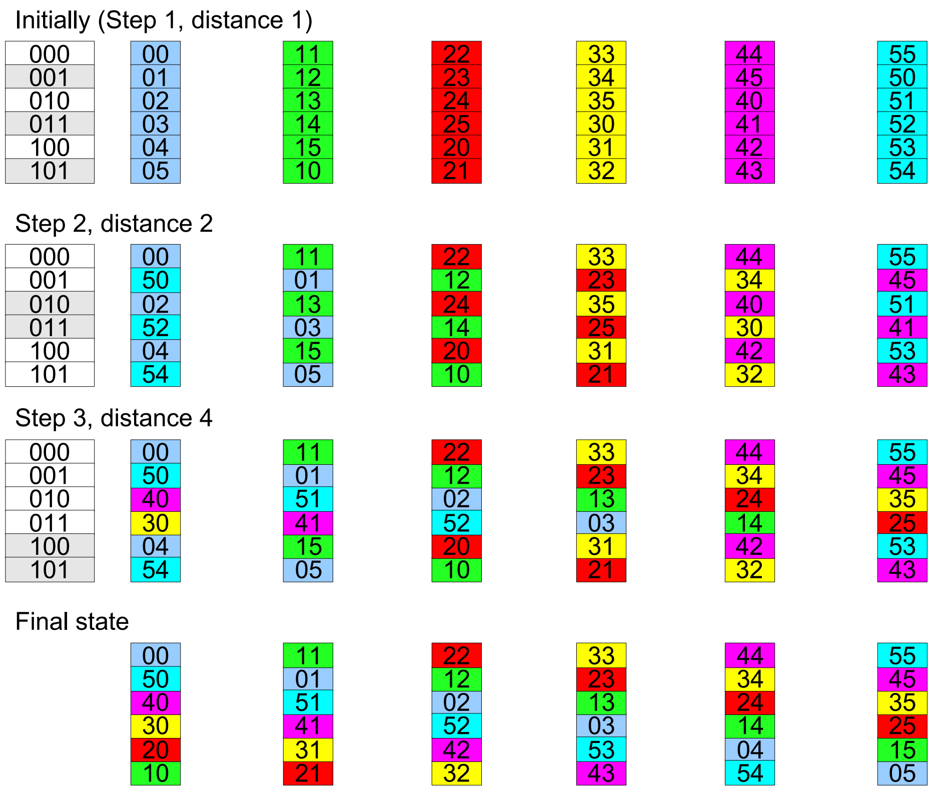


Figure 4.7: Communication loop of bruck algorithm for alltoall on six processes.

Both alltoallv and alltoallw can be implemented using either the **linear** or **pairwise exchange** algorithm. Unlike in the case of the allgatherv operation, the counts, displacements, and datatype parameters for these operations are known between the pairs of processes. Thus, in order to utilize algorithms such as bruck, an allgather operation with counts, displacements, and datatype parameters would have to be executed to ensure proper allocation of temporary buffers and creation of non-contiguous datatypes at every step. As in the case of scatterv and gatherv, we decided to focus only on the two basic implementations of alltoallv and alltoallw operations.

Reduce

We consider the following algorithms for the reduce operation:

- **Generalized reduce with virtual topologies**
  This algorithm implements the reduce operation as a communication pipeline: for every segment s, rank r receives corresponding segments from all of its children, applies the specified operation, and forwards the complete result for segment s to the parent process. The actual implementation of the algorithm ensures that there exist overlap between the communication (receiving the data) and computation (applying operation on already received segments).

  Since the order of operations is relevant for non-commutative operations, we consider only **linear** and **in-order binary** topologies for these operations. For commutative operations, we have analyzed the **pipeline**, **flat tree**, **binary tree**, and **binomial tree** virtual topologies for this algorithm. (See Figure 4.1).

  Unlike other virtual topologies and in order to achieve the best performance for binomial topology, we reverse the communication/computation loop to: for every child, for every segment. The models presented later assume this ordering.

- **Rabenseifner's algorithm and its variations**
  This original Rabenseifner algorithm was introduced in [Rabenseifner, 1997], and later expanded and improved in [Rabenseifner, 2004, Rabenseifner and Träff, 2004]. The original algorithm is designed to improve the performance of reduction operations for large message sizes. The algorithm distributes the computation of a reduction operation on all nodes, while utilizing $O(log_2(P))$ steps.

  The algorithm conceptually implements the reduce and allreduce operations as a reduce_scatter followed by gather and allgather, respectively. At every step, the computation cost is split between two communicating nodes by exchanging half of the data. Figure 4.8 shows the computation, reduce_scatter, phase for both the reduce and allreduce operations using this algorithm. For the case when the process count, P, is a power of two case, at every step, k, an active part of the input buffer of rank r is split in half, and one of the halves is exchanged with the partner rank, r XOR 2^k. Each of the ranks continues the computation only on the most up to date part of its buffer. At the end of the computation loop, each of the ranks is responsible for (approximately) $1/P$ portion of the data.

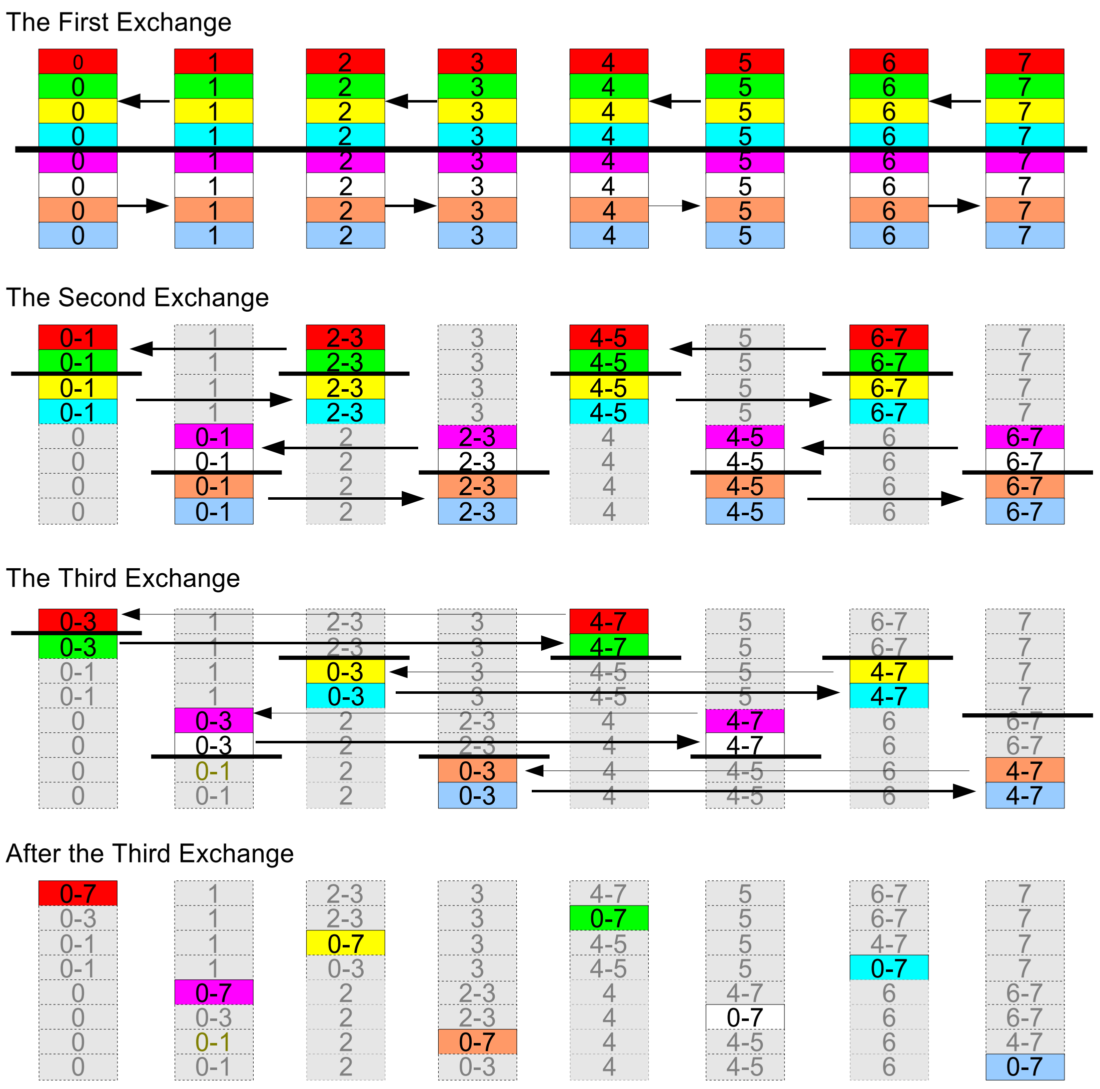


Figure 4.8: Computation phase on eight processes for Rabenseifner's original reduction algorithm. Shaded areas in data buffers are not exchanged.

At the end of the computation loop, an appropriate gather operation is invoked. The different variations of this algorithm, such as the **binary blocks** algorithms, differ mainly in the way they handle non-power-of-two communicator sizes.

In order to minimize number of local copy operations, this algorithm implements operation as result = lhs *op* rhs. The algorithm itself maintains the order of operations that would work for a non-commutative case. However, the requirement for MPI_Op to support three arguments means that each of the operations must be implemented separately, implying that user defined operations cannot be used directly.

Other authors consider **recursive doubling** algorithm. While the actual implementation is different, the communication and computation pattern of this algorithm is equivalent to the **non-segmented binomial tree** algorithm. Thus, we decided not to consider this algorithm separately.

Reduce_scatter

The reduce_scatter operation can be implemented using the following algorithms:

- **Reduce + scatterv**
  The basic implementation utilizes the reduce operation followed by the corresponding scatterv operation. The cost of this algorithm depends on the underlying implementation of each of the composing operations.

- **Recursive halving algorithm**
  The recursive halving algorithm implements reduce_scatter for commutative operations. For the power-of-two communicator size, P, at every step k, rank r exchanges data with rank r XOR $2^{log_2(P)-k}$. Only data necessary for the computation in the other group of processes is exchanged - the number of exchanged blocks is halved at every step. The actual amount of transferred data depends on the receive counts of the operation. Figure 4.9 shows an example of the recursive halving algorithm for reduce_scatter on eight processes.

  For non-power-of-two communicator sizes, P, the number of processes is reduced to the nearest smaller power-of-two process count, $P - r$, in the similar manner the non-power-of-two case was handled by the recurisve doubling algorithm for Barrier. The r processes are selected to send their data to r processes that will remain in the $P - r$ group. Then, the recursive scatter operation is executed with $P - r$ processes, and the final results are sent to the "extra" processes at the end of operation.

  A similar algorithm, recursive doubling, exists for non-commutative operations. The data flow in this case is reverse - processes first communicate with their nearest neighbors and the larger amount of data is exchanged. Since the performance of this algorithm is slightly worse than the performance of recursive halving, we itentioonally did not consider this algorithm in this dissertation.

- **Ring**
  This algorithm implements the reduce_scatter operation for commutative operations. The algorithm requires $(P - 1)$ steps and $(2 \times \max\{\text{rcounts}\} + \Sigma\{\text{rcounts}\})$ extra

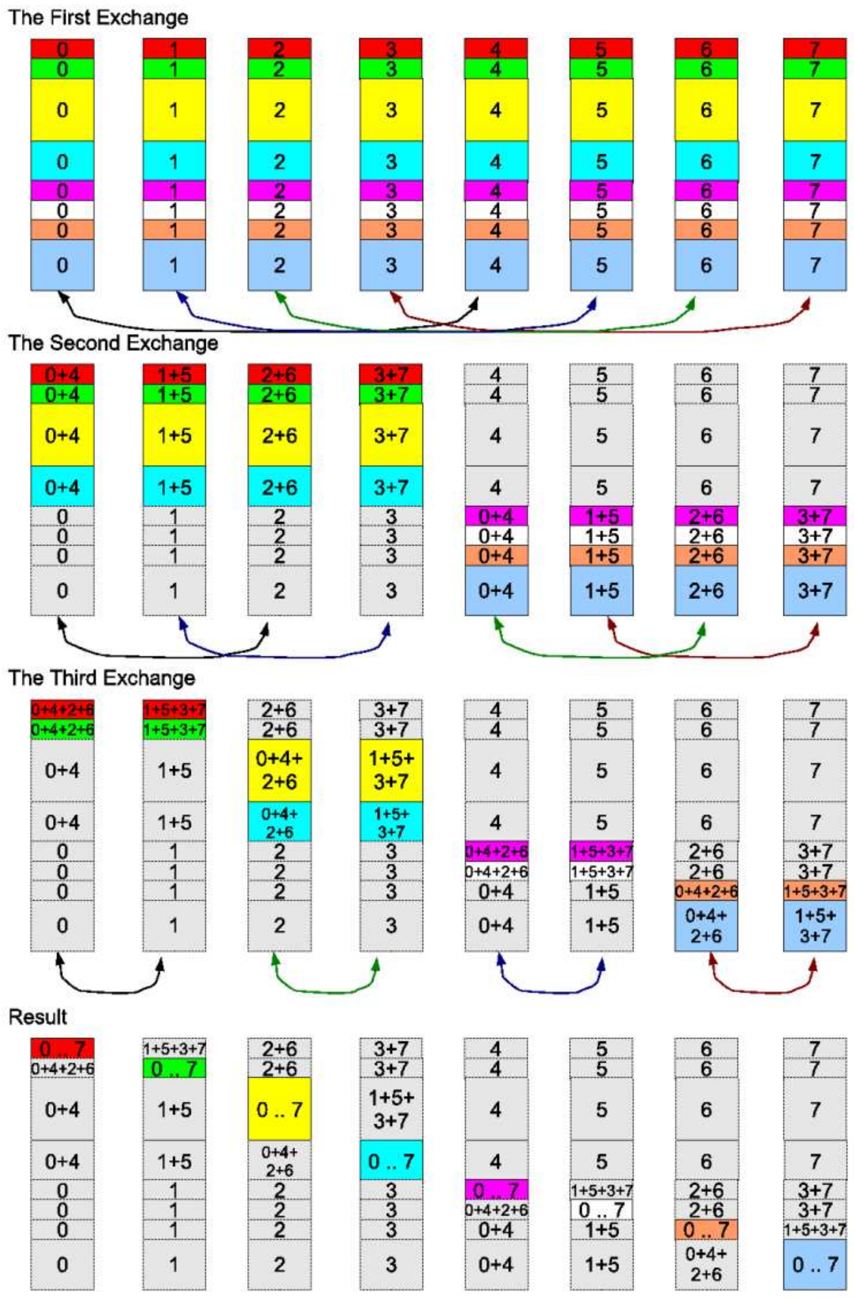


Figure 4.9: Recursive halving algorithm for reduce_scatter on eight processes. Shaded areas in data buffers are not exchanged.

buffering. An additional copy of the send buffer needs to be allocated to prevent the input buffer from overwriting. The algorithm proceeds as follows: at step k, rank r, posts non-blocking receive for block $(r-1+P-k)\%P$ from rank $(r+P-1)\%P$, waits on block $(r+P-k)\%P$ to arrive, applies the computation on that block, and then sends the result to rank $(r+1)\%P$.

Allreduce

Some of the common implementations of the allreduce operation are:

- **Reduce + Broadcast**
  The basic implementation of allreduce utilizes the reduce operation to a preselected root, followed by a broadcast operation. The cost of this algorithm depends on the underlying implementation of each of the composing operations.

- **Recursive doubling**
  The recursive doubling for allreduce is similar to the recursive doubling algorithm for allgather, discussed previously in Section 4.1.2. This algorithm preserves the order of operations so it can be used for non-commutative operations. When the communicator size, P, is an exact power of two, the algorithm takes $log_2(P)$ steps. At step k, rank r exchanges the whole data with rank r XOR 2^k, and applies the specified operation in correct order. In the case that the communicator size is not an exact power of two, the algorithm takes $\lfloor log_2(P) \rfloor + 2$ steps. The fist step is applied to reduce the number of nodes that will participate in the computation loop. The last step in this case is the distribution of the final result to the "removed" processes. Figure 4.10 shows an example of the recursive doubling algorithm for allreduce on six processes.

- **Rabenseifner's algorithm**
  The Rabenseifner's algorithm for allreduce was described in Section 4.1.2 when we discussed Rabenseifner's algorithm for reduce. The algorithm preserves the order of operations, but since it implements the MPI operation as result $=$ lhs $+$ rhs it cannot be used for user defined operations.

  Figure 4.8 shows the computation loop for this algorithm on eight processes. At the end of computation loop, an allgather operation is performed to ensure that all processes received the complete result. Different versions of Rabenseifner's algorithm will perform different allgather implementations. By default, recursive doubling is used.

- **Ring without segmentation**
  This is an allreduce algorithm for commutative operations. The unsegmented version of the algorithm takes $2*P-1$ steps to complete. The main benefits of this algorithm are automatic segmentation and a nearest neighbor communication pattern. The send buffer is split into P blocks of approximately the same size. If P does not divide send_count evenly, then the first $m =$ send_count $\%P$ blocks, *early blocks*, will have size $ebs = \lceil$ send_count $/P\rceil$, and the remaining $(P-m)$ blocks, *late blocks*, will have size $lbs = \lfloor$ send_count $/P\rfloor$. The main limitation of this algorithm is that it can be applied only when the send count is greater than the total number of processes.

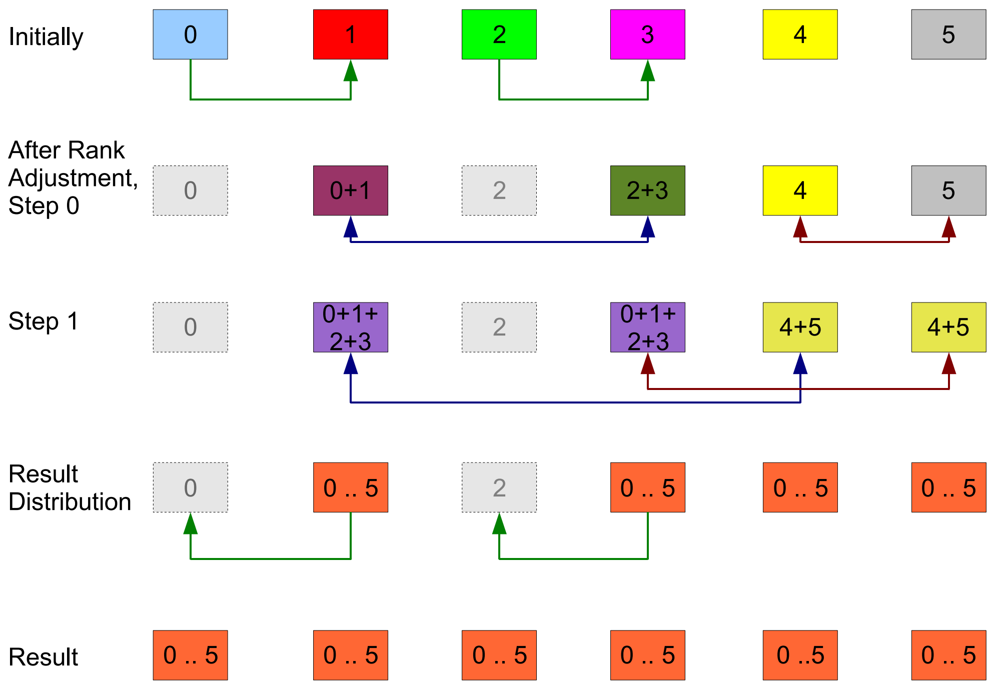


Figure 4.10: Recursive doubling algorithm for allreduce on six processes.

The algorithm takes two phases: computation and result distribution (allgather type of data exchange). In the computation phase, at step k, rank r posts the non-blocking receive from rank $(r-1+P)\%P$ for block $(r-k+P)\%P$, waits for block $(r-k+1+P)\%P$ to arrive, applies the specified computation on that block, and then sends the result to rank $(r+1)\%P$. Figure 4.11 shows an example of the computation loop on four processes. At the end of computation phase, each of the ranks has the final result for block $(r+1)\%P$. The result distribution is achieved by using an equivalent of the ring algorithm for allgather operation.

- **Ring with segmentation ***
  A segmented version of this algorithm breaks each of the blocks into segments of specified size. Figure 4.12 shows the data distribution, which corresponds to the initial state in Figure 4.11. The computation loop is then executed in block-cyclic fashion for each of the segment groups (a, b, and c in this example). There is the total of $\lceil block\_size/segment\_size \rceil$ computational phases that need to be executed. At the end of these phases, the result is distributed in the same manner as in the last step of Figure 4.11. The result distribution phase is the same in both algorithms. The main limitation of this algorithm is that it can be applied only when the send count is greater than the number of phases times communicator size, $P \times \lceil block\_size/segment\_size \rceil$.

Scan

The number of algorithms for MPI_Scan operation is discussed in [Sanders and Träff, 2006]: binomial tree, simultaneous binomial tree, and pipelined binary tree variants. However, in this dissertation we focus only on the following algorithms:

- **Linear without segmentation**
  The basic implementation of the scan collective requires P steps. At step k, process $r = (k+1)$ receives partial result from rank $(r-1)$, applies the specified operation, and sends the intermediate result to the rank $(r+1)$. After this step, rank r is done with the operation. This algorithm can be used for non-commutative operations.

- **Linear with segmentation ***
  In order to improve throughput of the linear algorithm, the computation can be overlapped with communication by utilizing segmentation. We propose the segmented version of the linear algorithm. This algorithm can be implemented in the following manner: for a process rank r:

  1. post non-blocking receive for the first segment from rank $(r-1)$
  2. for all remaining segments
     (a) post non-blocking receive for the current segment
     (b) wait on the previous segment
     (c) apply operation on the previous segment
     (d) send the previous segment to the rank $(r+1)$
  3. wait for the last segment

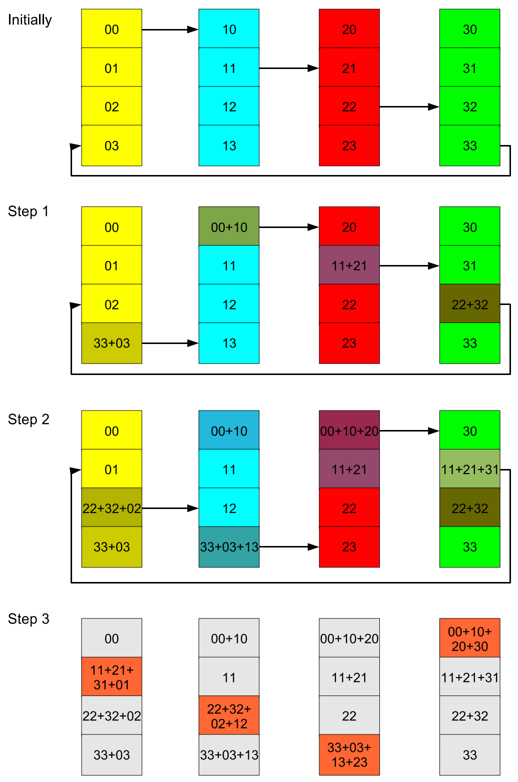


Figure 4.11: Computation loop of ring algorithm for allreduce on 4 processes.

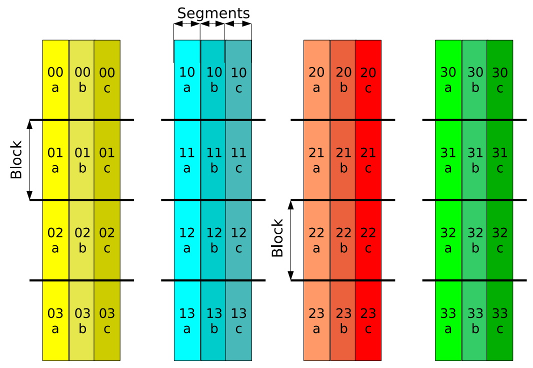


Figure 4.12: Initial data distribution for segmented ring algorithm for allreduce.

4. apply the operation on the last segment
5. send the last segment to the rank $(r + 1)$
6. rank r is done

- **Binomial algorithm**
  This logarithmic algorithm is referred to as the simultaneous binomial algorithm in [Sanders and Träff, 2006]. The algorithm takes $\lceil log_2(P) \rceil$ steps. At step k, rank r sends partial result to rank $r + 2^k$ and receives partial result from rank $r - 2^k$. The newly received data is combined with the current result to be sent out in the next iteration. This algorithm preservers the order of operations and can be used for non-commutative operations.

Exscan

The exclusive scan algorithms are equivalent to the scan algorithms with exception that the result should not contain a local component. All of the algorithms discussed in Section 4.1.2 can be easily adapted to the new operation.

4.2 Parallel communication models

This section introduces performance models of collective algorithms discussed in the previous section. Our work is built upon mathematical models of parallel communication. For better understanding of how we use these models, we describe them in more detail below. Since MPI collective operations consist of the communication and computation part of the algorithm, both network and computation aspects of the collective need to be modeled for any meaningful analysis.

4.2.1 Modeling point-to-point communication

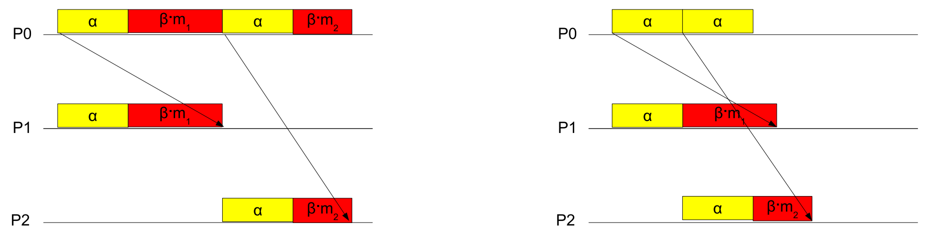
The parallel communication models (PCMs) are a basis for design and analysis of parallel algorithms. A good model includes the smallest possible number of parameters, while still being able to sufficiently capture the complexity of the run-time system across a number of possible criteria. This section introduces three commonly used point-to-point communication models: Hockney, LogP/LogGP, and PLogP.

Hockney model

The Hockney model [Hockney, 1994] assumes that the time to send a message of size m between two nodes is defined as

$$T = \alpha + \beta \cdot m \tag{4.1}$$

In the equation above, α refers to the message startup time, also refered to as latency, which includes the time to prepare the message (copy to system buffer, etc.) and the time for the first byte of the message to arrive at the receiver. β represents the transfer time per byte or reciprocal of the network bandwidth. The original Hockney model used asymptotic values for these two parameters.



(a) (b)

Figure 4.13: Hockney model (a) without, and (b) with communication overlap.

The time to receive message is $\alpha + \beta \cdot m$, but the time when the sending process can start sending another message can be defined in a number of different ways. In a worst-case scenario, no communication overlap is allowed, thus the sender must wait for the receiver to receive the message fully before it is allowed to initiate another send. In a best-case scenario, a process is allowed to initiate the second send after the latency has expired. Figure 4.13 depicts the two cases. In our analysis, we assume the worst-case scenario: a sender is allowed to initiate the next send operation after $\alpha + \beta \cdot m$ time.

One of the limitations of this model is that network congestion cannot be modeled directly. Different authors address this issue. For example, Chan et al. in [Chan et al., 2004] extend the basic model in the presence of network conflicts. The time to receive a message becomes $T = \alpha + k \cdot \beta \cdot m$, where k is the maximum number of network conflicts at the time. However, in our analysis, we did not consider these additional changes to the model.

LogP/LogGP models

The LogP model was introduced by Culler et al. in [Culler et al., 1993]. The model attempts to capture the properties of parallel computation in terms of the number of processors, the communication delay, the communication bandwidth (gap), and the communication overhead. The time to receive a message between two processes according to the LogP model is defined as

$$T = L + 2o \tag{4.2}$$

where L refers to network latency and o is the communication overhead. In LogP model, the sender is allowed to initiate new send operation after g period of time. This implies that the network allows transmission of at most $\lfloor L/g \rfloor$ messages simultaneously. Since none of the parameters depend on message size, the model assumes that only constant-size, small messages are communicated between the nodes. Figure 4.14 (a) displays a communication diagram for LogP model parameters.

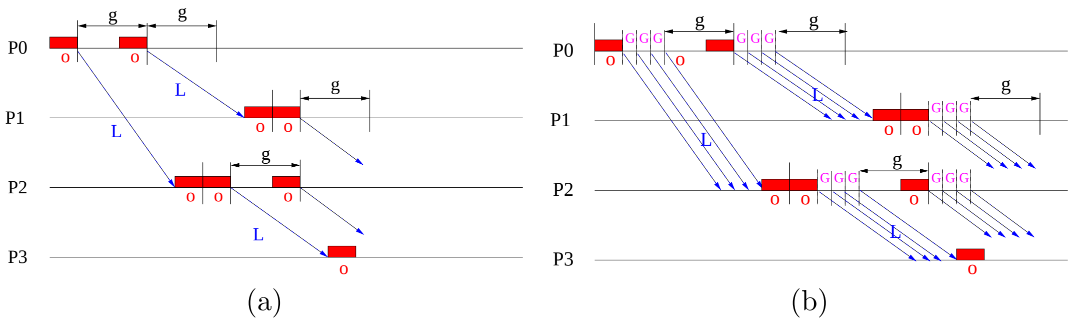


Figure 4.14: (a) LogP and (b) LogGP model parameters. The figure shows broadcast-like communication pattern: $P0 \rightarrow \{P2, P1\}$, $P1 \rightarrow \{...\}$, and $P2 \rightarrow \{P3, ...\}$.

LogGP was introduced in [Alexandrov et al., 1995] as an extension of the LogP model that handles large messages. The model introduces the *gap per byte* parameter G, to capture the cost of sending large messages across the network. The LogGP model predicts the time to send a message of size m between two processes as

$$T = L + 2o + (m-1)G \tag{4.3}$$

As in the case of the LogP model, the sender is able to initiate a new message only after time g expires. Figure 4.14 (b) shows the LogGP model parameters in a simple communication pattern.

PLogP model

The *Parametrized LogP* model, PLogP, [Kielmann et al., 2000] is an extension of the LogP model. The PLogP model is defined in terms of end-to-end latency L, sender and receiver overheads, $o_s(m)$ and $o_r(m)$ respectively, gap per message $g(m)$, and the number of processes involved in communication P. In this model, sender and receiver overheads and the gap per message parameters depend on the message size, m. The time to receive a message of size m in the PLogP model is defined as

$$T = L + g(m) \tag{4.4}$$

Figure 4.15 shows the PLogP model parameters in action in a simple, broadcast-like communication pattern.

The notion of latency and gap in the PLogP model slightly differs from that of the LogP and LogGP models. In addition to the message transfer time, latency in the PLogP model includes all contributing factors, such as copying data to and from network interfaces. The gap parameter in the PLogP model is defined as the minimum time interval between consecutive message transmissions or receptions, implying that at all times $g(m) \geq o_s(m)$ and $g(m) \geq o_r(m)$. If $g(m)$ is a linear function of message size m and L excludes the sender overhead, then the PLogP model is equivalent to a LogGP model that distinguishes between sender and receiver overheads. Kielmann et al. in [Kielmann et al., 2000] provide a

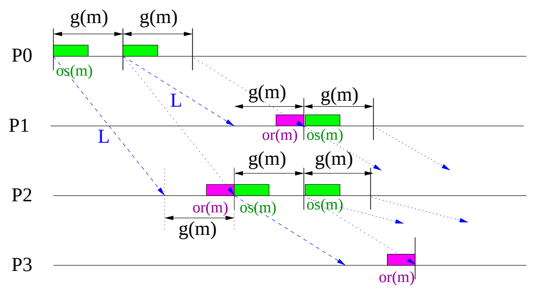


Figure 4.15: PLogP model parameters. The figure shows broadcast-like communication pattern: $P0 \rightarrow \{P2, P1\}$, $P1 \rightarrow \{...\}$, and $P2 \rightarrow \{P3, ...\}$.

LogP/LogGP		PLogP
L	$=$	$L + g(1) - o_s(1) - o_r(1)$
o	$=$	$\frac{o_s(1)+o_r(1)}{2}$
g	$=$	$g(1)$
G	$=$	$\lim_{m \rightarrow \infty} \left(\frac{g(m)}{m}\right)$
P	$=$	P

Table 4.1: LogP/LogGP parameters in terms of PLogP parameters.

transformation from PLogP parameters to LogP/LogGP parameters. Table 4.1 reproduces this dependence.

4.2.2 Modeling computation

We assume that the time spent in computation on data in a message of size m is $\gamma \cdot m$, where γ is the computation time per byte. This linear model ignores effects caused by memory access patterns and cache behavior, but is able to provide a lower limit on time spent in computation.

In addition, some of the algorithms require copying of local data from one memory location to another. Depending on the data size, the cost of this operation can be negligible or substantial. We model the time to perform a local copy of a message size m as $\delta \cdot m$, where δ is the local memory bandwidth. We expect that $\delta >> G$.

4.3 Performance models of MPI collective operations

For each of the implemented algorithms, we have created a numeric reference model based on the point-to-point communication models discussed in Section 4.2. We assume a full-duplex network which allows us to exchange or send-receive a message in the same amount of time as completing a single receive.

This section contains models of all algorithms discussed in Section 4.1. Due to large number of models, we present them in the tabular form. For readability and clarity purposes, we show three examples of building a performance models: segmented split-binary broadcast, linear gather with synchronization, and recursive doubling allgather algorithms.

Most of the algorithms we consider are well known in the literature. Thus, we provide reference to the authors who analyzed the algorithm using the particular point-to-point model as a base. The actual formula we give may differ from the one in other publications. The differences can be due to different notation (for example, specifying total message size instead of individual message size), or due to different assumptions (not assuming full-duplex network).

4.3.1 Building a performance model: split-binary broadcast

We start with an example of building the performance model of the split-binary broadcast algorithm. The goal is to broadcast a message of size m, to P processes using the segmented split-binary algorithm with segment size m_s and total of n_s segments. For simplicity purposes, assume that $m = n_s \cdot m_s$.

The split-binary algorithm utilizes the balanced binary tree virtual topology (See Figure 4.2) and contains two phases: forwarding and exchange. In the forwarding phase, at the root process, the original message of size m is split in half, and each of the halves is sent down the left and right subtree respectively. In addition, these halves can be segmented, so each of the subtrees will forward $\frac{n_s}{2}$ segments of size m_s. At the end of the forwarding phase, all processes in the left subtree contain the first half of the message, and all the processes in the right subtree contain the remainder of the message. In the balanced binary tree, the left subtree can have at most one extra node than the right subtree. Thus, every node, except possibly one, in the left subtree has a pair in the right subtree. In the exchange phase of this algorithm, node pairs exchange their messages and the broadcast operation completes. The unpaired node in the left subtree receives its second half of the message from the root process. Figure 4.16 depicts an example of a segmented split-binary broadcast on 7 processes when rank 0 is the root.

The following analysis assumes a complete binary tree, i.e., number of processes, P, is one less than an exact power of two, $P = 2^k - 1$. If the tree is not complete, the execution time will be similar to, but less than, the time to complete the broadcast operation on P^* processes, where $P^* = 2^{\lceil \log_2(P) \rceil}$.

The time to complete a broadcast operation using the segmented split-binary algorithm is the longest time taken by a process to execute the forwarding and exchange phases. Using the balanced binary tree topology with root rank 0, this corresponds to the process with the highest rank. In a 7-process case with the rank distribution as in Figure 4.16, the last process to finish the forwarding phase under ideal conditions is process $P6$.

$$T_{sbin} = T_{fwd} + T_{exchg} \tag{4.5}$$

The core of the split-binary algorithm is two independent pipelines: one in each of the subtrees. The root process feeds the data to the pipelines, but the actual rate at which data arrives to nodes can be slower than the insertion rate. The maximum cost of the forwarding phase is equal to the sum of times to fill up the pipeline plus the time to receive remaining $\lceil \frac{n_s}{2} \rceil$ segments at the pipeline rate.

$$T_{fwd} = \text{time to fill up the pipe} + (\lceil \tfrac{n_s}{2} \rceil - 1) \cdot \text{pipeline rate} \tag{4.6}$$

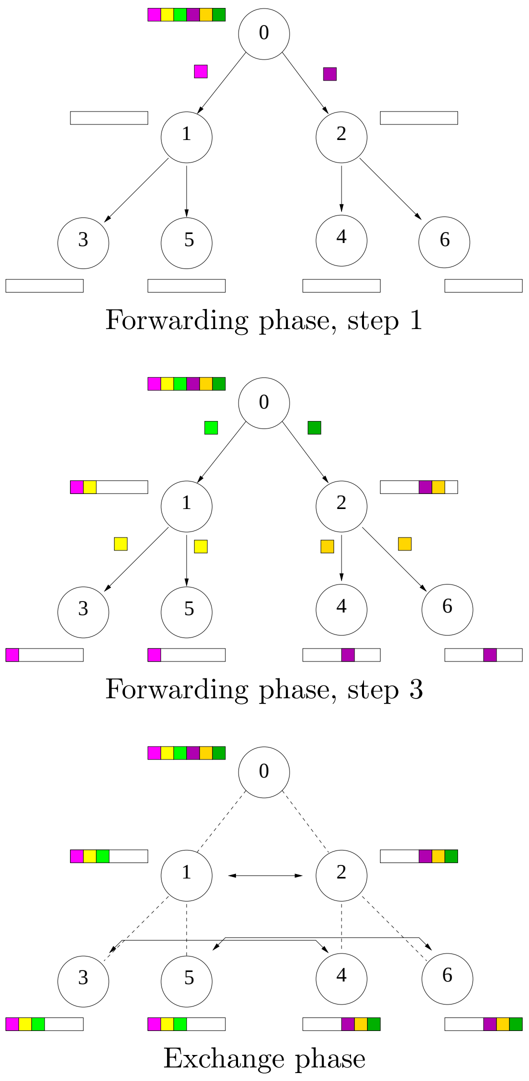


Figure 4.16: Example of segmented split-binary broadcast on 7 processes. The original message was split into 2×3 segments.

The time to fill up the pipeline is the time for the first segment to arrive at the right-most leaf, the process $(P-1)$ in this case. The pipeline rate is the maximum between the data insertion rate at the root and the time it takes the individual nodes to forward the data to their children.

Figure 4.17 shows the forwarding phase of this example according to different point-to-point models. We analyze the completion time separately for each of the models.

Hockney model

The time for the first segment to arrive at the "last" process is the time for the second segment to go down the right most branch of the tree. In a complete binary tree, the length of this branch is $\log_2(P+1)-1$. Thus, this time is equal to $(\log_2(P+1)-1)\cdot 2\cdot(\alpha+\beta\cdot m)$. Since we assume a full-duplex network, the pipeline rate in the Hockney model is the same as the pipeline insertion rate at the root $2\cdot(\alpha+\beta\cdot m)$. The time to complete the forwarding phase of the segmented split-binary broadcast under the Hockney model is then

$$\begin{aligned} T_{fwd} = &\ 2\cdot(\log_2(P+1)-1)\cdot(\alpha+\beta\cdot m_s)+ \\ &\ 2\cdot(\lceil \tfrac{n_s}{2}\rceil -1)\cdot(\alpha+\beta\cdot m_s) \\ T_{fwd} = &\ 2\cdot(\log_2(P+1)+\lceil \tfrac{n_s}{2}\rceil -2)\cdot(\alpha+\beta\cdot m_s) \end{aligned} \tag{4.7}$$

Adding the cost to exchange a message of size $\frac{m}{2}$, the time to complete the segmented split-binary broadcast under the Hockney model is

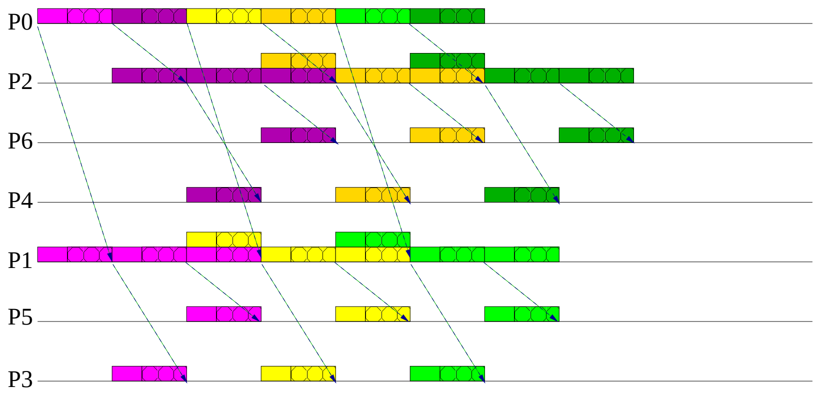
$$T_{sbin}^{Hock} = 2\cdot(\log_2(P+1)+\lceil \tfrac{n_s}{2}\rceil -2)\cdot(\alpha+\beta\cdot m_s)+(\alpha+\beta\cdot \tfrac{m}{2}) \tag{4.8}$$

LogGP model

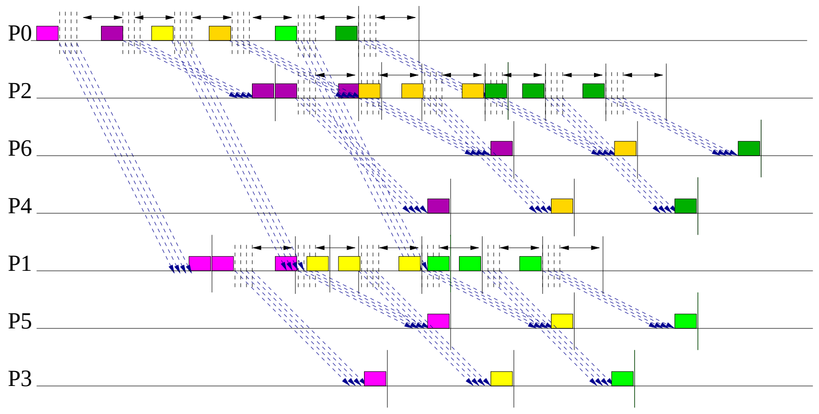
Applying the similar logic as in the Hockney-model case, we compute the total time as the time to receive first segment at the "last" node, plus the time to receive remaining $\lceil \frac{n_s}{2}\rceil$ segments plus the time to exchange the other half of the message.

The length of the most right branch, again, is $\log_2(P+1)-1$, and the cost of forwarding a message to a right child under LogGP model is $o+(m_s-1)\cdot G+g+(m_s-1)\cdot G+L+o$. The time to receive the first segment on the rightmost leaf under the LogGP model is $(\log_2(P+1)-1)\cdot(L+2\cdot o+2\cdot(m_s-1)\cdot G+g)$.

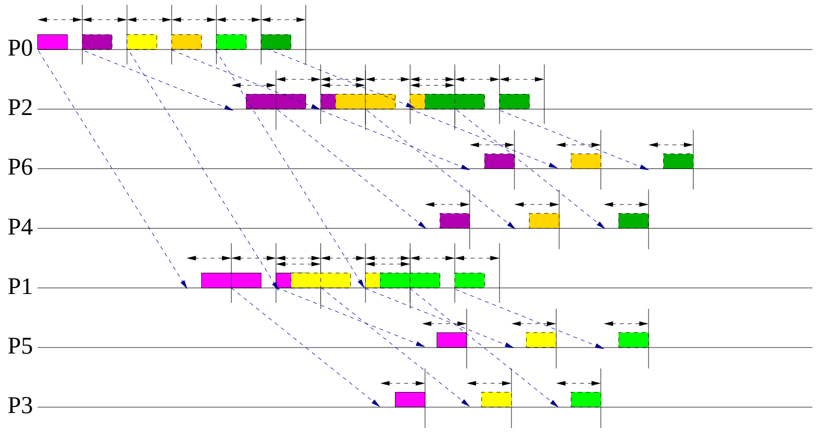
The pipeline rate is harder to compute in this case. Root inserts data into the pipelines every $2\cdot(g+(m_s-1)\cdot G)$ time-steps. However, the internal node must first receive the segment and then forward it to two of its children. In the best case, the next segment arrival overlaps with the second gap, maintaining the pipeline rate at $2\cdot(g+(m_s-1)\cdot G)$. This situation is shown in Figure 4.17 (b) where the arrival of the next segment overlaps with the second $(m_s-1)\cdot G+g$ period for the current segment. But, in the worst case, the new message arrival is not hidden, and the pipeline rate is $o+2\cdot(g+(m_s-1)\cdot G)$. As we are looking for the upper bound on the completion time, we model the pipeline rate with the worst-case scenario, $o+2\cdot(g+(m_s-1)\cdot G)$.



(a) Hockney



(b) LogGP



(c) PLogP

Figure 4.17: Forwarding phase of segmented split-binary broadcast timings in a 7-process case according to different point-to-point models: (a) Hockney, (b) LogGP, and (c) PLogP. The process ranks have been reordered to increase readability of the figure. The notation used on this figure corresponds to notation used in Figures 4.13, 4.14, and 4.15 to describe model parameters. The block colors corresponds to the segment colors used in Figure 4.16.

Adding the time to exchange a message of size $\frac{m}{2}$ to the equation, we get

$$\begin{aligned} T_{sbin}^{LogGP} = \; & (\log_2(P+1)-1)\cdot(L+2\cdot o+2\cdot(m_s-1)\cdot G+g)+ \\ & (\lceil \tfrac{n_s}{2} \rceil -1)\cdot(o+2\cdot(g+(m_s-1)\cdot G))+ \\ & L+2\cdot o+(\tfrac{m}{2}-1)\cdot G \end{aligned} \tag{4.9}$$

PLogP model

Similarly to the previous two cases, the time to send the first segment down the right-most branch of the tree under PLogP model can be computed as $(\log_2(P+1)-1)\cdot(g(m_s)+L+g(m_s))$. The root inserts packets down the pipeline every $2\cdot g(m_s)$ time-steps, and the nodes forward the message to their second child after either $2\cdot g(m_s)$ or $o_r(m_s)+g(m_s)+o_s(m_s)$ timesteps, whichever is larger.

Adding the time to exchange the second half of the message, we obtain the following formula for the segmented split-binary algorithm under the PLogP model

$$\begin{aligned} T_{sbin}^{PLogP} = \; & (\log_2(P+1)-1)\cdot(L+2\cdot g(m_s))+ \\ & (\lceil \tfrac{n_s}{2} \rceil -1)\cdot \max\{2\cdot g(m_s),\, o_r(m_s)+g(m_s)+o_s(m_s)\}+ \\ & L+g(\tfrac{m}{2}) \end{aligned} \tag{4.10}$$

4.3.2 Building a performance model: linear gather with synchronization

The linear gather algorithm with synchronization is introduced to prevent root process from being overloaded. The non-root processes split their incoming message into two: first segment and remainder of the message. The root process sends an explicit, zero-byte "go-ahead" message to the child process to start sending its data as soon as the first segment from the previous child has arrived. The detailed description of this algorithm is available in Section 4.1.

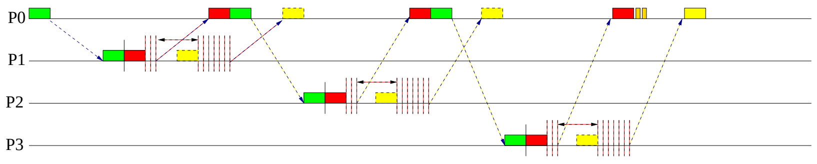
Hockney model

According to the Hockney model as we use it, the root process must be fully involved in the complete process of sending and receiving a message. However, since we utilize full-duplex network, root can send a zero-byte message to the next child, while it is receiving the second segment. Thus, the cost of the linear gather with synchronization is the cost to send the first zero-byte message, α, plus $(P-1)$ times the time to send the first and second segment, $\alpha+\beta\cdot m_s+\alpha+\beta\cdot(m-m_s)=2\cdot\alpha+\beta\cdot m$.

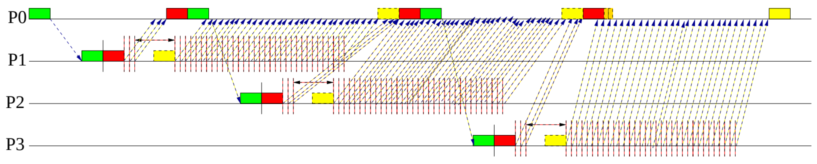
$$T_{lsync}^{Hock} = \alpha+(P-1)\cdot(2\cdot\alpha+\beta\cdot m) \tag{4.11}$$

LogGP model

Analyzing linear gather with synchronization under the LogGP model involves more steps. We must distinguish two cases. In the first case, at the root, receiving the second segment can be completely overlapped with the sending of the "go-ahead" message to the next child. This case occurs when the cost of sending the second segment $(g+(m-m_s-1)\cdot G+L+o)$ is less than the cost to send the "go-ahead" message and start receiving the first segment



(a)



(b)

Figure 4.18: Linear gather with synchronization cases analyzed using the LogGP model. (a) $g + m \cdot G < L + 2 \cdot (o + m_s \cdot G)$ (b) otherwise.

of the next child, $o + L + o + o + (m_s - 1) \cdot G + L$. After basic algebraic manipulation, this condition becomes:

$$g + m \cdot G \quad < L + 2 \cdot (o + m_s \cdot G) \tag{4.12}$$

In the latter case, this is not possible due to the size of the second segment, and messages are effectively serialized at the root process. Figure 4.18 depicts both cases.

The algorithm implementation does not wait for the second segment to arrive until all processes have sent their first segment. Thus, even though the second segment most likely arrives at the root process before that time, it will not be "waited-for" until the end of the operation. Most of the overhead associated with this message takes place at the time the message arrives, but some overhead will be incurred at the time of the "wait" call. Thus, in the Figure 4.18 we denote the receive overhead for the second segment in dashed lines. The "second" overhead should be negligible because all of the second segments should have been expected messages. In any case, the LogGP model does not provide tools for handling the non-blocking communication in an elegant manner.

Finally, if the condition in Equation 4.12 is met, the cost of this operation is $(P-1)$ times the time to send "go-ahead" message, $L + 2 \cdot o$, plus the time to receive the first segment, $o + (m_s - 1) \cdot G + L + o$, minus the $L + o$ for the first segment on the last node, and the time to receive the second segment from the last child, $g + (m - m_s - 1) \cdot G + L + o$. Otherwise, the cost of the operation is the cost to send the first "go-ahead" message plus the time to receive $(P-1)$ second segments in a series $o + (m_s - 1) \cdot G + g + (m - m_s - 1) \cdot G$ plus $L + o$ for the last segment.

$$T_{lsync}^{LogGP} = \left\{ \begin{array}{l} (P-1) \cdot (2 \cdot L + 4 \cdot o + (m_s - 1) \cdot G) + g + (m - m_s - 1) \cdot G, \\ \qquad\qquad \text{if } g + m \cdot G < L + 2 \cdot (o + m_s \cdot G) \\ 2 \cdot L + 3 \cdot o + (P-1) \cdot (o + (m-2) \cdot G + g), \\ \qquad\qquad \text{otherwise} \end{array} \right\} \tag{4.13}$$

PLogP model

Analogous to the LogGP case, we have to distinguish two use-cases for this algorithm. In the first scenario, the cost of sending the second segment $L + g(m - m_s)$ is less than the cost of sending the "go-ahead" message and receiving the first byte from the next child, $L + g(0) + L$. In the second case, the cost of receiving the second segment is high enough to completely hide the "go-ahead" message and the latency associated with the first segment of the next child. Thus, the cost of the algorithm becomes the cost for the first "go-ahead" message $L + g(0)$, plus the cost of the first segment, $L + g(m_s)$, and the remaining data will arrive every $g(m_s) + g(m - m_s)$ timesteps.

$$T_{lsync}^{PLogP} = \left\{ \begin{array}{ll} (P-1)\cdot(2\cdot L + g(o) + g(m_s)), & \text{if } g(m-m_s) < L + g(0) \\ 2\cdot L + g(0) + (P-1)\cdot(g(m_s) + g(m-m_s)), & \text{otherwise} \end{array} \right\} \tag{4.14}$$

4.3.3 Building a performance model: recursive doubling allgather

So far, we covered building models for algorithms that support segmentation and non-blocking communication. Now, we look at an algorithm which exchanges a different amount of data on every step. This example covers recursive doubling algorithm for allgather. For simplicity purposes, we assume that the number of processes P is an exact power of two, $(\exists k \in \mathbb{N})P = 2^k$.

In recursive doubling allgather, at every step k, process r exchanges a message with the process r XOR 2^k. Initially, only local data is sent, but on every step, all new data is exchanged. Both processes contribute data of size m, with the total buffer size $P \cdot m$. Thus, at every step, the amount of data being exchanged doubles.

Hockney model

Under the Hockney model, the cost of the recursive doubling allgather algorithm is simply the cost to perform $\log_2(P)$ message exchanges in which the message size doubles at every step: $\alpha + \beta \cdot m + \alpha + \beta \cdot 2 \cdot m + ... + \alpha + \beta \cdot 2^{log_2(P)-1} \cdot m = \log_2(P) \cdot \alpha + (2^0 + 2^1 + ... + 2^{log_2(P)-1}) \cdot m$. The series $(2^0 + 2^1 + ... + 2^{log_2(P)-1})$ adds up to $(P-1)$. Thus, the algorithm duration according to the Hockney model is

$$T_{rdbl}^{Hock} = \log_2(P) \cdot \alpha + (P-1) \cdot \beta \cdot m \tag{4.15}$$

LogGP model

Applying the same analysis as in the Hockney model case, the cost of recursive doubling allgather under LogGP is the cost to perform $\log_2(P)$ message exchanges. The only difference from the Hockney model is that in LogGP we must wait g time before we can send another message. Thus, the cost of the algorithm is: $o + (m-1) \cdot G + \max\{L + o,\ g\} + o + (2 \cdot m - 1) \cdot G + \max\{L + o,\ g\} + ... + o + (2^{\log_2(P)} \cdot m - 1) \cdot G + \max\{L + o,\ g\}$. After grouping, this equates to $\log_2(P) \cdot (o + \max\{L + o,\ g\} - G) + G \cdot \sum_{k=2}^{log_2(P)} 2^k \cdot m$, which gives us

$$T_{rdbl}^{LogGP} = \log_2(P) \cdot (o + \max\{L + o,\ g\} - G) + (P-1) \cdot m \cdot G \tag{4.16}$$

PLogP model

Finally, the PLogP-based performance model of recursive doubling allgather is similar to the Hockney and LogGP models we already developed. The cost of exchanging a message of size M in PLogP is $L + g(M)$. We can send another message after $g(M)$ time, which is automatically fulfilled in this case, so no need to max term. With all this in mind, the cost of this operation becomes $L + g(m) + L + g(2 \cdot m) + ... + L + g(2^{\log_2(P)} \cdot m)$, or

$$T_{rdbl}^{PLogP} = \log_2(P) \cdot L + \sum_{k=0}^{\log_2(P)-1} g(2^k \cdot m) \tag{4.17}$$

Unlike in the previous two cases, we are not able to factor out the $(P-1)$ term because the gap in PLogP is a possibly non-linear function of message size.

4.3.4 Performance models of collective algorithms

We construct performance models for all collective algorithms discussed previously, analogous to the approaches discussed in Sections 4.3.1, 4.3.2, and 4.3.3. Tables 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, and 4.11, show formulas for barrier, broadcast, scatter, gather, allgather, alltoall, reduce, reduce_scatter, allreduce, and scan collectives, respectively. If applicable, the displayed formulas account for message segmentation. Message segmentation allows us to divide a message of size m into a number of segments, n_s, of the specified segment size m_s. In the PLogP model, parameter values depend on the message size. The LogP formulas can be obtained from LogGP by setting the gap per byte parameter G to zero. The specified tables also provide references to relevant and similar work done by other groups. In scatter, gather, allgather, alltoall, and reduce_scatter formulas, m refers to individual block size, meaning that all data on a process is $P \cdot m$. In these models, the formulas are devised for non-vector types of operations: each of the processes is contributing the same amount of data, m.

The difference between recursive doubling formulas for allgather and allreduce in Tables 4.6 and 4.10 comes from the way we define the amount of data per process in our models. In both cases, each node contributes m data, which in case of allgather means that the total data size is $P \cdot m$, while for allreduce, the total data size is still m.

The model of the flat-tree barrier algorithm performance in Table 4.2 requires additional explanation. The conservative model of the flat-tree barrier algorithm would include time to receive (P-1) messages sent in parallel to the same node, and the time to send (P-1) messages from the root. In the first phase, the root process posts (P-1) non-blocking receives followed by a single waitall call. Our experiments show that, on our systems, all MPI implementations we examined were able to deliver (P-1) zero-byte messages sent in parallel to the root close to the time to deliver a single message. Thus, we model the total duration of this algorithm as the time it takes to receive a single zero-byte message plus the time to send (P-1) zero-byte messages.

Barrier	**Model**	**Duration**	**Related work**
Flat-Tree	*Hockney*	$T = (P-1) \cdot \alpha$	
Flat-Tree	*LogP / LogGP*	$T_{min} = (P-2) \cdot g + 2 \cdot (L + 2 \cdot o)$ $T_{max} = (P-2) \cdot (g+o) + 2 \cdot (L + 2 \cdot o)$	
Flat-Tree	*PLogP*	$T_{min} = P \cdot g + 2 \cdot L$ $T_{max} = P \cdot (g + o_r) + 2 \cdot (L - o_r)$	
Double Ring	*Hockney*	$T = 2 \cdot P \cdot \alpha$	
Double Ring	*LogP / LogGP*	$T = 2 \cdot P \cdot (L + o + g)$	
Double Ring	*PLogP*	$T = 2 \cdot P \cdot (L + g)$	
Recursive Doubling	*Hockney*	$T = \left\{ \begin{array}{ll} \log_2(P) \cdot \alpha, & \text{if P is an exact power of 2} \\ (\log_2(P) + 2) \cdot \alpha, & \text{otherwise} \end{array} \right\}$	[Thakur et al., 2005]
Recursive Doubling	*LogP / LogGP*	$T = \left\{ \begin{array}{ll} \log_2(P) \cdot (L + o + g), & \text{if P is exact power of 2} \\ (\lfloor \log_2(P) \rfloor + 2) \cdot (L + o + g), & \text{otherwise} \end{array} \right\}$	
Recursive Doubling	*PLogP*	$T = \left\{ \begin{array}{ll} \log_2(P) \cdot (L + g), & \text{if P is exact power of 2} \\ (\lfloor \log_2(P) \rfloor + 2) \cdot (L + g), & \text{otherwise} \end{array} \right\}$	
Bruck	*Hockney*	$T = \lceil \log_2(P) \rceil \cdot \alpha$	[Thakur et al., 2005]
Bruck	*LogP / LogGP*	$T = \lceil \log_2(P) \rceil \cdot (L + o + g)$	
Bruck	*PLogP*	$T = \lceil \log_2(P) \rceil \cdot (L + g)$	

Table 4.2: Analysis of different barrier algorithms.

Broadcast	**Model**	**Duration**	**Related work**
Linear	*Hockney*	$T = \quad n_s \cdot (P-1) \cdot (\alpha + \beta \cdot m_s)$	[Thakur et al., 2005], [Chan et al., 2004]
Linear	*LogP / LogGP*	$T = \quad L + 2 \cdot o - g + n_s \cdot (P-1) \cdot ((m_s - 1) \cdot G + g)$	
Linear	*PLogP*	$T = \quad L + n_s \cdot (P-1) \cdot g(m_s)$	[Kielmann et al., 2001], [Barchet-Estefanel and Mounié, 2004]
Pipeline	*Hockney*	$T = \quad (P + n_s - 2) \cdot (\alpha + \beta \cdot m_s)$	
Pipeline	*LogP / LogGP*	$T = \quad (P-1) \cdot (L + 2 \cdot o + (m_s - 1) \cdot G) + (n_s - 1) \cdot (g + (m_s - 1) \cdot G + o)$	
Pipeline	*PLogP*	$T = \quad (P-1) \cdot (L + g(m_s)) + (n_s - 1) \cdot g(m_s)$	[Barchet-Estefanel and Mounié, 2004]
Binomial	*Hockney*	$T = \quad \lceil log_2(P) \rceil \cdot n_s \cdot (\alpha + \beta \cdot m_s)$	[Thakur et al., 2005], [Chan et al., 2004]
Binomial	*LogP / LogGP*	$T = \lceil log_2(P) \rceil \cdot \left(\begin{array}{l} L + 2 \cdot o + (m_s - 1) \cdot G + \\ (n_s - 1) \cdot (g + (m_s - 1) \cdot G) \end{array} \right)$	[Culler et al., 1993], [Alexandrov et al., 1995]
Binomial	*PLogP*	$T = \quad \lceil log_2(P) \rceil \cdot (L + n_s \cdot g(m_s))$	[Kielmann et al., 2001], [Barchet-Estefanel and Mounié, 2004]
Binary	*Hockney*	$T = \quad 2 \cdot (\lceil log_2(P+1) \rceil + n_s - 2) \cdot (\alpha + \beta \cdot m_s)$	
Binary	*LogP / LogGP*	$T = \quad (\lceil log_2(P+1) \rceil - 1) \cdot (L + g + 2 \cdot (o + (m_s - 1) \cdot G)) + (n_s - 1) \cdot (o + 2 \cdot (g + (m_s - 1) \cdot G))$	[Culler et al., 1993], [Alexandrov et al., 1995]
Binary	*PLogP*	$T = \quad (\lceil log_2(P+1) \rceil - 1) \cdot (L + 2 \cdot g(m_s)) + (n_s - 1) \cdot \max\{2 \cdot g(m_s),\ o_r(m_s) + g(m_s) + o_s(m_s)\}$	[Kielmann et al., 2001], [Barchet-Estefanel and Mounié, 2004]
Split-binary	*Hockney*	$T = \quad 2 \cdot (\lceil log_2(P+1) \rceil + \lceil \frac{n_s}{2} \rceil - 2) \cdot (\alpha + \beta \cdot m_s) \ + \ \alpha + \beta \cdot \frac{m}{2}$	
Split-binary	*LogP / LogGP*	$T = \quad (\lceil log_2(P+1) \rceil - 1) \cdot (L + g + 2 \cdot (o + (m_s - 1) \cdot G)) \ + (\lceil \frac{n_s}{2} \rceil - 1) \cdot (o + 2 \cdot (g + (m_s - 1) \cdot G)) \ + L + 2 \cdot o + (\frac{m}{2} - 1) \cdot G$	
Split-binary	*PLogP*	$T = \quad (\lceil log_2(P+1) \rceil - 1) \cdot (L + 2 \cdot g(m_s)) \ + (\lceil \frac{n_s}{2} \rceil - 1) \cdot \max\{2 \cdot g(m_s),\ o_r(m_s) + g(m_s) + o_s(m_s)\} + L + g(\frac{m}{2})$	

Table 4.3: Analysis of different broadcast algorithms.

Scatter	**Model**	**Duration**	**Related work**
Linear	*Hockney*	$T = (P-1)(\alpha + \beta \cdot m)$	[Thakur et al., 2005], [Chan et al., 2004]
Linear	*LogP / LogGP*	$T = L + 2 \cdot o + (P-2) \cdot ((m-1) \cdot G + g)$	
Linear	*PLogP*	$T = L + (P-1) \cdot g(m)$	[Kielmann et al., 2001], [Barchet-Estefanel and Mounié, 2004]
Binomial	*Hockney*	$T = \log_2(P) \cdot \alpha + (P-1) \cdot \beta \cdot m$	
Binomial	*LogP / LogGP*	$T = L + 2 \cdot o + (m \cdot (P-1) - \lceil \log_2(P) \rceil) \cdot G + (\lceil \log_2(P) \rceil - 1) \cdot \max\{L + 2 \cdot o,\ g\}$	
Binomial	*PLogP*	$T = \sum_{k=0}^{\lfloor \log_2(P) \rfloor - 1} (L + g(2^k \cdot m))$	[Barchet-Estefanel and Mounié, 2004]

Table 4.4: Analysis of different scatter algorithms.

Gather	**Model**	**Duration**	**Related work**
Linear	*Hockney*	$T = (P-1) \cdot (\alpha + \beta \cdot m)$	[Thakur et al., 2005], [Chan et al., 2004]
Linear	*LogP / LogGP*	$T = L + 2 \cdot o + (P-2) \cdot ((m-1) \cdot G + g)$	
Linear	*PLogP*	$T = L + (P-1) \cdot g(m)$	[Kielmann et al., 2001], [Barchet-Estefanel and Mounié, 2004]
Linear with sync.	*Hockney*	$T = \alpha + (P-1) \cdot (2 \cdot \alpha + \beta \cdot m)$	
Linear with sync.	*LogP / LogGP*	$T = \left\{ \begin{array}{l} (P-1) \cdot (2 \cdot L + 4 \cdot o + (m_s - 1) \cdot G) + g + (m - m_s - 1) \cdot G, \\ \qquad \text{if } g + m \cdot G < L + 2 \cdot (o + m_s \cdot G) \\ (P-1) \cdot (o + g + (m-2) \cdot G + L + o), \quad \text{otherwise} \end{array} \right\}$	
Linear with sync.	*PLogP*	$T = \left\{ \begin{array}{l} (P-1) \cdot (2 \cdot L + g(m_s) + g(0)) + g(m - m_s), \\ \qquad \text{if } g(m - m_s) < L + g(0) \\ 2 \cdot L + g(0) + (P-1) \cdot (g(m_s) + g(m - m_s)), \quad \text{otherwise} \end{array} \right\}$	
Binomial	*Hockney*	$T = \log_2(P) \cdot \alpha + P \cdot \beta \cdot m$	
Binomial	*LogP / LogGP*	$T = (\lfloor \log_2(P) \rfloor - 1) \cdot (L + 2 \cdot o) + ((2 \cdot P - 1) \cdot m - \lfloor \log_2(P) \rfloor) \cdot G$	
Binomial	*PLogP*	$T = \sum_{k=0}^{\lfloor \log_2(P) \rfloor - 1} (L + g(2^k \cdot m))$	[Barchet-Estefanel and Mounié, 2004]

Table 4.5: Analysis of different gather algorithms.

Allgather	**Model**	**Duration**	**Related work**
Bruck	*Hockney*	$T = \log_2(P) \cdot \alpha + P \cdot \beta \cdot m + P \cdot \delta \cdot m$	[Thakur et al., 2005], [Chan et al., 2004]
Bruck	*LogP / LogGP*	$T = \lfloor \log_2(P) \rfloor \cdot (o + \max\{\, g,\ L + o \,\} - G) + (P-1) \cdot m \cdot G + P \cdot \delta \cdot m$	
Bruck	*PLogP*	$T = \sum_{k=0}^{\lfloor \log_2(P) \rfloor - 1} (L + g(2^k \cdot m)) + L + g((P - \sum_{k=0}^{\lfloor \log_2(P) \rfloor - 1} 2^k) \cdot m) + P \cdot \delta \cdot m$	
Recursive doubling	*Hockney*	$T = \log_2(P) \cdot \alpha + (P-1) \cdot \beta \cdot m$	[Thakur et al., 2005], [Chan et al., 2004]
Recursive doubling	*LogP / LogGP*	$T = \log_2(P) \cdot (o + \max\{\, g,\ L + o \,\} - G) + (P-1) \cdot m \cdot G$	
Recursive doubling	*PLogP*	$T = \sum_{k=0}^{\log_2(P) - 1} (L + g(2^k \cdot m))$	
Ring	*Hockney*	$T = (P-1) \cdot (\alpha + \beta \cdot m)$	[Thakur et al., 2005], [Chan et al., 2004]
Ring	*LogP / LogGP*	$T = (P-1) \cdot (L + 2 \cdot o + (m-1) \cdot G)$	
Ring	*PLogP*	$T = (P-1) \cdot (L + g(m))$	
Neighbor exchange	*Hockney*	$T = \alpha + \beta \cdot m + (\frac{P}{2} - 1) \cdot (\alpha + \beta \cdot 2 \cdot m)$	[Chen et al., 2005]
Neighbor exchange	*LogP / LogGP*	$T = \frac{P}{2} \cdot (L + 2 \cdot o + (2 \cdot m - 1) \cdot G) - m \cdot G$	
Neighbor exchange	*PLogP*	$T = L + g(m) + (\frac{P}{2} - 1) \cdot g(2 \cdot m)$	

Table 4.6: Analysis of different allgather algorithms.

Alltoall	**Model**	**Duration**	**Related work**
Linear	*Hockney*	$T = (P-1) \cdot (\alpha + \beta \cdot m)$	[Thakur et al., 2005]
Linear	*LogP / LogGP*	$T = L + 2 \cdot o + (m-1) \cdot G + 2 \cdot (P-1) \cdot g$	[Culler et al., 1993]
Linear	*PLogP*	$T = L + 2 \cdot (P-1) \cdot g(m)$	
Pairwise exchange	*Hockney*	$T = (P-1) \cdot (\alpha + \beta \cdot m)$	[Thakur et al., 2005]
Pairwise exchange	*LogP / LogGP*	$T = (P-1) \cdot (L + o + (m-1) \cdot G + g)$	
Pairwise exchange	*PLogP*	$T = (P-1) \cdot (L + g(m))$	
Bruck	*Hockney*	$T = \begin{array}{l} \lceil \log_2(P) \rceil \cdot \alpha + \\ \lfloor \log_2(P) \rfloor \cdot (\beta + \delta) \cdot \frac{P}{2} \cdot m + \delta \cdot P \cdot m + \\ (\beta + \delta) \cdot (P - 2^{\lfloor \log_2(P) \rfloor}) \cdot m \end{array}$	[Thakur et al., 2005], [Bruck et al., 1997]
Bruck	*LogP / LogGP*	$T = \left\{ \begin{array}{l} \log_2(P) \cdot (o + (\frac{P}{2} \cdot m - 1) \cdot G + \delta \cdot \frac{P}{2} \cdot m + \max\{g,\ L + o\}) + \delta \cdot P \cdot m, \\ \text{if } P = 2^k \\ \lfloor \log_2(P) \rfloor \cdot (o + (\frac{P}{2} \cdot m - 1) \cdot G + \delta \cdot \frac{P}{2} \cdot m + \max\{g,\ L + o\}) + \delta \cdot P \cdot m + \\ o + ((P - 2^{\lfloor \log_2(P) \rfloor}) \cdot m - 1) \cdot G + \delta \cdot (P - 2^{\lfloor \log_2(P) \rfloor}) \cdot m + \max\{g,\ L + o\} \\ \text{otherwise} \end{array} \right\}$	
Bruck	*PLogP*	$T = \begin{array}{l} \lceil \log_2(P) \rceil \cdot L + \\ \lfloor \log_2(P) \rfloor \cdot (g(\frac{P}{2} \cdot m) + \delta \cdot \frac{P}{2} \cdot m) + \delta \cdot P \cdot m + \\ g((P - 2^{\lfloor \log_2(P) \rfloor}) \cdot m) + \delta \cdot (P - 2^{\lfloor \log_2(P) \rfloor}) \cdot m \end{array}$	

Table 4.7: Analysis of different alltoall algorithms.

Reduce	**Model**	**Duration**	**Related work**
Flat Tree	*Hockney*	$T = \; n_s \cdot (P-1) \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s)$	[Thakur et al., 2005], [Chan et al., 2004]
Flat Tree	*LogP / LogGP*	$T = \; \begin{array}{l} o + (m_s - 1) \cdot G + L + \\ n_s \cdot \max \left\{ \begin{array}{l} g, \\ (P-1) \cdot (o + (m_s - 1) \cdot G + \gamma m_s) \end{array} \right\} \end{array}$	
Flat Tree	*PLogP*	$T = \; L + (P-1) \cdot n_s \cdot \max\{g(m_s),\, o_r(m_s) + \gamma m_s\}$	[Kielmann et al., 2001]
Pipeline	*Hockney*	$T = \; (P + n_s - 2) \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s)$	
Pipeline	*LogP / LogGP*	$T = \; \begin{array}{l} (P-1) \cdot (L + 2 \cdot o + (m_s - 1) \cdot G + \gamma m_s) + \\ (n_s - 1) \cdot \max\{g,\, 2 \cdot o + (m_s - 1) \cdot G + \gamma m_s\} \end{array}$	
Pipeline	*PLogP*	$T = \; \begin{array}{l} (P-1) \cdot (L + \max\{g(m_s),\, o_r(m_s) + \gamma m_s\}) + \\ (n_s - 1) \cdot \left(\; \max\{g(m_a),\, o_r(m_s) + \gamma m_s\} + o_s(m_s) \; \right) \end{array}$	
Binomial	*Hockney*	$T = \; n_s \cdot \lceil log_2(P) \rceil \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s)$	[Thakur et al., 2005], [Chan et al., 2004]
Binomial	*LogP / LogGP*	$T = \; \lceil log_2 P \rceil \cdot \left(\begin{array}{l} o + L + n_s \cdot ((m_s - 1) \cdot G + \\ \max\{g,\, o + \gamma m_s\}) \end{array} \right)$	[Culler et al., 1993], [Alexandrov et al., 1995]
Binomial	*PLogP*	$T = \; \lceil log_2 P \rceil \cdot \left(\; L + n_s \cdot \max\left\{ \; g(m_s),\, o_r(m_s) + \gamma m_s + o_s(m_s) \; \right\} \; \right)$	
Binary	*Hockney*	$T = \; 2 \cdot (\lceil \log_2(P+1) \rceil + n_s - 2) \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_S)$	[Thakur et al., 2005], [Chan et al., 2004]
Binary	*LogP / LogGP*	$T = \; \begin{array}{l} (\lceil log_2(P+1) \rceil - 1) \cdot \\ \left(\begin{array}{l} L + 3 \cdot o + (m_s - 1) \cdot G + 2\gamma m_s + \\ (n_s - 1) \cdot \left(\begin{array}{l} (m_s - 1) \cdot G + \\ \max\{g,\, 3o + 2 \cdot \gamma m_s\} \end{array} \right) \end{array} \right) \end{array}$	[Culler et al., 1993], [Alexandrov et al., 1995]
Binary	*PLogP*	$T = \; \begin{array}{l} (\lceil log_2(P+1) \rceil - 1) \cdot \left(\; L + 2 \cdot \max\left\{ \; g(m_s),\, o_r(m_s) + \gamma m_s \; \right\} \; \right) + \\ (n_s - 1) \cdot \left(\; o_s(m_s) + 2 \cdot \max\left\{ \; g(m_s),\, o_r(m_s) + \gamma m_s \; \right\} \; \right) \end{array}$	
Raben-seifner	*Hockney*	$T = \; 2 \cdot \log_2(P) \cdot \alpha + 2 \cdot (P-1) \cdot \beta \cdot m + (P-1) \cdot \gamma \cdot m$	[Thakur et al., 2005], [Rabenseifner, 2004]
Raben-seifner	*LogP / LogGP*	$T = \; 2 \cdot \log_2(P) \cdot (L + 2 \cdot o) + 2 \cdot ((P-1) \cdot m - \log_2(P)) \cdot G + (P-1) \cdot \gamma \cdot m$	
Raben-seifner	*PLogP*	$T = \; 2 \cdot \log_2(P) \cdot L + (P-1) \cdot \gamma \cdot m + \sum_{k=1}^{\log_2(P)} g(\frac{m}{2^k})$	

Table 4.8: Analysis of different reduce algorithms.

Reduce-scatter	**Model**	**Duration**	**Related work**
Recursive halving	*Hockney*	$T = \left\{ \begin{array}{l} \log_2(P) \cdot \alpha + (P-1) \cdot (\beta + \gamma) \cdot m, \\ \qquad \text{if } P \text{ is an exact power of 2} \\ (\lfloor \log_2(P) \rfloor + 2) \cdot \alpha + 2 \cdot P \cdot \beta \cdot m + \\ (2 \cdot P - 1) \cdot \gamma \cdot m \\ \qquad \text{if } P \text{ is not an exact power of 2} \end{array} \right\}$	[Thakur et al., 2005]
Recursive halving	*LogP / LogGP*	$T = \left\{ \begin{array}{l} \log_2(P) \cdot (L + 2 \cdot o) + ((P-1) \cdot m - \log_2(P)) \cdot G + (P-1) \cdot \gamma \cdot m, \\ \qquad \text{if } P \text{ is an exact power of 2} \\ (\lfloor \log_2(P) \rfloor + 2)(L+2) + (2 \cdot (P \cdot m - 1) - \lfloor \log_2(P) \rfloor) \cdot G + (2 \cdot P - 1) \cdot \gamma \cdot m, \\ \qquad \text{if } P \text{ is not an exact power of 2} \end{array} \right\}$	
Recursive halving	*PLogP*	$T = \left\{ \begin{array}{l} \log_2(P) \cdot L + (P-1) \cdot \gamma \cdot m + \sum_{k=1}^{\log_2(P)} g(\frac{P}{2^k}), \quad \text{if } P \text{ is an exact power of 2} \\ (\lfloor \log_2(P) \rfloor + 2) \cdot L + g(m) + (2P-1) \cdot \gamma \cdot m + \sum_{k=0}^{\log_2(P)} g(\frac{P}{2^k}), \\ \qquad \text{if } P \text{ is not an exact power of 2} \end{array} \right\}$	
Ring	*Hockney*	$T = \ (P-1) \cdot (\alpha + \beta \cdot m + \gamma \cdot m) + P \cdot \delta \cdot m$	
Ring	*LogP / LogGP*	$T = \ (P-1) \cdot (L + 2 \cdot o + (m-1) \cdot G + \gamma \cdot m) + P \cdot \delta \cdot m$	
Ring	*PLogP*	$T = \ (P-1) \cdot (L + g(m) + \gamma \cdot m) + P \cdot \delta \cdot m$	

Table 4.9: Analysis of different reduce_scatter algorithms.

Allreduce	**Model**	**Duration**	**Related work**
Recursive doubling	*Hockney*	$T = \begin{Bmatrix} \log_2(P)\cdot(\alpha+\beta\cdot m+\gamma\cdot m), \\ \text{if } P \text{ is an exact power of 2} \\ (\lfloor\log_2(P)\rfloor+2)\cdot(\alpha+\beta\cdot m+\gamma\cdot m)-\gamma\cdot m, \\ \text{if } P \text{ is not an exact power of 2} \end{Bmatrix}$	[Thakur et al., 2005]
Recursive doubling	*LogP / LogGP*	$T = \begin{Bmatrix} \log_2(P)\cdot(L+2\cdot o+(m-1)\cdot G+\gamma\cdot m), \\ \text{if } P \text{ is an exact power of 2} \\ (\lfloor\log_2(P)\rfloor+2)\cdot(L+2\cdot o+(m-1)\cdot G+\gamma\cdot m)- \\ \gamma\cdot m, \\ \text{if } P \text{ is not an exact power of 2} \end{Bmatrix}$	
Recursive doubling	*PLogP*	$T = \begin{Bmatrix} \log_2(P)\cdot(L+g(m)+\gamma\cdot m), \\ \text{if } P \text{ is an exact power of 2} \\ (\lfloor\log_2(P)\rfloor+2)\cdot(L+g(m)+\gamma\cdot m)-\gamma\cdot m, \\ \text{if } P \text{ is not an exact power of 2} \end{Bmatrix}$	
Ring	*Hockney*	$T = 2\cdot(P-1)\cdot(\alpha+\beta\cdot\lceil\frac{m}{P}\rceil)+(P-1)\cdot\gamma\cdot\lceil\frac{m}{P}\rceil$	
Ring	*LogP / LogGP*	$T = \begin{Bmatrix} 2\cdot(P-1)\cdot(L+2\cdot o+(\lceil\frac{m}{P}\rceil-1)\cdot G)+ \\ (P-1)\cdot\gamma\cdot\lceil\frac{m}{P}\rceil \end{Bmatrix}$	
Ring	*PLogP*	$T = 2\cdot(P-1)\cdot(L+g(\lceil\frac{m}{P}\rceil))+(P-1)\cdot\gamma\cdot\lceil\frac{m}{P}\rceil)$	
Ring with segment.	*Hockney*	$T = (P+n_s-2)\cdot(\alpha+\beta\cdot m_s+\gamma\cdot m_s)+(P-1)\cdot(\alpha+\beta\cdot\lceil\frac{m}{P}\rceil)$	
Ring with segment.	*LogP / LogGP*	$T = \begin{Bmatrix} (P-1)\cdot(L+2\cdot o+(m_s-1)\cdot G)+ \\ (n_s-1)\cdot(\max\{g,\ (\gamma\cdot m_s+o)\}+(m_s-1)\cdot G)+ \\ (P-1)\cdot(L+2\cdot o+(\lceil\frac{m}{P}\rceil-1)\cdot G) \end{Bmatrix}$	
Ring with segment.	*PLogP*	$T = \begin{Bmatrix} (P-1)\cdot(L+g(m_s)+\gamma\cdot m_s)+(n_s-1)\cdot(g(m_s)+\gamma\cdot m_s)+ \\ (P-1)\cdot(L+g(\lceil\frac{m}{P}\rceil)) \end{Bmatrix}$	
Raben-seifner	*Hockney*	$T = 2\cdot\log_2(P)\cdot\alpha+2\cdot\frac{(P-1)}{P}\cdot\beta\cdot m+\frac{(P-1)}{P}\cdot\gamma\cdot m$	[Thakur et al., 2005]
Raben-seifner	*LogP / LogGP*	$T = \begin{Bmatrix} 2\cdot\log_2(P)\cdot(L+2\cdot o)+ \\ 2\cdot((P-1)\cdot m-\log_2(P))\cdot G+(P-1)\cdot\gamma\cdot\frac{m}{P}+ \\ \log_2(P)\cdot(o+\max\{g,\ L+o\}-G)+ \\ (P-1)\cdot\frac{m}{P}\cdot G \end{Bmatrix}$	
Raben-seifner	*PLogP*	$T = 3\cdot\log_2(P)\cdot L+(P-1)\cdot\gamma\cdot\frac{m}{P}+3\cdot\sum_{k=1}^{\log_2(P)-1} g(\frac{m}{2^k})$	

Table 4.10: Analysis of different allreduce algorithms.

Scan	**Model**	**Duration**	**Related work**
Linear	*Hockney*	$T = \quad (P-1)\cdot(\alpha+\beta\cdot m+\gamma\cdot m)$	
Linear	*LogP / LogGP*	$T = \quad (P-1)\cdot(L+2\cdot o+(m-1)\cdot G+\gamma\cdot m)$	
Linear	*PLogP*	$T = \quad (P-1)(L+g(m)+\gamma\cdot m)$	
Linear with segment.	*Hockney*	$T = \quad (P+n_s-2)\cdot(\alpha+\beta\cdot m_s+\gamma\cdot m_s)$	
Linear with segment.	*LogP / LogGP*	$T = \left\{ \begin{array}{l} (P-1)\cdot(L+2\cdot o+(m_s-1)\cdot G+\gamma\cdot m_s)+ \\ (n_s-1)\cdot(\max\{g,\ 2\cdot o+(m_s-1)\cdot G+\gamma\cdot m_s\}) \end{array} \right\}$	
Linear with segment.	*PLogP*	$T = \left\{ \begin{array}{l} (P-1)(L+g(m_s)+\gamma\cdot m_s)+ \\ (n_s-1)\cdot\max\{g(m_s), o_s(m_s)+o_r(m_s)+\gamma\cdot m_s\} \end{array} \right\}$	
Binomial	*Hockney*	$T = \quad \lceil\log_2(P)\rceil\cdot(\alpha+\beta\cdot m+\gamma\cdot m)$	
Binomial	*LogP / LogGP*	$T =$ $\lceil\log_2(P)\rceil\cdot(L+2\cdot o+(m-1)\cdot G+\max\{g,\ o+\gamma\cdot m\})$	
Binomial	*PLogP*	$T = \quad \lceil\log_2(P)\rceil\cdot(L+g(m)+\max\{g,\ o_s(m)+\gamma\cdot m\})$	

Table 4.11: Analysis of different scan algorithms.

Hockney model		**Grig (mx)**	**Grig (tcp)**	**Boba**	**Frodo (mx)**	**Frodo (tcp)**
Latency, α	$[\mu sec]$	5.64	110.65	47.6	4.23	80.2
Transfer time, β	$[\frac{\mu sec}{byte}]$	0.0042	0.0851	0.0151	0.0042	0.0854
Bandwidth, $\frac{1}{\beta}$	$[\frac{byte}{\mu sec}]$	239.58	12.33	66.36	235.51	11.71
Computation per byte, γ	$[\frac{\mu sec}{byte}]$	5×10^{-4}	5×10^{-4}	1.7×10^{-3}	3×10^{-3}	3×10^{-3}

Table 4.12: Hockney model parameter values on our systems.

4.4 Evaluation of MPI collective operation models

This section provides evaluation of some of the proposed models of MPI collective operations on high-performance systems at the University of Tennessee, Knoxville. The goal of this analysis is to assess the accuracy and limitations of the proposed models in order to reduce the total number of approaches that need to be evaluated in the algorithm selection/decision making section.

The first cluster, Grig, consists of 64 nodes, each equipped with quad Intel®Xeon^{TM}CPUs at 3.20 GHz, 1MB cache per CPU and a total of 4GB RAM per node. The system is interconnected via 100Mbps Fast Ethernet and Myrinet MX version 1.1.5.

The second cluster, Boba, consists of 32 Dell Precision 530s nodes, each with Dual Pentium IV Xeon 2.4 GHz processors, 512 KB Cache, 2 GB memory, connected via Gigabit Ethernet.

The third cluster, Frodo, consists of 32 nodes, each containing dual Opteron processor, 2 GB memory, connected via 100 Mbps Ethernet and Myrinet MX. In the results presented in this section, we did not utilize the Myrinet interconnect on the Frodo cluster.

We measured the model parameters using different MPI implementations, and obtained similar values. On the Grig cluster using Open MPI [*] [Open MPI, 2005], we used the developer version 1.2.4. The parameters from the Boba and Frodo cluster were measured using FT-MPI and MPICH-2.0.97 [FT-MPI, 2003, MPICH2, 2002]. The experimental data from the Grig cluster was collected using the SKaMPI benchmark [SKaMPI, 2005] version 5.0.1. Results from the Boba and Frodo clusters were collected using OCC benchmark [OCC, 2005].

4.4.1 Model parameters

The Hockney model parameters were measured using the NetPIPE benchmark [NetPIPE, 2005]. The selected latency and bandwidth values were absolute minimum and maximum, respectively, achieved by the NetPIPE benchmark. Table 4.12 contains the Hockney model parameter values on our systems.

To measure the PLogP model parameters we used the `logp_mpi` software suite provided by Kielmann et al. [Kielmann et al., 2000]. The selected values were taken from Send-Receive

[*]Open MPI supports two lower-lever interfaces to MX: `PML CM, MTL MX`, and `PML OB1, BTL MX`. The experimental data was collected using the latter, `PMLOB1, BTL MX`.

LogP / LogGP model		**Grig (mx)**	**Grig (tcp)**	**Boba**	**Frodo (mx)**	**Frodo (tcp)**
Latency, L	$[\mu sec]$	2.1	106.6	30.40	1.8	61.22
Overhead, o	$[\mu sec]$	1.6	19.5	8.15	1.3	8.2
Gap, g	$[\mu sec]$	2.3	1	8.683	1.6	23.8
Gap-per-byte, G	$[\frac{\mu sec}{byte}]$	0.0044	0.0859	0.0150	0.0044	0.0840
Computation per byte, γ	$[\frac{\mu sec}{byte}]$	5×10^{-4}	5×10^{-4}	1.7×10^{-3}	3×10^{-3}	3×10^{-3}

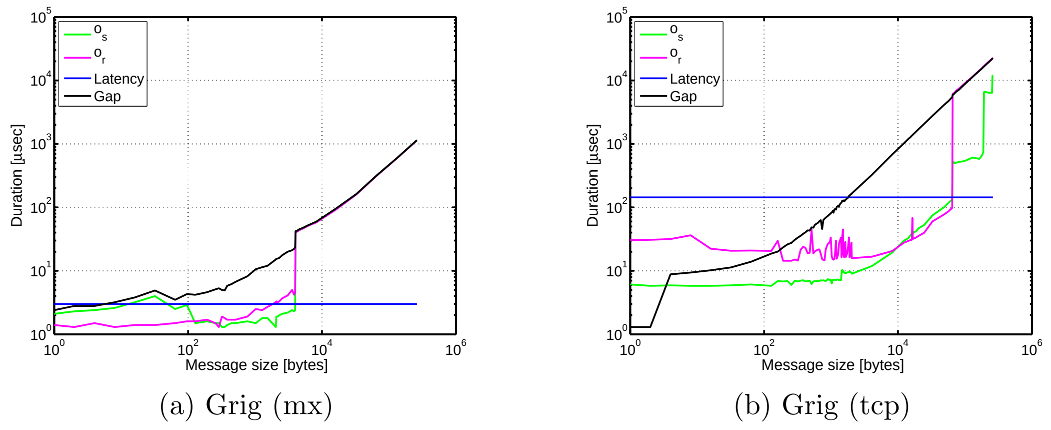
Table 4.13: LogP/LogGP model parameters on our clusters.

measurements. The measured parameter values on Frodo and Boba were obtained by averaging the values obtained between different communication points in the same system. For the PLogP model on these systems, we also experimented with directly fitting model parameters to the experimental data, and applying those parameter values to model other collective operations. Parameter fitting was done under the assumption that the sender and receiver overheads do not depend on the network behavior, and as such we used values measured by the `log_mpi` library. We obtained the values of fitted PLogP parameters by analyzing the performance of the non-segmented pipelined broadcast and flat-tree barrier algorithm over various communicator and message sizes. We chose to fit model parameters to these algorithms as the communication pattern of non-segmented pipelined broadcast's data algorithm (linear sending and receiving message) is the closest match to the point-to-point tests used to measure model parameters in the `logp_mpi` and similar libraries. At the same time, flat-tree barrier formulas in Table 4.2 provide the most direct way of computing the gap per message parameter for zero-byte messages for PLogP and LogP/LogGP models. Results obtained using these values matched more closely the overall experimental data, thus all the PLogP model results in this Section were obtained using fitted parameters. Parameter values alues of the LogP and LogGP on these systems were obtained from the fitted PLogP values as explained by Kielmann et al. in [Kielmann et al., 2000]. Figure 4.19 and Table 4.13 contain values of the PLogP and LogP/LogGP parameters on our systems, respectively.

The computation time per byte was measured using a simple benchmark, which measures the time it takes to perform the operation on a buffer of specified size. The timings were measured using the high-accuracy PAPI timer [PAPI, 2005], and caching effects were avoided by creeping through arrays. Figure 4.20 shows the computation time per element and byte for two common operations: integer sum and product, and double precision sum and product on the Grig cluster. Similar results were obtained from Boba and Frodo. We chose to model the computation time per byte using the steady state value.

4.4.2 Performance of different collective algorithms

We executed performance tests on various algorithms for barrier, broadcast, reduce, and alltoall collective operations using Open MPI, FT-MPI, MPICH-1, and MPICH-2. We then



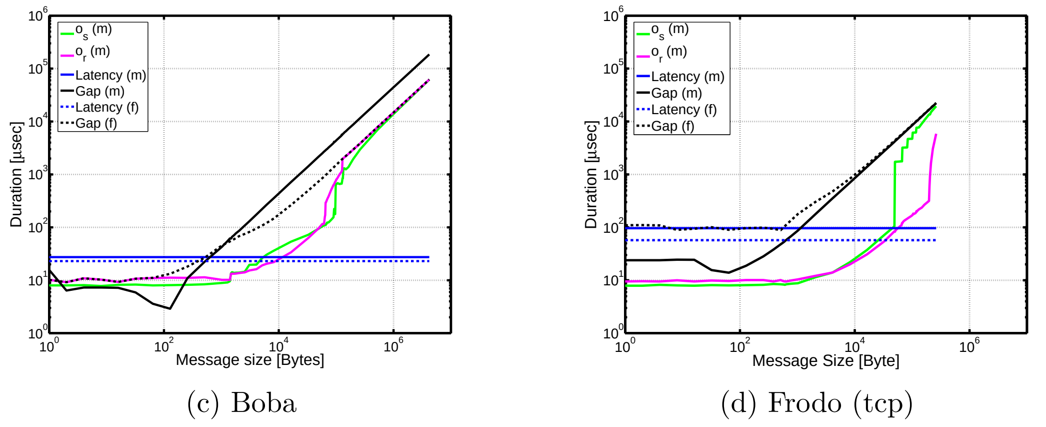


Figure 4.19: PLogP parameter values on our systems: (a) Grig (mx), (b) Grig (tcp), (c) Boba, and (d) Frodo. On Boba and Frodo subfigures (b) and (c), "(m)" denotes measured values, while "(f)" denotes fitted values of gap and latency.

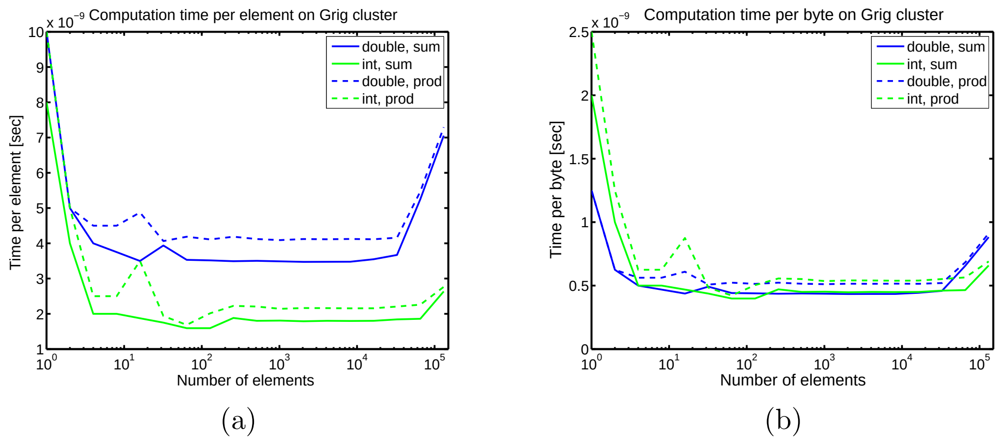
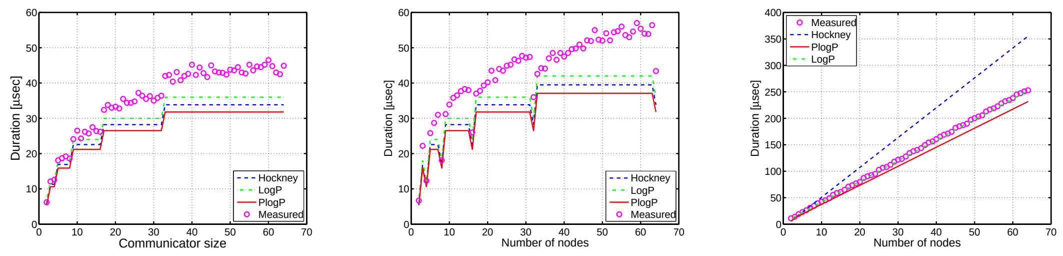


Figure 4.20: Computation time on the Grig cluster: (a) per element and (b) per byte.



(a) Bruck (b) Recursive Doubling (c) Flat-tree

Figure 4.21: Performance of barrier algorithms. Experimentally measured values are indicated by circles. (Open MPI, Grig cluster, MX).

analyzed the algorithm performance of various collective operations using the described parallel communication models, Hockney, LogP/LogGP, and PLogP. When predicting performance of the collective operations that exchanged actual data (message size $>$ 0) we did not consider pure LogP predictions, but used LogGP instead (See Section 4.2.1).

Additional model performance evaluation results are reported in [Pješivac-Grbović et al., 2007a].

Barrier performance

Figure 4.21 illustrates measured and predicted performance of bruck, recursive doubling, and linear fan-in-fan-out/flat-tree barrier algorithms on Grig cluster. The measurements were collected using the SKaMPI benchmark and Open MPI.

The performance of both the bruck and recursive doubling algorithm was underestimated by all models. We believe that this is due to the "full-duplex" assumption. Even though MX is a high-performance network, it seems that assuming that a process can exchange a message at the cost of single receive is optimistic.

The measured data for the flat-tree barrier algorithm exhibits the perfect linear behavior. Based on the PLogP and LogP/LogGP models for the maximum duration (T_{max}) of performance showed in Table 4.2, the duration of this algorithm grows linearly with communicator size, and the slope of the line is equal to the zero-byte gap plus overhead. The LogP model captures the measured slope very well. PLogP is less successful but its predictions are still fairly accurate. The Hockney model overestimates the value grossly. Since the Hockney model assumes that the minimum time between sending two messages is equal to the latency, the prediction for this model for flat-tree barrier is largely overestimated.

However, even accounting for all known discrepancies, the models captured the relative performance of these barrier algorithms sufficiently correctly.

The barrier algorithm performance on the Boba cluster was analyzed in [Pješivac-Grbović et al., 2007a]. In this case, the models were more accurate when we modeled communication using the "best-case" scenario (T_{min} in Table 4.2). In addition, the flat-tree barrier exhibited linear behavior as expected, but the slope changed after 16 processes.

Reduce performance

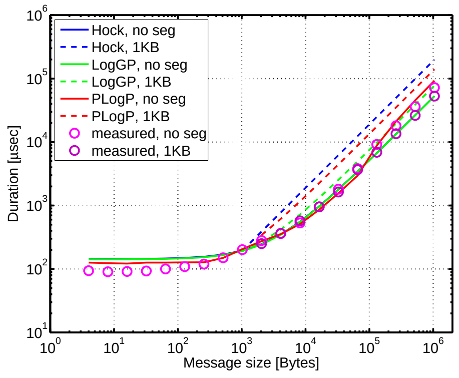
Figure 4.22 displays measured and predicted performance of the non-segmented and segmented versions of binomial and pipeline reduce algorithms for two communicator sizes on the Boba cluster. Results indicate that for small message sizes, the non-segmented binomial algorithm outperforms the pipeline algorithm, while for large message sizes, the segmented pipeline algorithm would have best performance.

Experimental data for non-segmented binomial and pipeline reduce algorithms exhibits a non-linear increase in duration for the message sizes in the range from 1KB to 10KB. The similar increase can be observed for large message sizes ($>$ 100KB) on the non-segmented binomial algorithm. All three models were able to capture the relative performance of non-segmented algorithms in question. However, Hockney and LogP/LogGP failed to capture the non-linear increase in duration for the intermediate sized messages. PLogP was the only model that captured the non-linear behavior of the non-segmented binomial algorithm for both large and intermediate message sizes. We can explain these shortcomings by considering the model parameters. The Hockney and LogP/LogGP models assume linear dependence between the time to send/receive a message and message size. However, the results in Figure 4.22 show that in general, this is certainly not the case. The gap and overhead parameters of the PLogP model are a function of message size, so some of the nonlinear effects can be accounted for. We believe that the non-linear changes in values of sender and receiver overheads (Figure 4.19) enabled PLogP to capture the performance of these methods.

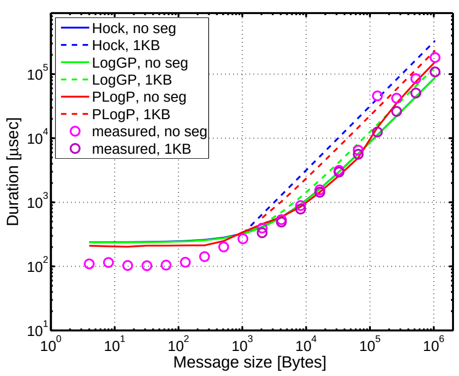
In the experiments in Figure 4.22, segmentation using 1KB segments improved performance of both pipeline and binomial reduce algorithms. While segmentation incurs overhead for managing multiple messages, it also enables higher bandwidth utilization due to an increased number of concurrent messages, it provides an opportunity to overlap multiple communications and computation, and it limits the size of internal buffers required by the algorithm. The models of pipeline reduce in Table 4.8 dictate that as the number of segments, n_s, increases (total message size increases), the algorithm should achieve asymptotically optimal performance. In the asymptotic case, the segmented pipeline reduce algorithm should take a constant amount of time for a message of size m and should not depend on number of processes, P. The results in Figure 4.22 (c) and (d) are consistent with this observation: the duration of the segmented pipeline reduce on 8 and 24 nodes takes around $4 \times 10^4 \mu sec$. All three models correctly captured the relative performance of the segmented pipeline algorithm, and the PLogP model had the best estimate of the absolute duration of the operation.

Modeling performance of the segmented binomial reduce algorithm proved to be a challenge for all three models. Contrary to the measured results, the formulas in Table 4.8 seem to indicate that with an increased the number of segments the duration of the binomial reduce operation should increase, and model predictions in Figure 4.22 agree with that. However, to determine if models are capable of recognizing the benefit of segmentation for the binomial reduce algorithm we have to analyze these formulas in more detail.

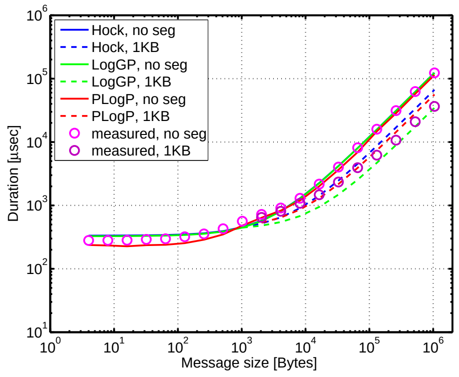
According to the Hockney model of the segmented binomial reduce algorithm, the segmentation should improve operation performance when $n_s \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s) < \alpha + \beta \cdot m + \gamma \cdot m$. However, given that $n_s \geq 1$ and $n_s \cdot m_s = m$, we conclude that this



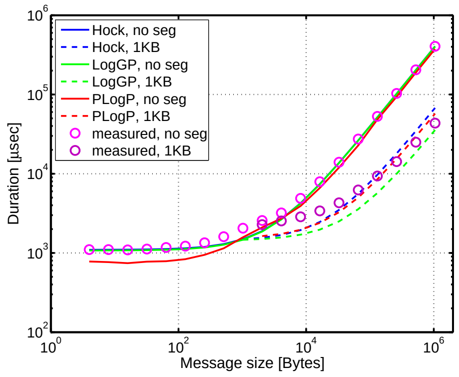
(a) Binomial reduce, 8 nodes



(b) Binomial reduce, 24 nodes



(c) Pipeline reduce, 8 nodes



(d) Pipeline reduce, 24 nodes

Figure 4.22: Performance of segmented binomial and pipelined reduce methods on 8 and 24 nodes. (MPICH-2, Boba cluster, GigE).

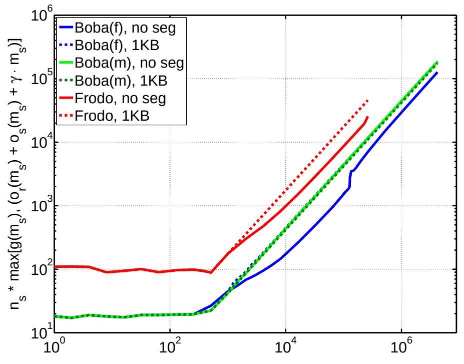


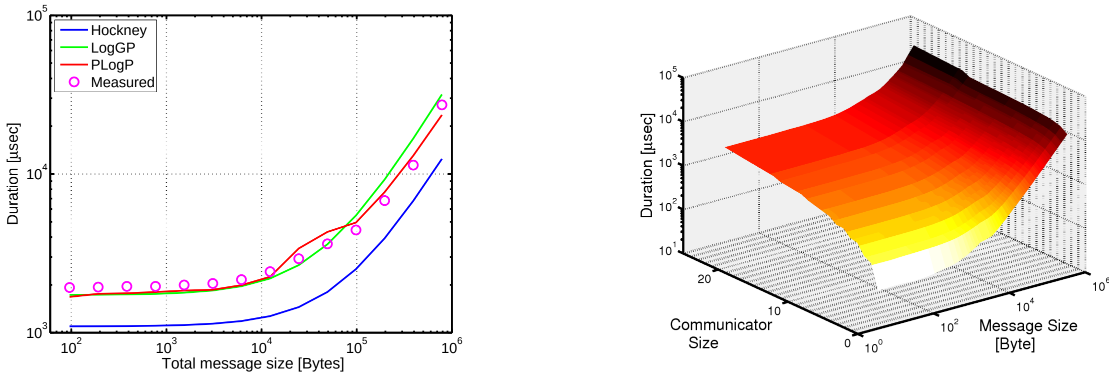
Figure 4.23: Necessary condition for segmentation using 1KB segments of binomial reduce algorithm to improve algorithm performance using PLogP model. In this Figure we use parameters from Figure 4.19 (b) and assume segment size of 1KB.

condition cannot be satisfied. The Hockney model cannot predict that segmentation can improve performance of the binomial reduce algorithm.

In the LogP/LogGP model of the segmented binomial reduce algorithm, the only condition under which segmentation would be beneficial is $n_s \cdot ((m_s - 1) \cdot G + \max\{g,\, o + \gamma \cdot m_s\}) < (m-1) \cdot G + \max\{g,\, o + \gamma \cdot m\}$, or equivalently $(m - n_s) \cdot G + \max\{n_s \cdot g,\, n_s \cdot o + \gamma \cdot m\} < (m-1) \cdot G + \max\{g,\, o + \gamma \cdot m\}$. This condition finally reduces to either $n_s \cdot G + n_s \cdot g < G + g$ or $n_s \cdot G + n_s \cdot o < G + o$. However, by the definition of segmentation, n_s is greater or equal to 1. Thus, under the LogP/LogGP model it is impossible to get a result in which the message segmentation would improve performance of the binomial reduce algorithm.

The PLogP model of the segmented binomial reduce algorithm shows that the following condition is necessary for segmentation to improve performance of this algorithm: $(\max\{g(m), (o_r(m) + o_s(m) + \gamma \cdot m)\}) > \max\{(n_s \cdot g(m_s)), [n_s \cdot (o_r(m_s) + o_s(m_s)) + \gamma \cdot m]\})$. Figure 4.23 illustrates this condition as a function of the message size. In case of fitted parameters on both clusters (See discussion from Section 4.4.1), the non-segmented version of binomial reduce algorithm always outperforms the one with 1KB segments. However, when the directly measured PLogP parameters on the Boba cluster are used to evaluate the condition, the segmented version outperforms the non-segmented version by a slight margin. Thus, using the measured parameters, the PLogP model of the binomial reduce algorithm would capture the segmentation effect correctly.

The analysis of model parameters and the effect of segmentation on binomial reduce show that the models are fairly sensitive to the values of model parameters. As observed in the case of the flat-tree barrier algorithm in [Pješivac-Grbović et al., 2007a], the gap between messages on the Boba cluster depends on the number of nodes we are communicating with, and for communicator sizes greater than 16 nodes, it decreased in comparison to smaller communicator sizes. However, PLogP and LogP/LogGP models cannot include this dependence, and the Hockney model does not even have the notion of gap. Additionally, MPI libraries in this experiment used the TCP/IP stack. The TCP window size on our systems is 128KB. This means that sending messages larger than the TCP window could



(a) (b)

Figure 4.24: Performance of the pairwise exchange alltoall algorithm: (a) Measured performance and predictions for 24 nodes, and (b) Measured performance on 2 to 24 nodes. The message size represents the total send buffer size (FT-MPI, Boba cluster, GigE).

require resizing the window and an extra memory copy operation per pair of communicating parties (which in this case is $log_2(P)$ times). Only the PLogP model considers sender and receiver overheads to depend on message size, LogP/LogGP and Hockney do not have this notion.

Alltoall performance

Figure 4.24 shows the performance of the pairwise-exchange alltoall algorithm. The alltoall type of collectives can cause network flooding even when we attempt to carefully schedule communication between the nodes. The Hockney model does not have the notion of network congestion and this is one of the possible reasons why it significantly underestimates the completion time of the collective operation. While we did not explicitly include a congestion component in the PLogP and LogGP model formulas, they were able to predict measured performance with reasonable accuracy. This indicates that in the test, the communication was scheduled correctly and we did not over-flood the switch.

4.4.3 Final comments about parallel computation models

The performance models of a collective algorithm can give us valuable insight into expected running time of the algorithm. In some cases, such as barrier and pipeline reduce, all models we considered gave satisfying results. However, in the case of binomial reduce, for example, two of the models, Hockney and LogP/LogGP, were not able to capture observed behavior even theoretically. Considering that segmentation is one of the optimization techniques we utilize often, this is a serious impediment.

Based on results reported in this chapter and in [Pješivac-Grbović et al., 2007a], we conclude that the PLogP model is the most suitable for our purposes: determining the best algorithm to run. However, since the parameters of this model are a function of message size, analytical analysis and formula simplification can be harder than in the case of LogP/LogGP and Hockney models.

In the next chapter, we will apply these models to determine optimal implementation of a collective on a system and compare analytical results against experimentally optimal implementation.

Chapter 5

Decision Construction/Algorithm Selection Methods

The goal of this dissertation is an improvement of the performance of MPI collective operations and, implicitly, an improvement of the performance of the applications which use them. To achieve this, we focus on MPI collective method selection process and the automatic, system-specific, run-time decision/selection function generation.

Starting from the available method performance data, which can be obtained either experimentally or using parallel communication models, we employ the following techniques to automatically generate system-specific, run-time, MPI collective method selection functions: performance modeling, graphics encoding schemas, and statistical learning methods (Figure 5.1).

We represent the optimal implementation of a collective on a system using *decision maps*. The decision map specifies the best performing method for the specified communicator, message size pair. Figure 5.2 illustrates a decision map for the reduce operation on the Frodo cluster at the University of Tennessee, Knoxville.

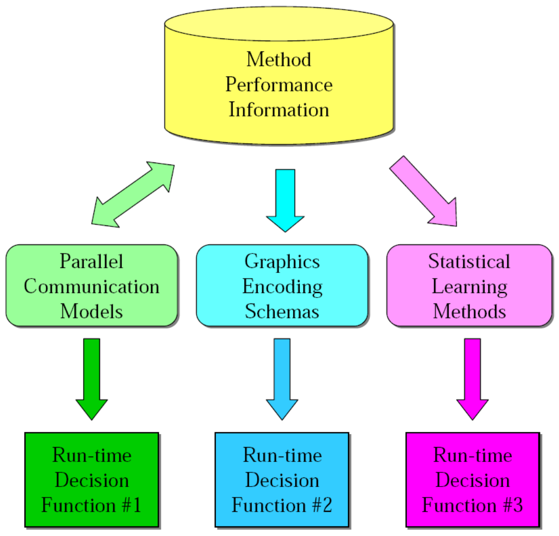
5.1 Formal Problem Statement

Let A^c be a set of all available algorithms for an MPI collective operation c, $A^c = \{a_1^c, a_2^c, ...a_n^c\}$ and let s_y denote a segment size, which can have any meaningful value. The set of available methods for collective c, is M^c: $M^c = \{m_1^c, m_2^c, ...m_k^c\}$, where for all i, $m_i = (a_x, s_y)$. If $p_S(m_i^c, tpin_{ij}^c)$ denotes performance information for method m_i^c on system S and (test) input parameters $tpin_{ij}^c$, the available performance information for the method m_i^c on system S, $P_S^{m_i^c}$, can be defined as:

$$P_S^{m_i^c} = \bigcup_{j=1}^{n_i} p_S(m_i^c, tpin_{ij}^c) \tag{5.1}$$

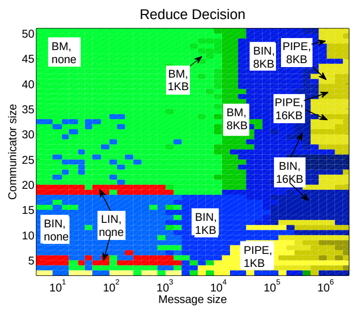
The available performance information on system S for a collective c can be represented as:

$$P_S^c = \bigcup_{i=1}^{k} P_S^{m_i^c} = \bigcup_{i=1}^{k}\bigcup_{j=1}^{n_i} p_S(m_i^c, tpin_{ij}^c). \tag{5.2}$$



Method Performance Information
Parallel Communication Models
Graphics Encoding Schemas
Statistical Learning Methods
Run-time Decision Function #1
Run-time Decision Function #2
Run-time Decision Function #3

Figure 5.1: Approach for decision function construction



(a)

Comm size	**Msg size**	**Algorithm**	**Seg size**	**Method index**
3	1	Linear	none	15
3	2	Linear	none	15
...	...	...	...	...
50	1MB	Pipeline	8KB	24
...	...	...	...	...

(b)

Figure 5.2: **(a)** Reduce decision map from Frodo cluster. Different colors correspond to different method indexes. **(b)** An example of decision map in tabular form.

where k is the total number of methods and n_i is the total number of performance measurements for the method m_i. In general, each of the methods could have been tested on a unique set of input parameters, $tpin^c_{ij}$.

If PIN^c is the set of all possible combination of input parameters for a collective c, we are interested in different approaches to construct relation D which maps the set of input parameters to the set of available methods M^c while minimizing collective operation duration.

$$D : PIN^c \rightarrow M^c \tag{5.3}$$

In our notation, the relation D corresponds to the system-specific decision map of a collective.

Problem Statement

Given a system S, an MPI collective operation c, the set of methods for this collective M^c, the available performance information P^c_S, and the set of all possible collective operation input parameters PIN^c, define the relation D, such that for an instance of an input parameters pin^c, it selects a method, m^c, such that the duration of the collective operation is minimized according to a preselected criteria F^c.

$$D(pin^c) = \{m^c, \qquad \text{such that} \quad F^c(m^c(pin^c)) = \min_i F^c(m^c_i(pin^c))\}. \tag{5.4}$$

The obvious value for the preselected criterion is the absolute duration of a method. The simplest way to define $F^c(m^c(pin^c))$ is as the best performing method for the input pin^c if the performance data exists for that datapoint. In case that the performance data for the particular set of input parameters is missing, $F^c(m^c(pin^c))$ can be estimated based on the available performance information:

$$F^c(m^c(pin^c)) = \left\{ \begin{array}{ll} p^c_S(m^c(pin^c)), & \text{if } p^c_S(m^c(pin^c)) \in P^c_S \\ \Phi(P^c_S), & \text{otherwise} \end{array} \right\} \tag{5.5}$$

Alternatively, we can consider selecting a method other than the one that achieved the best performance for the particular set of input parameters. It is possible that there was a noise in the experimental data, and the optimal method performed worse for the particular data point. In addition, the difference between the two best methods can be within the measurement error. This approximation results in less accurate, but possibly less complicated and more stable method selection. Thus, the following more relaxed definition of $F^c(m^c(pin^c))$ is also useful:

$$F^c(m^c(pin^c)) = \Phi(P^c_S) \tag{5.6}$$

where Φ is a function of the available performance data. This dissertation proposes, implements, analyzes, and evaluates different ways of defining F^c.

As we consider one collective at the time, marker c can be removed from the notation.

5.2 Analytical methods: Parallel communication models

Given performance models of MPI collective algorithms, point-to-point model parameters for the system of interest, and the collective operation parameters, one can predict the completion time of an instance of the collective operation. By comparing completion times of the available collective methods, we can determine the best available method. This section discusses the analytical approach to the optimal collective method selection problem.

5.2.1 Predicting the collective algorithm performance

Section 4.3 introduces a number of different performance models for MPI collective operations. Combined with the system parameters and the operation input parameters, these models can be used to predict the completion time for each of the described algorithms.

In the case of Hockney- and LogP/LogGP-based models, the process includes evaluation of an appropriate formula from the Tables 4.2 through 4.11. PLogP-based collective operation models may require an additional step during the evaluation. PLogP parameters, gap and send and receive overheads, are message-size dependant. If either of the parameter values is missing for the particular message size, its value needs to be approximated. Linear interpolation using neighboring points is one way to find an approximate value for this parameter.

Some of the algorithms of interest support explicit message segmentation. The segment size to be used can be either predetermined or selected such that it minimizes the predicted duration of the method. The predetermined segment size can be useful when we want to restrict the resources (such as the internal buffers) the algorithm uses. If this is not the case, we should utilize the segment size that minimizes the duration of the collective.

5.2.2 Computing the optimal segment size

The optimal segment size, m_s for the algorithm a, given the algorithm performance model $T_a(pin, m_s)$ and input parameters $pin = (P, m,\ \text{etc.})$, can be computed as

$$\frac{\partial T_a(pin,s)}{\partial m_s} = 0 \quad \text{, where } m = n_s \cdot m_s \tag{5.7}$$

Equation 5.7 can be used to find an analytical formula for the optimal segment size for Hockney- and LogP/LogGP-based performance models. For PLogP-based performance models the partial derivative can be computed by definition based on measured parameter values. Thus, if the data points in which the gap and overheads are evaluated are far apart, a search through probable segment size values is the most appropriate approach.

In what follows, we compute the optimal segment size in the case of the split-binary broadcast algorithm, and provide analytical formulas for the remaining segmented algorithms from the Section 4.3. We assume that an optimal segment size can be any real number, even though the only physically meaningful segment sizes have integer values. In addition, we assume that the functions describing models are smooth.

Theorem 5.1 (Optimal segment size for split-binary broadcast, Hockney)**.** *The optimal segment size for the split-binary broadcast algorithm according to Hockney model is* $m_s = \sqrt{\frac{\alpha \cdot m}{2 \cdot \beta \cdot (\lceil \log_2(P+1) \rceil - 2)}}$.

Proof. Starting from the Hockney-based performance model for the split-binary broadcast from Table 4.3 and Equation 5.7 we compute the optimal segment size m_s using Fermat's theorem and the Second derivative test [Stewart, 1995].

$$\begin{aligned} T &= 2 \cdot (\lceil log_2(P+1) \rceil + \lceil \tfrac{n_s}{2} \rceil - 2) \cdot (\alpha + \beta \cdot m_s) \ + \ \alpha + \beta \cdot \tfrac{m}{2} \\ &= 2 \cdot (\lceil log_2(P+1) \rceil + \lceil \tfrac{m}{2 \cdot m_s} \rceil - 2) \cdot (\alpha + \beta \cdot m_s) \ + \ \alpha + \beta \cdot \tfrac{m}{2} \\ &= 2 \cdot \alpha \lceil log_2(P+1) \rceil + \alpha \lceil \tfrac{m}{m_s} \rceil - 4 \cdot \alpha + \\ &\quad 2 \cdot \beta \lceil log_2(P+1) \rceil \cdot m_s - 4 \cdot \beta \cdot m_s + \\ &\quad \beta \cdot m + \alpha + \beta \cdot \tfrac{m}{2} \end{aligned}$$

$$\begin{aligned} \tfrac{\partial T}{\partial m_s} &= 0 + \alpha \cdot m \cdot (-\tfrac{1}{m_s^2}) - 0 + 2 \cdot \beta \lceil log_2(P+1) \rceil - 4 \cdot \beta + 0 \\ &= \alpha \cdot m \cdot (-\tfrac{1}{m_s^2}) + 2 \cdot \beta (\lceil log_2(P+1) \rceil - 2) \end{aligned}$$

Fermat's theorem states that at critical points, $\frac{\partial T}{\partial m_s} = 0$. Rearranging the last equation above and considering only a positive root, we obtain

$$m_s = \sqrt{\frac{\alpha \cdot m}{2 \cdot \beta \cdot (\lceil \log_2(P+1) \rceil - 2)}} \tag{5.8}$$

Now, we need to show that the value of m_s in Equation 5.8 achieves the minimum. Sufficient condition for this is that the second derivative $\frac{\partial^2 T}{\partial m_s^2} > 0$.

$$\begin{aligned} \tfrac{\partial^2 T}{\partial m_s^2} &= \frac{\partial \Big(\alpha \cdot m \cdot (-\frac{1}{m_s^2}) + 2 \cdot \beta (\lceil log_2(P+1) \rceil - 2) \Big)}{\partial m_s} \\ &= \tfrac{2 \cdot \alpha \cdot m}{m_s^3} \end{aligned} \tag{5.9}$$

As all the variables in Equation 5.9 are positive real numbers, we have that

$$\frac{\alpha \cdot m}{m_s^3} > 0$$

which is sufficient to prove that the segment size m_s from the equation minimizes the algorithm duration. □

Theorem 5.2 (Optimal segment size for split-binary broadcast, LogP/LogGP)**.** *The optimal segment size for the split-binary broadcast algorithm according to LogP/LogGP models is* $m_s = \sqrt{\frac{m \cdot (o + 2 \cdot g - 2 \cdot G)}{4 \cdot (\lceil \log_2(P+1) \rceil - 2) \cdot G}}$ *for all* $P > 2$.

Proof. Starting from the LogP/LogGP-based performance model for the split-binary broadcast from Table 4.3 and Equation 5.7, we compute the optimal segment size m_s using Fermat's theorem and the Second derivative test [Stewart, 1995].

$$\begin{aligned} T &= (\lceil log_2(P+1) \rceil - 1) \cdot (L + g + 2 \cdot (o + (m_s - 1) \cdot G)) + \\ &\quad (\lceil \tfrac{m}{2 \cdot m_s} \rceil - 1) \cdot (o + 2 \cdot (g + (m_s - 1) \cdot G) + \\ &\quad L + 2 \cdot o + (\tfrac{m}{2} - 1) \cdot G \\ &= (\lceil log_2(P+1) \rceil - 1) \cdot (L + g + 2 \cdot (o + (m_s - 1) \cdot G)) + \\ &\quad \lceil \tfrac{m}{2 \cdot m_s} \rceil \cdot (o + 2 \cdot g - 2 \cdot G) + m \cdot G \ - \ (o + 2 \cdot g + 2 \cdot (m_s - 1) \cdot G) + \\ &\quad L + 2 \cdot o + (\tfrac{m}{2} - 1) \cdot G \end{aligned}$$

$$\begin{aligned}\frac{\partial T}{\partial m_s} &= (\lceil log_2(P+1)\rceil - 1)\cdot 2\cdot G - \frac{m}{2\cdot m_s^2}\cdot(o+2\cdot g-2\cdot G) - 2\cdot G\\ & 2\cdot(\lceil log_2(P+1)\rceil - 2)\cdot G - \frac{m}{2\cdot m_s^2}\cdot(o+2\cdot g-2\cdot G)\end{aligned}$$

To find critical points, we set $\frac{\partial T}{\partial m_s} = 0$. This gives us the following value for the optimal segment size:

$$m_s = \sqrt{\frac{m\cdot(o+2\cdot g-2\cdot G)}{4\cdot(\lceil\log_2(P+1)\rceil-2)\cdot G}} \tag{5.10}$$

To verify that the computed m_s value is indeed minimum, we compute $\frac{\partial^2 T}{\partial m_s^2}$

$$\frac{\partial^2 T}{\partial m_s^2} = \frac{m\cdot(o+2\cdot g-2\cdot G)}{m_s^3} = \frac{4\cdot(\lceil\log_2(P+1)\rceil-2)\cdot G}{m_s}$$

Since m_s is a positive number, the condition that $\frac{\partial^2 T}{\partial m_s^2} > 0$ is fulfilled when $P > 2$. □

Theorem 5.3 (Optimal segment size for split-binary broadcast, PLogP)**.** *The optimal segment size for the split-binary broadcast algorithm according to the PLogP model can be computed analytically as a solution to*

$$\begin{aligned}0 &= 2\cdot(\lceil\log_2(P+1)\rceil-1)\cdot g'(m_s)\\ &\quad -\frac{m}{2\cdot m_s^2}\cdot(g(m_s)+f(m_s)) + (\frac{m}{2\cdot m_s}-1)\cdot(g'(m_s)+f'(m_s))\end{aligned}$$

where $f(m_s) = \max\{g(m_s), o_r(m_s)+o_s(m_s)\}$ and $g'(m)$ and $f'(m)$ are defined for all message sizes, m.

Proof. To prove this statement, it is enough to note that $\frac{\partial T}{\partial m_s}$ for the split-binary broadcast operation according to the PLogP model corresponds to the right-hand side of the equation above.

$$\begin{aligned}T &= (\lceil\log_2(P+1)\rceil-1)\cdot(L+2\cdot g(m_s)) +\\ &\quad (\frac{m}{2\cdot m_s}-1)\cdot(g(m_s)+f(m_s)) +\\ &\quad L + g(\frac{m}{2})\\ \frac{\partial T}{\partial m_s} &= 2\cdot(\lceil\log_2(P+1)\rceil-1)\cdot g'(m_s) +\\ &\quad -\frac{m}{2\cdot m_s^2}\cdot(g(m_s)+f(m_s)) + (\frac{m}{2\cdot m_s}-1)\cdot(g'(m_s)+f'(m_s))\end{aligned}$$

From here, the optimal segment size can be found using Fermat's theorem, by solving $\frac{\partial T}{\partial m_s} = 0$. □

In the PLogP-model case, in general, the analytical form for $g(m)$, $o_s(m)$, and $o_r(m)$ does not exist. However, different interpolation or polynomial fit methods can be used to approximate the analytical form for these parameters.

Corollary 5.4 (Optimal segment size for split-binary broadcast and piecewise linear $g(m)$, $o_s(m)$, and $o_r(m)$, PLogP)**.** *Let $g(m)$, $o_s(m)$, and $o_r(m)$, be piecewise linear and continuous functions of the message size: $g(m) = \cup_{k=0}^{n_g} C_k\cdot m + c_k$, $o_s(m) = \cup_{k=0}^{n_s} A_k\cdot m + a_k$, and $o_r(m) = \cup_{k=0}^{n_r} B_k\cdot m + b_k$, where n_g, n_s, and n_r correspond to the number of linear segments. Then, the solution for an optimal segment size for the split-binary broadcast does not exist within linear intervals, and the only candidate points are interval boundaries and at the the cross sections of $g(m)$ and $o_s(m)+o_r(m)$.*

Proof. Starting from the PLogP-based model for the split-binary algorithm in Table 4.3, we compute T using the new values for $g(m)$, $o_s(m)$, and $o_r(m)$. Let segments i_g, i_s, and i_r cover message size m_s and segments j_g, j_s, and j_r cover message size $\frac{m}{2}$.

$$\begin{aligned}
T &= (\lceil log_2(P+1)\rceil - 1)\cdot(L + 2\cdot g(m_s)) \; + \\
&\quad (\lceil \tfrac{n_s}{2}\rceil - 1)\cdot \max\{2\cdot g(m_s),\;\; o_r(m_s)+g(m_s)+o_s(m_s)\} + \\
&\quad L + g(\tfrac{m}{2}) \\
T &= (\lceil log_2(P+1)\rceil - 1)\cdot(L + 2\cdot C_{i_g}\cdot m_s) \;\; + L + C_{j_g}\cdot \tfrac{m}{2} + \\
&\quad (\lceil \tfrac{m}{2\cdot m_s}\rceil - 1)\cdot(C_{i_g}\cdot m_s + m_s\cdot \max\{C_{i_g},\,(A_{i_s}+B_{i_r})\})
\end{aligned}$$

We treat the three different cases separately.

Assume $g(m_s) \geq o_r(m_s) + o_s(m_s)$.

$$\begin{aligned}
T &= (\lceil log_2(P+1)\rceil - 1)\cdot(L + 2\cdot C_{i_g}\cdot m_s) \;\; + L + C_{j_g}\cdot \tfrac{m}{2} \\
&\quad 2\cdot(\lceil \tfrac{m}{2\cdot m_s}\rceil - 1)\cdot(C_{i_g}\cdot m_s) \\
\tfrac{\partial T}{\partial m_s} &= 2\cdot(\lceil log_2(P+1)\rceil - 1)\cdot C_{i_g} + \\
&\quad 2\cdot(-\tfrac{m}{2\cdot m_s^2})\cdot C_{i_g}\cdot m_s + 2\cdot(\tfrac{m}{2\cdot m_s} - 1)\cdot C_{i_g}
\end{aligned}$$

Employ Fermat's theorem $\frac{\partial T}{\partial m_s} = 0$ to find critical values of m_s.

$$\begin{aligned}
0 &= (\lceil log_2(P+1)\rceil - 1) - \tfrac{m}{2\cdot m_s^2}\cdot m_s + (\lceil \tfrac{m}{2\cdot m_s}\rceil - 1) \\
0 &= (\lceil log_2(P+1)\rceil - 1) - \tfrac{m}{2\cdot m_s} + \lceil \tfrac{m}{2\cdot m_s}\rceil - 1 \\
0 &= \lceil log_2(P+1)\rceil - 2
\end{aligned}$$

The equation above has no solution if $P \notin \{2,3\}$. If $P \in \{2,3\}$ then there are an infinite number of solutions for m_s, and thus the optimal one does not exist.

Assume $g(m_s) < o_r(m_s) + o_s(m_s)$.

$$\begin{aligned}
T &= (\lceil log_2(P+1)\rceil - 1)\cdot(L + 2\cdot C_{i_g}\cdot m_s) \;\; + L + C_{j_g}\cdot \tfrac{m}{2} + \\
&\quad (\lceil \tfrac{m}{2\cdot m_s}\rceil - 1)\cdot(C_{i_g} + A_{i_s} + B_{i_r})\cdot m_s) \\
\tfrac{\partial T}{\partial m_s} &= (\lceil log_2(P+1)\rceil - 1)\cdot 2\cdot C_{i_g} \\
&\quad -\lceil \tfrac{m}{2\cdot m_s^2}\rceil\cdot(C_{i_g} + A_{i_s} + B_{i_r})\cdot m_s + (\lceil \tfrac{m}{2\cdot m_s}\rceil - 1)\cdot(C_{i_g} + A_{i_s} + B_{i_r}) \\
&= (\lceil log_2(P+1)\rceil - 1)\cdot 2\cdot C_{i_g} \\
&\quad -\lceil \tfrac{m}{2\cdot m_s}\rceil\cdot(C_{i_g} + A_{i_s} + B_{i_r}) + (\lceil \tfrac{m}{2\cdot m_s}\rceil - 1)\cdot(C_{i_g} + A_{i_s} + B_{i_r}) \\
&= (2\cdot\lceil log_2(P+1)\rceil - 3)\cdot C_{i_g} + A_{i_s} + B_{i_r} \\
0 &= (2\cdot\lceil log_2(P+1)\rceil - 3)\cdot C_{i_g} + A_{i_s} + B_{i_r}
\end{aligned}$$

The equation $\frac{\partial T}{\partial m_s} = 0$ has no solution for $P \geq 2$ since $C_{i_g} > 0$, $A_{i_s} > 0$, and $B_{i_r} > 0$ are greater than zero, making the whole expression greater than zero.

Without the loss of generality, assume that at segment k, $(m_{k_0},\, m_{k_1})$, $g(m_s) < o_r(m_s) + o_s(m_s)$ for $m_s < m_b$, and $g(m_s) > o_r(m_s) + o_s(m_s)$ for $m_s > m_b$, and $m_{k_0} \leq m_b m_{k_1}$. The m_b is a cross over point between $g(m_s)$ and $o_r(m_s) + o_s(m_s)$. Then the previous analysis applies to message segments $(m_{k_0},\, m_b)$ and $(m_b,\, m_{k_1})$. Thus, the only points at which we can have maximum or minimum in addition to m_{k_0} and m_{k_1} is m_b.

Thus, an optimal segment size does not exist within the linear intervals. The only other option for solution is to evalute function values at interval boundary points (including no segmentation) and determine whether any of the segment sizes decreases overall duration. □

Remark Corollary 5.4 implies that using linear interpolation for $g(m)$, $o_s(m)$, and $o_r(m)$ reduces the problem of finding the optimal segment size for the split-binary broadcast algorithm according to PLogP model, to the process of evaluating the model prediction at measured values of gap and overheads and selecting the one which achieves minimum.

Corollary 5.5 (Optimal segment size for split-binary broadcast and exponential $g(m)$, PLogP)**.** *Let $g(m)$ be an exponential function of message size, $g(m) = a_g \cdot e^{b_g \cdot m}$ and $g(m_s) > o_s(m_s)+o_r(m_s)$. Then, the solution for the optimal segment size for the split-binary broadcast is*

$$m_s = \frac{-m \cdot b_g + \sqrt{m^2 \cdot b_g^2 + 8 \cdot m \cdot b_g \cdot (\lceil \log_2(P+1) \rceil - 1)}}{4 \cdot (\lceil \log_2(P+1) \rceil - 1)} \tag{5.11}$$

Proof. The proof is analogous to the proofs of Theorems 5.1 and 5.2 . We compute $\frac{\partial T}{\partial m_s}$ and solve $\frac{\partial T}{\partial m_s} = 0$ for m_s. The positive root is the segment size that minimizes duration of the split-binary broadcast operation, because the original model represents a concave function without an upper bound.

$$\begin{aligned} T &= (\lceil \log_2(P+1) \rceil - 1) \cdot (L + 2 \cdot g(m_s)) + (\tfrac{m}{2 \cdot m_s} - 1) \cdot 2 \cdot g(m_s) + L + g(\tfrac{m}{2}) \\ &= \lceil \log_2(P+1) \rceil \cdot L + (\lceil \log_2(P+1) \rceil + \tfrac{m}{2 \cdot m_s} - 2) \cdot 2 \cdot a_g \cdot e^{b_g \cdot m_s}) + a_g \cdot e^{b_g \cdot \frac{m}{2}} \\ \frac{\partial T}{\partial m_s} &= -\tfrac{m}{m_s^2} \cdot a_g \cdot e^{b_g \cdot m_s} + (\lceil \log_2(P+1) \rceil + \tfrac{m}{2 \cdot m_s} - 2) \cdot 2 \cdot a_g \cdot b_g \cdot e^{b_g \cdot m_s} \end{aligned}$$

Now, we compute values of the critical points of T using Fermat's theorem.

$$\begin{aligned} \frac{\partial T}{\partial m_s} &= 0 \\ 0 &= 2 \cdot (\lceil \log_2(P+1) \rceil - 1) \cdot b_g + \tfrac{m}{m_s} \cdot b_g - \tfrac{m}{m_s^2} \\ 0 &= 2 \cdot (\lceil \log_2(P+1) \rceil - 1) \cdot b_g \cdot m_s^2 + m \cdot b_g \cdot m_s - m \end{aligned}$$

The solution to the quadratic equation above is

$$m_s = \frac{-m \cdot b_g \pm \sqrt{m^2 \cdot b_g^2 + 8 \cdot m \cdot b_g \cdot (\lceil \log_2(P+1) \rceil - 1)}}{4 \cdot (\lceil \log_2(P+1) \rceil - 1)}$$

The positive root of the equation above is selected as an optimal segment size. □

Remark *The case $g(m_s) < o_r(m_s) + o_s(m_s)$ does not have closed form solution for m_s. Approximation methods need to be used to find the m_s satisfying $\frac{\partial T}{\partial m_s} = 0$ in that case.*

Theorem 5.6 (Optimal segment size for linear broadcast does not exist)**.** *The optimal segment size for the linear broadcast algorithm according to all considered models does not exist.*

Proof. We take the same approach as in proofs for Theorems 5.1 and 5.2. Starting from models in Table 4.3, we use Fermat's theorem to find critical points for the appropriate function.

According to the Hockney model of segmented linear broadcast in Table 4.3, the critical point at which $\frac{\partial T}{\partial m_s} = 0$ occurs when $-(P-1) \cdot \frac{m}{m_s} \cdot \alpha = 0$. Since $\alpha > 0$ and $m > 0$ for non-trivial case, this condition cannot be fulfilled. Hence, the optimal segment size according to the Hockney model does not exist.

As far as the LogP/LogGP model is concerned, the $\frac{\partial T}{\partial m_s} = 0$ condition becomes $(P-1) \cdot \frac{m}{m_s^2} \cdot (G-g) = 0$. All model parameter values are positive real numbers greater than 0, implying that the solution for this equation does not exist. According to the segmented linear broadcast LogP/LogGP model, an optimal segment size does not exist.

The PLogP-based model requires additional analysis as we have to deal with $g(m_s)$. From the basic model, $T = L + (P-1) \cdot \frac{m}{m_s} \cdot g(m_s)$, we compute $\frac{\partial T}{\partial m_s} = (P-1) \cdot \frac{m}{m_s} \cdot (-\frac{1}{m_s} \cdot g(m_s) + g'(m_s))$. The condition $\frac{\partial T}{\partial m_s} = 0$ then translates to

$$g'(m_s) - \frac{1}{m_s} \cdot g(m_s) = 0. \tag{5.12}$$

Whether we can satisfy this condition depends on the value of $g(m_s)$. The equation 5.12 is a homogenous linear differential equation. We can solve it for $g(m_s)$:

$$\begin{aligned}
g'(m_s) - \tfrac{1}{m_s} \cdot g(m_s) &= 0 \\
\tfrac{1}{m_s} \cdot g'(m_s) - \tfrac{1}{m_s^2} \cdot g(m_s) &= 0 \\
\tfrac{d}{dm_s}\left(\tfrac{1}{m_s} \cdot g(m_s)\right) &= 0 \\
\tfrac{g(m_s)}{m_s} &= C \\
g(m_s) &= C \cdot m_s
\end{aligned}$$

The result implies that only the situation in which $\frac{\partial T}{\partial m_s} = 0$ can occur is when there is a linear dependence between $g(m)$ and m, $g(m) = C \cdot m$. However, in this case, Equation 5.12 has an infinite number of solutions. Thus, the optimal segment size does not exist even in this case. $\square$

The previous analysis shows that the Hockney- and LogP/LogGP-based models are amenable to analytical analysis and interpretation. The major strength of the PLogP-based models, the parameter dependence on message size, also proved to be its major weakness in this case. As long as the analytical model for the parameter values is lacking, we cannot use analytical analysis to optimize these functions.

It is worth noting that even though we may know the optimal segment size, it may not be practical to use it at run time. With the collective operation implementation on top of point-to-point communication, the segment size must be a multiple of the operation datatype. Segmentation cannot be used for a collective with a single element of rather large datatype (eg. upper triangular matrix). Thus, our choices are selecting a segment size in the vicinity of the optimal one, when applicable, or not using segmentation at all.

In the remainder of this section, we provide the optimal segment sizes for all Hockney- and LogP/LogGP-based performance models of segmented algorithms from Section 4.3 without proof (Tables 5.1 through 5.4.)

Broadcast	**Model**	**Optimal segment size**
Linear	Hockney	none
Linear	LogP / LogGP	none
Pipeline	Hockney	$m_s = \sqrt{\frac{m\cdot\alpha}{(P-2)\cdot\beta}}$
Pipeline	LogP / LogGP	$m_s = \sqrt{\frac{m\cdot(o+g-G)}{(P-2)\cdot G}}$
Binomial	Hockney	none
Binomial	LogP / LogGP	none
Binary	Hockney	$m_s = \sqrt{\frac{2\cdot m\cdot\alpha}{(\lceil\log_2(P+1)\rceil-2)\cdot\beta}}$
Binary	LogP / LogGP	$m_s = \sqrt{\frac{m\cdot(o+2\cdot(g-G))}{2\cdot(\lceil\log_2(P+1)\rceil-2)\cdot G}}$
Split-binary	Hockney	$m_s = \sqrt{\frac{m\cdot\alpha}{2\cdot(\lceil\log_2(P+1)\rceil-2)\cdot\beta}}$
Split-binary	LogP / LogGP	$m_s = \sqrt{\frac{m\cdot(o+2\cdot(g-G))}{4\cdot(\lceil\log_2(P+1)\rceil-2)\cdot G}}$

Table 5.1: Optimal segment size for segmented broadcast algorithms.

Reduce	**Model**	**Optimal segment size**
Linear	Hockney	none
Linear	LogP / LogGP	$m_s = \sqrt{m\cdot\frac{(P-1)\cdot(o-G)}{G}}$, if $(P-1)\cdot(o+(m_s-1)\cdot G+\gamma\cdot m_s > g$
Pipeline	Hockney	$m_s = \sqrt{\frac{m\cdot\alpha}{(P-2)\cdot(\beta+\gamma)}}$
Pipeline	LogP / LogGP	$m_s = \sqrt{\frac{m\cdot(2\cdot o-G)}{(P-2)\cdot(G+\gamma)}}$ if $(2\cdot o+(m_s-1)\cdot G+\gamma\cdot m_s > g$
Binomial	Hockney	none
Binomial	LogP / LogGP	none
Binary	Hockney	$m_s = \sqrt{\frac{m\cdot\alpha}{(\lceil\log_2(P+1)\rceil-2)\cdot(\beta+\gamma)}}$
Binary	LogP / LogGP	$m_s = \left\{\begin{array}{l}\sqrt{\frac{m\cdot(g-G)}{2\cdot\gamma}},\ \text{if } g\geq o+\gamma\cdot m_s \\ \sqrt{\frac{m\cdot(o-G)}{2\cdot\gamma}},\ \text{if } g< o+\gamma\cdot m_s\end{array}\right\}$

Table 5.2: Optimal segment size for segmented reduce algorithms.

Allreduce	**Model**	**Optimal segment size**
Ring with segs.	Hockney	$m_s = \sqrt{\frac{m \cdot \alpha}{(P-2)\cdot(\beta+\gamma)}}$
Ring with segs.	LogP / LogGP	$m_s = \left\{ \begin{array}{ll} \sqrt{\frac{m\cdot(g-G)}{(P-2)\cdot G}} & \text{if } g \geq o + \gamma \cdot m_s \\ \sqrt{\frac{m\cdot(o-G)}{(P-2)\cdot G-\gamma}} & \text{otherwise} \end{array} \right\}$

Table 5.3: Optimal segment size for segmented allreduce algorithms.

Scan	**Model**	**Optimal segment size**
Linear with segs.	Hockney	$m_s = \sqrt{\frac{m \cdot \alpha}{(P-2)\cdot(\beta+\gamma)}}$
Linear with segs.	LogP / LogGP	$m_s = \sqrt{\frac{m\cdot(2\cdot o-G)}{(P-2)\cdot(G+\gamma)}}$

Table 5.4: Optimal segment size for segmented scan algorithms.

5.2.3 Generating decision function source code

The simplest approach to select a collective method using analytical methods is to select one point-to-point model and compare the results of the corresponding collective performance models for a particular set of system and collective operation input parameters. Alternatively, we can query all available models and select the method with most "votes." In the case all three models predict differently, we can give weight to each of the predictions and select the most trusted one. Based on model evaluation results from Section 4.4, PLogP- and LogP/LogGP-based predictions should be given more weight than the Hockney-based models.

The decision function source code in this case contains two parts: model querying and result comparison. The model querying part can invoke either single or multiple performance models and can consider a couple of segment size candidates (including none). Result comparison is a search for the best performing method based on performance model results.

5.2.4 Limitations

Limitations of the analytical approach fall into the following groups: limitations of performance models themselves; limitations in finding optimal segment size; and difficulty of implementation.

The limitations of performance models were discussed in Section 4.4. The limitations regarding optimal segment size were pointed out in the previous section. The implementation of this approach requires either an expression parser or that all the models are hard coded. In addition, the functionality to interpolate values of PLogP model parameters is needed as well. For the purpose of this dissertation, we implemented this functionality in Matlab.

5.3 Graphical encoding methods: Quadtrees

One can interpret the information about the optimal collective implementation on a system (decision map), as an image and apply a standard compression algorithms to it. The encoded structure can be used to generate either a decision function code or an in-memory decision structure that can be queried at run-time. In this section, we introduce a quadtree-based approach to storing, analyzing, and retrieving optimal method information for a collective.

5.3.1 Quadtrees

Quadtree is a hierarchical data structure originally introduced by Finkel and Bentley in [Finkel and Bentley, 1974] to facilitate data retrieval on composite keys. The records with two-dimensional keys are stored in a tree structure in which every node has four children. A node itself represents a test on both attribute values, splitting the attribute parameter space into four regions. The data is stored in leaf nodes of the tree. The one-dimensional analog of quadtrees are binary trees, while the three-dimensional counterpart are octrees. Today, quadtree and its variants are used for image representation, computer aided design, spatial indexing, storing sparse data, etc.

A basic quadtree implementation contains a pointer to data and four pointers to children elements. In spatial context, quadtrees partition a two-dimensional space by recursively subdividing it into four quadrants (*NW*, *SW*, *SE*, and *NE*), by halving each of the region edges. A quadtree is defined in a planar square universe consisting of $2^k \times 2^k$ basic cells. Figure 5.3 shows an example of a region divided by a quadtree and corresponding data structure. Depending on the application, the node may need to keep the range of attribute values it covers [Knuth, 1998]. A complete quadtree with k levels contains $\frac{4^{(k+1)}-1}{4-1}$ nodes. On the average, a search operation on a random quadtree takes $O(n \cdot \log n)$ steps, but the worst case performance is $O(n^2)$ [Finkel and Bentley, 1974].

5.3.2 Quadtree encoding and MPI collective operations

We use the decision map of an optimal collective implementation on a particular system to construct a quadtree that encodes this information. In the encoding scheme, every non-leaf node in the quadtree corresponds to a test that matches both communicator and message size values. The leaf nodes contain information about the optimal method for the particular communicator and message size ranges. Thus, leaves represent the regions into which the decision map is divided, and the internal nodes represent the rules of the corresponding decision function. As a consequence, quadtrees allow us to perform a recursive binary search in a two-dimensional space. The input parameter space can be extended by utilizing higher dimensional trees, such as octrees.

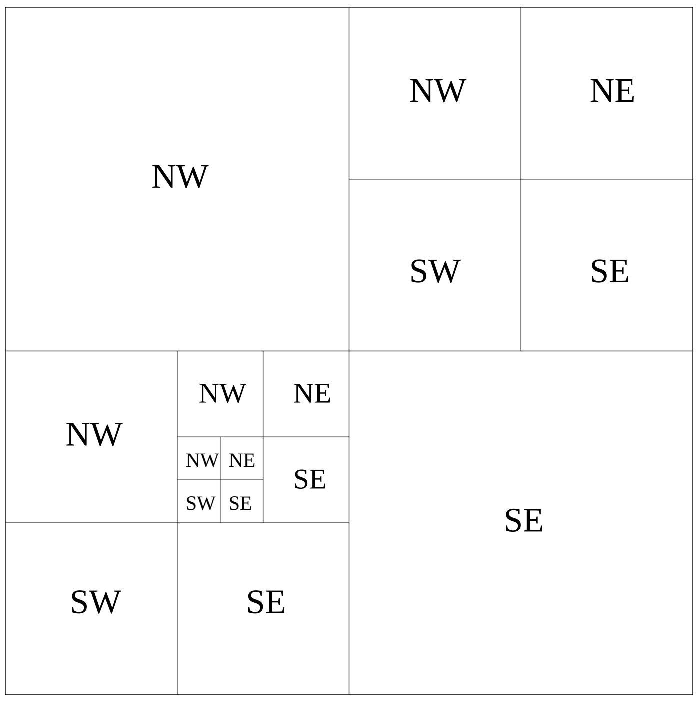
The decision map used to initialize the quadtree must be a *complete* and *square* matrix with a power of two dimension size, $(2^k \times 2^k)$. A complete decision map means that tests must cover all message and communicator sizes of interest. Neither of these requirements are real limitations, as the missing data can be interpolated, and the size of the map can be adjusted by replicating some of the entries. The replication process does not affect the quadtree decisions but may affect the efficiency of the encoding (both in a positive and negative manner).

We split the region based on the number of data points, not the actual communicator and message size values. Thus, our quadtree data structure must include information about the communicator and the message size ranges covered by the particular node. This approach ensures that all sibling nodes cover an equal number of data points. However, different regions of a decision map can have different complexities, so the tree structure may still end up being unbalanced.

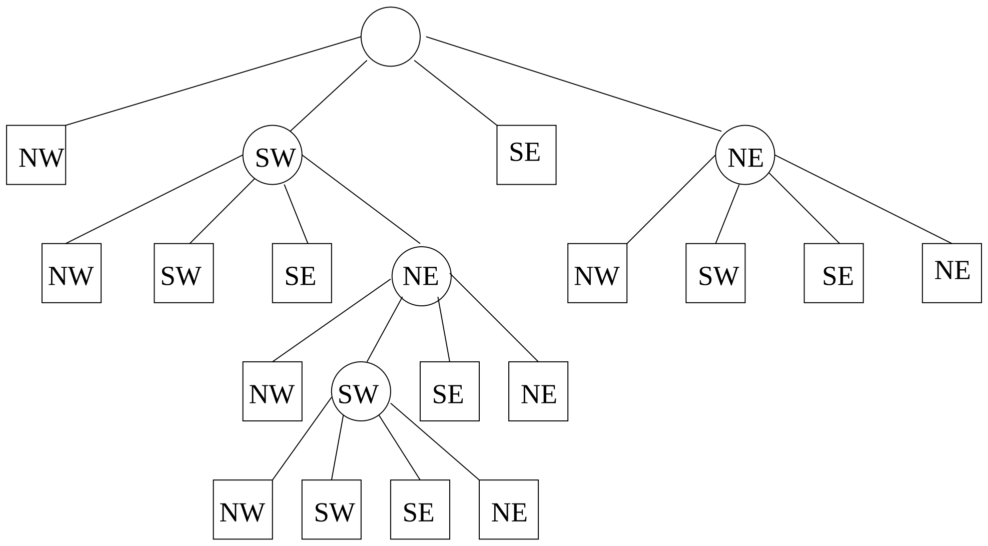
Once the encoded quadtree structure is populated, the average tree depth corresponds to the average number of tests that need to be evaluated to reach the decision, i.e., select appropriate algorithm. The minimum and the maximum tree depths correspond to the best-case and worst-case scenario, respectively. Thus, manipulating the size of a quadtree can provide us with a trade-off between the tree accuracy and the performance (time-to-decision) of the tree.

Exact quadtree

An *exact* quadtree truthfully represents the experimental data. If the decision map originally contained N data points, the maximum depth of an exact quadtree that can encode this amount of information is $k = \lceil \log_4 N \rceil$.



(a)



(b)

Figure 5.3: **(a)** A rectangular region split using quadtree encoding. **(b)** A point quadtree data-structure corresponding to the regions on the left. Square nodes denote leaves, and round nodes represent branching points.

Maximum-depth limited quadtree

Limiting the absolute tree depth sets the upper limit to the maximum number of tests we may need to execute in order to determine the method index for the specified communicator and message size. At the last level, the node is assigned the prevalent method color. By convention, in case there is a tie, the method of the point in the upper right corner is selected if that method was one of the ties, if not, the smallest method index among the methods in a tie is chosen.

Accuracy-threshold limited quadtree

Alternatively, the tree growth can be stopped once a large enough portion of datapoints in the region belong to the same method. For example, once 75% of the points belong to the binomial algorithm without segmentation, denote the whole region as "binomial without segmentation" ignoring the remaining points. We refer to this tree-building constraint as *accuracy threshold*. Setting the accuracy threshold helps smooth the experimental data, thus possibly making the decision function more resistant to anomalies in measurements.

A quadtree with either a threshold or a maximum depth limit allows us to reduce the size of the tree at the cost of prediction accuracy, as it is no longer an exact copy of the original data. Applying the maximum depth and/or the accuracy thresholds is equivalent to applying low-pass filters to the original data set.

5.3.3 Generating decision function source code

Generating decision function source code from a constructed quadtree is straight forward. Recursively, for every internal node in the quadtree the following code segment can be generated:

```
if (NW) {...}
else if (SW) {...}
else if (SE) {...}
else if (NE) {...}
else {error}
```

where *NW*, *NE*, *SW*, and *SE* correspond to the north-west, north-east, south-west, and south-east quadrants of the region, respectively.

This implementation is functional but lacks possible optimizations, such as the ability to merge conditions belonging to a same method (effectively combining two siblings into one). As the collective operation parameter input space is theoretically infinite, by convention, the conditions for the dataset boundary points (minimum and maximum communicator and message sizes) are expanded to fully cover that region. For example, the decision for minimum communicator size will be applied to all communicator sizes smaller than minimum.

5.3.4 In-memory quadtree decision structure

An alternative to generating the decision function source code is maintaining an in-memory quadtree decision structure, which can be queried at run-time.

An optimized quadtree structure contains four pointers and one method field, which could probably be a single byte or an integer value. Thus, the size of a node of the tree would be around 36 bytes on 64-bit architectures.[*] In addition, the system needs to maintain an in-memory mapping of (algorithm, segment size) pairs to method indexes, as well as the communicator and message sizes used to construct the quadtree.

The maximum depth decision quadtree we encountered in our tests had six levels. This means that in the worst case, the 6-level decision quadtree could take up to $\frac{4^7-1}{4-1} = 5461$ nodes, which would occupy around 192KB of memory. However, our results indicate that the quadtrees with three levels can still produce reasonably good decisions. A 3-level quadtree would occupy at most 3060 bytes and as such could fit into one 4KB page of main memory. As the decision function will be called occasionally (i.e., once for each tuple (*collective, message size*)), the in-memory quadtree will not be cached. Therefore, each invocation of the decision system is expected to generate a large number of cache misses.

Our prototype implementation provides tools for managing the in-memory decision functions - however the internal node structure occupies 48 bytes instead of the minimal 36 bytes. The associated structure for method mapping requires 6B per method, and the communicator and message sizes are represented using integer arrays (2B per element). Querying the structure involves determining the indexes of specified communicator and message sizes. We believe that a fully-optimized version should achieve better performance than this prototype version.

5.3.5 Limitations

Quadtrees are designed to encode two-dimensional data. Thus, a major limitation of this encoding is that it will not be able to capture the decisions that are optimal for single communicator values, e.g., communicator sizes that are power of two. The same problem is exacerbated if the performance measurement data used to construct trees is too sparse. The sparse data set and single line rules are high-frequency information, and applying low-pass filters to it can cause loss of important information.

In addition, the quadtree-based approach is limited to two input parameters. While octrees can be used to increase the number of the parameters to three, extending this approach further is not necessarily feasible. However, as most current MPI implementations do not make additional run-time information (such as processor load or network utilization) globally available, limiting the input parameter space to two or possibly three dimensions may not be a real restriction.

Finally, the decision map reshaping process to convert measured data from an $n \times m$ shape to $2^k \times 2^k$ affects encoding efficiency of the quadtree. In our current study, we did not address this issue.

5.4 Statistical learning methods: C4.5 decision trees

Data mining techniques can be applied to the algorithm selection problem by replacing the original problem with an equivalent classification problem. The new problem is to classify

[*]In this analysis, we ignore data alignment issues which could lead to even larger size of the structure.

collective parameters, eg. *(collective operation, communicator size, message size, root)*, into a correct category, a method in our case, to be used at run-time.

Unsupervised learning methods, such as clustering, can be used to discover methods with similar characteristics and performance. However, querying the clustering system at run-time can be computationally expensive. Alternatively, it is possible to use regression trees to predict the performance of individual methods instead of the analytical performance models (See Section 5.2). However, we are interested in capturing the patterns that occur in the optimal implementation of a collective operation and translating them into a run-time decision function. Thus, our problem considers only predefined set of classes, and each data point belongs to a single class. Under these problem constraints, supervised learning methods, such as decision trees, are the most natural choice. In this section we introduce C4.5 decision trees and discuss their applicability to the problem at hand.

5.4.1 C4.5 algorithm

C4.5 is a supervised learning classification algorithm used to construct decision trees from the data [Quinlan, 1993]. C4.5 can be applied to the data that fulfills the following requirements:

- *Attribute-value description*: information about a single entry in the data must be described in terms of attributes. The attribute values can be discrete or continuous and, in some cases, the attribute value may be missing or can be ignored.
- *Predefined classes*: the training data has to be divided into predefined classes or categories. This is a standard requirement for supervised learning algorithms.
- *Discrete classes*: the classes must be clearly separated and a single training case either belongs to a class or it does not. C4.5 cannot be used to predict continuous class values such as the cost of a transaction.
- *Sufficient data*: the C4.5 algorithm utilizes an inductive generalization process by searching for patterns in data. For this approach to work, the patterns must be distinguishable from random occurrences. What constitutes the "sufficient" amount of data depends on a particular data set, its attribute and class values, and the number of regions that need to be discovered. In general, statistical methods used in C4.5 to generate tests require a reasonably large amount of data.
- *"Logical" classification models*: generated classification models must be represented as either decision trees or a set of production rules [Quinlan, 1993].

The C4.5 algorithm constructs the initial decision tree using a variation of the Hunt's method for decision tree construction (Figure 5.4). The main difference between C4.5 and other similar decision tree building algorithms is in the test selection and evaluation process (the last case in Hunt's algorithm, Figure 5.4). The C4.5 utilizes information *gain ratio* criterion, which maximizes normalized information gain by partitioning T in accordance with a particular test [Quinlan, 1993].

To define the *gain ratio* we have to look at the information conveyed by classified cases. Consider a set T of k training cases. If we select a single case $t \in T$ and decide that it belongs

Hunt's method for decision tree construction [Quinlan, 1993]

Given a set of training cases, T, and set of classes $C = \{C_1, C_2, ..., C_k\}$,
the tree is constructed recursively by testing for the following cases:

1) T contains one or more cases which all belong to the same class C_j:
   A leaf node is created for T and is denoted to belong to C_j class;
2) T contains no cases:
   A leaf node is created for T and is assigned the most most frequent class at the parent node;
3) T contains cases that belong to more than one class:
   Find a test that splits the T set to a single-class collections of cases.
   This test is based on a single attribute value and is selected such that it results in one or more mutually exclusive outcomes $\{O_1, O_2, ...O_n\}$.
   The set T is then split into subsets $\{T_1, T_2, ...T_n\}$ such that the set T_i contains all cases in T with outcome O_i.
   The algorithm is then called recursively on all subsets of T.

Figure 5.4: Hunt's method for decision tree construction [Quinlan, 1993].

to class C_j, then the probability of this message is $\frac{freq(C_j,T)}{|T|}$ and it conveys $-log_2(\frac{freq(C_j,T)}{|T|})$ bits of information. Thus, the average amount of information needed to identify the class of a case in set T, $info(T)$, can be computed as a weighted sum of per-case information amounts [Quinlan, 1993]:

$$info(T) = -\sum_{j=1}^{k} \frac{freq(C_j,T)}{|T|} \times log_2(\frac{freq(C_j,T)}{|T|}) \tag{5.13}$$

If the set T was partitioned into n subsets based on outcomes of test X, we can compute a similar information requirement, $info_X(T)$, [Quinlan, 1993]:

$$info_X(T) = \sum_{i=1}^{n} \frac{|T_i|}{|T|} \times info(T_i) \tag{5.14}$$

Then, the information gained by partitioning T in accordance with the test X can be computed as:

$$gain(X) = info(T) - info_X(T) \tag{5.15}$$

The predecessor to the C4.5 method, the ID3 algorithm, used *gain* criterion in Equation 5.15 to select the test for partition. However, the *gain* criterion is biased towards the high frequency data. To ameliorate this problem, C4.5 normalizes the information *gain* by the amount of the potential information generated by dividing T into n subsets, *split* $info(X)$:

$$split\ info(X) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times log_2(\frac{|T_i|}{|T|}) \tag{5.16}$$

The condition on which C4.5 selects the test to partition the set of available cases is defined as:

$$gain\ ratio(X) = \frac{gain(X)}{split\ info(X)} \tag{5.17}$$

C4.5 selects the test that maximizes the *gain ratio* value.

Once the initial decision tree is constructed, a pruning procedure is initiated to decrease the overall tree size and decrease the estimated error rate of the tree [Quinlan, 1993]. In the pruning process, the estimated classification error of a subtree is compared against the classification error of the case when the subtree was replaced by a leaf node classified as the most frequent class. If the classification error of a leaf is lower than the subtree - the subtree is replaced by the leaf. Thus, the pruning process prefers the large classes in cases when the distinction between the small and large class is not clear.

Additional parameters that affect the resulting decision tree are

- *weight*, which specifies the minimum number of cases of at least two outcomes of a test. For example, a weight set to 20, would require at least two child nodes to cover more than or equal to 20 test cases. There is no limit on remaining children, which are allowed not to have any test cases as well. This parameter prevents near-trivial splits that would result in almost flat and really wide trees.

- *confidence level*, which is used for prediction of tree error rates and affects the pruning process. The confidence level is used to estimate the true error rate of a leaf node on unseen cases. If a leaf node covers N cases and incorrectly classifies E cases of the training set, the estimated true error rate of this leaf on unseen cases is the upper limit, $U_{CF}(E, N)$, of the confidence interval for the E/N error rate of some specified confidence level CF. Thus, the lower the confidence level, the greater the amount of pruning that takes place.

- *attribute grouping*, which can be used to create attribute value groups for discrete attributes and possibly infer patterns occurring in sets of cases with different values of an attribute, but do not occur for other values of that attribute.

- *windowing*, which enables construction of multiple trees in iterative fashion, based on a portion of the test data. At every iteration, new training cases are added to the existing pool of training cases. The process can stop either after a specific number of iterations or once the accuracy of the new trees does not improve.

5.4.2 MPI collectives performance data and C4.5

The collected performance data can be described using the collective name, communicator and message size attributes. The collective name attribute has discrete values such as broadcast, reduce, etc. Communicator and message size attributes have continuous values. Additionally, constructive induction can be used to create composite attributes that can capture additional system information. For example, a *total data per node* attribute can be used to distinguish between a single-process-per-node and two-processes-per-node run. Moreover, such attributes can potentially indirectly capture information about the system bottlenecks.

```
Decision Tree:
message_size <= 512 :
|  communicator_size <= 4 :
| |  message_size <= 32 : ring (12.0/1.3)
| |  message_size > 32 : linear (8.0/2.4)
| communicator_size > 4 :
| |  communicator_size > 8 : bruck (100.0/1.4)
| |  communicator_size <= 8 :
| | |  message_size <= 128 : bruck (8.0/1.3)
| | |  message_size > 128 : linear (2.0/1.0)
message_size > 512 :
|  message_size > 1024 : linear (78.0/1.4)
|  message_size <= 1024 :
| |  communicator_size > 56 : linear (5.0/1.2)
| |  communicator_size <= 56 :
| | |  communicator_size <= 8 : linear (3.0/1.1)
| | |  communicator_size > 8 : bruck (5.0/1.2)
```

Figure 5.5: C4.5 decision tree for Alltoall on the Nano cluster. The numbers in parentheses represent the number of training cases covered by each leaf and the number of cases misclassified by that leaf.

The predefined set of classes in our case contains methods that were optimal for some of the data points. The class names consist of the algorithm name and segment size used, for example, Linear_0KB or SplitBinary_16KB. The classes are well defined, and by construction, the data with the same input parameters can belong to a single class only.

As far as the "sufficient" data requirement is concerned, the performance measurement data contains a considerable number of data points in the communicator - message size range. We do not cover every single possible communicator or message size, but our training data set usually contains around 1000 data points, so we feel that for this type of problem, the amount of collected data is sufficient to give reasonably accurate results.

Figure 5.5 shows a simple decision tree constructed by C4.5 from the data for an Alltoall collective on the Nano cluster.

5.4.3 Generating decision function source code

The goal of this work is the construction of decision functions, so we provide the functionality to generate the decision function source code in C from the constructed decision trees: the internal nodes are replaced by a corresponding *if* statement, and leaf nodes return the decision method index/name. We did not utilize the `c4.5rules` program for this purpose.

5.4.4 Limitations

If the class distribution is close to random, C4.5 (or any other classification algorithm for that matter) will be unable to produce accurate decision trees. The same is true for training sets that are highly biased towards one or two major classes: the pruning process may conclude that just classifying everything as a major class produces a tree with lower expected classification error.

In the presence of well defined regions, the main limitation of the C4.5 classification algorithm is that it divides the space using rectangular hyperplanes. Thus, it is unable to capture the borders which are function of two or more attributes. However, this problem can be addressed by defining the composite attribute that describes such dependence, but this approach requires the user to have prior knowledge about the data.

Chapter 6

Experimental Results

This chapter presents an experimental verification and comparison of collective operation optimization methods. We first analyze the results of the presented methods individually. Then we compare the performance of different methods for broadcast implementation on Grig cluster using FastEthernet. In addition, we provide large-scale results from the Thunderbird system at Sandia National Laboratory. Finally, we compare our optimized collectives against other MPI implementations and we visit the application tuning process.

Some, but not all, of the results in this chapter have been previously published in the following publications:

- [Pješivac-Grbović et al., 2007c]
  Jelena Pješivac-Grbović, George Bosilca, Graham E. Fagg, Thara Angskun, Jack J. Dongarra, "MPI Collective Algorithm Selection and Quadtree Encoding", *Parallel Computing*, vol. 33/9, pp. 613–623, September 2007.

- [Pješivac-Grbović, 2007]
  Jelena Pješivac-Grbović, Open MPI Collective Operation Performance on Thunderbird, *Technical Report, UT-CS-07-594*, The University of Tennessee, Computer Science Department, Knoxville, Tennessee, April, 2007.

- [Pješivac-Grbović et al., 2007a]
  Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, Jack J. Dongarra, "Performance analysis of MPI collective operations", *Cluster Computing*, vol. 10, pp. 127 – 143, June 2007.

- [Pješivac-Grbović et al., 2007b]
  Jelena Pješivac-Grbović, George Bosilca, Graham E. Fagg, Thara Angskun, Jack J. Dongarra, "Decision trees and MPI collective algorithm selection problem", *EuroPAR*, LNCS 4641, pp. 105–115, 2007.

Most results reported in this section were collected at the University of Tennessee clusters: Grig, Frodo, and Boba. The system characteristics of these clusters are available in Section 4.4. In addition, we report the results from an Infiniband cluster at Cisco and the large-scale results on the Thunderbird system at Sandia National Laboratory in Sections 6.3.3 and 6.4, respectively.

In this chapter, the results were collected using two standardized collective benchmarks, SKaMPI and OCC. SKaMPI (Special Karlsruher MPI - Benchmark) [SKaMPI, 2005] is an advanced benchmark with support for distributed clock synchronization, adaptive number of measurements, and etc. In SKaMPI, the processes synchronize their clocks initially, then schedule the collective operation to be executed in the future. The collective duration in this case is the maximum measured duration on all processes. The OCC (Optimized Collective Communication) benchmark [OCC, 2005] is a basic benchmark that measures the duration of an operation by repeating the operation number of times. To prevent pipelining effects, a balanced barrier (bruck algorithm) is inserted after every collective call. Thus, the OCC measures the time to execute collective operation plus the time for bruck barrier. The reported time is the maximum measured time on all processes, minus the time to execute bruck barrier. Both benchmarks report similar results. However, SKaMPI tends to collect results faster as it is capable of reducing the number of necessary tests automatically while maintaining the accuracy of the measurement. The OCC benchmark relies on the user to correctly adjust the total number of repetitions for every tests.

The reproducibility of the measured results is not within the scope of this paper, but both of the benchmarks we utilized follow the guidelines from [Gropp and Lusk, 1999] to ensure good quality measurements. Even so, the "exact" decision function corresponds to a particular data set, and the performance penalty of other decision functions is evaluated against the data that was used to generate the decision functions in the first place. Further more, performance analysis using micro-benchmarks does not guarantee an optimal application performance. It is possible that a particular collective algorithm performs very well in terms of micro-benchmark, but it is unbalanced and introduces process skew in the parallel application. The process skew, in turn, may or may not affect the overall application performance. At the same time, if the application has a form of a natural pipeline (for example there exists a single source that produces data to be distributed to other processes,) the algorithm that introduces a process skew inline with the application data flow can have better performance than a synchronous algorithm which takes smaller number of overall steps.

In this Chapter, the following acronyms are used in Figures depicting decision maps: "LIN" stands for "Linear", "BM" for "Binomial", "BIN" for "Binary", "SBIN" for "Split-Binary", and "PIPE" for "Pipeline" algorithm, while "none", "1KB", "8KB", "32KB", etc. are the corresponding segment sizes.

6.1 Analytical methods

This section expands the parallel communication model evaluation discussed in Section 4.4. We focus on process of determining the optimal segment size for the split-binary broadcast algorithm, and on determining the "optimal" and the "best-available" broadcast and allgather implementations on the Grig cluster.

6.1.1 Optimal segment size for split-binary broadcast

In Chapter 4 we introduced a number of segmented versions of collective algorithms. Introducing another, non-trivial parameter, increases the complexity of the decision functions

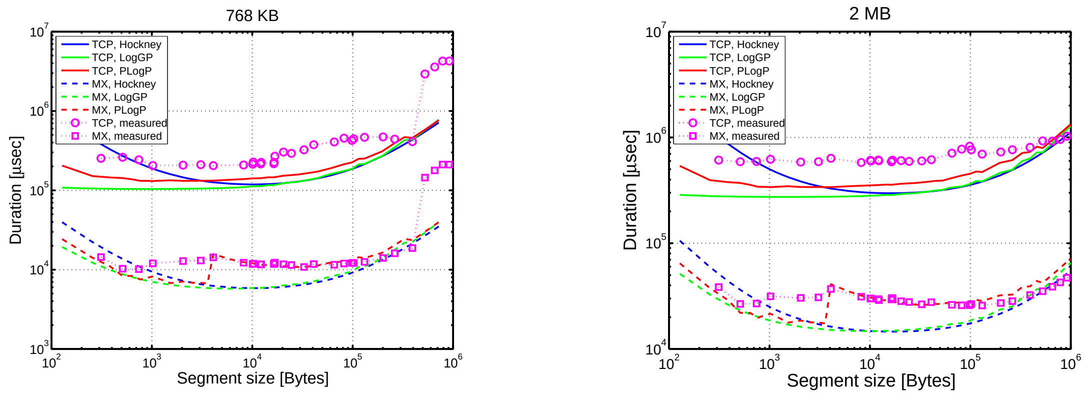


Figure 6.1: Effect of segmentation on split-binary broadcast performance on 64 processes for two message sizes: 768KB and 2MB on Grig cluster. Model predictions use the model parameter values from Tables 4.12 and 4.13 and Figure 4.19.

we need to construct. Thus, we would like to further examine the conditions under which the segmentation improves the performance and whether parallel communication models can be used for this purpose.

The effect of message segmentation on performance of the split-binary broadcast algorithm for an intermediate, 768KB, and a large, 2MB, data size on 64 processes was examined on Grig cluster. The model parameters used in this Section can be found in Tables 4.12 and 4.13, and Figure 4.19.

We first determine the optimal segment size according to the analytical models: Hockney and LogGP based on results of Theorems 5.1 and 5.2. The analytical solution for the optimal segment size according to PLogP model does not exist in this case, so we consider a large number of segment sizes in range from 312 B to 1 MB, and report the one with the shortest duration. Finally, we measure performance of the split-binary algorithm on 64 processes for 768 KB and 2 MB message sizes on the Grig cluster using 32 different segment sizes in the same segment size range.

The Figure 6.1 shows both the experimental results and model predictions. Table 6.1 summarizes these results. We report the optimal segment size on both MX and TCP / FastEthernet interconnect. For the PLogP model and experimental results, we report the segment size which achieved the absolute minimum, as well as the segment sizes that correspond to methods whose running time was within 5% and 10% of the predicted minimum. Figure 6.1 shows that neither of the models is properly capturing the TCP method performance, although the LogGP and PLogP models do have somewhat similar behavior. In the case of MX interconnect, the PLogP model is able to accurately follow measured results for segment sizes in range of 300 B to 100 KB. For large segment sizes and 2 MB message, all three models give acceptable estimate on the duration. However, neither of models is able to accurately capture the performance of the split-binary method with small segment sizes.

The results in Table 6.1 are not uniform. The Hockney model predictions were rather off the mark: approximately 10 KB segment size for 768 KB message in both MX and TCP case would take more than 10% extra time over the experimental minimum. Interestingly, according to the Hockney model the optimal segment size both over MX and FastEthernet

Model	**Message size**	**Optimal segment size**	
		MX	**TCP**
Hockney	768 KB	10308 B	10116 B
	2 MB	16834 B	16519 B
LogP / LogGP	768 KB	746 B	309 B
	2 MB	12185 B	505 B
PLogP	768 KB	3584 B	1024 B
		within 5%: 1536 B, 2560 B, 3072 B	**within 5%:** 640 B through 6144 B
	2 MB	3584 B	4096 B
		within 5%: 1536 B, 2560 B, 3 KB	**within 5%:** 768 B through 8192 B
Measured	768 KB	743 B	4096 B
		within 5%: 512 B	**within 5%:** 1 KB, 2 KB, 3 KB, 8 KB, 10 KB, 12 KB
		within 10%: 32 KB	**within 10%:** 16 KB
	2 MB	$128KB$ B **within 5%:** 512 B, 743 B, 32 KB, 64 KB, 80 KB, 90 KB, 100 KB **within 10%:** 20 KB, 24 KB, 40 KB, 200 KB, 256 KB	8192 B **within 5%:** 512 B, 743 B, 3 KB, 10 KB through 32 KB **within 10%:** 312 B through 40 KB

Table 6.1: Optimal segment size for split-binary broadcast on Grig cluster.

(TCP) is very similar. This is because the optimal segment size, m_s, is proportional to the square root of $\frac{\alpha}{\beta}$. The latency over MX is approximately 20 times lower than over TCP, while the transfer time is approximately 20 times higher over TCP, essentially making the ratio $\frac{\alpha}{\beta}$ constant.

The LogP / LogGP model predictions are surprisingly accurate for intermediate message size over MX, but the remaining predictions would incur more than 10% performance penalty. The PLogP predictions for TCP were fairly accurate. Utilizing one of the segment sizes which achieve algorithm duration within 5% of the absolute minimum, yields between 5% and 10% performance penalty. However, over the MX this is not the case, and again, we would incur more than 10% performance penalty.

The results from this experiment effectively show that using parallel communication models by themselves will not yield optimal results. In addition, there tends to be a number of methods which achieve comparable performance (within 5% from the minimum duration). Thus, utilizing only absolute minimum can lead to over-fitting and negligible run-time performance differences.

Finally, to answer the question whether the segmentation improves an algorithm performance we consider the right-most experimental value reported in Figures 6.1. This value corresponds to the non-segmented case, and in case of the split-binary broadcast algorithm on the Grig cluster, virtually using any segment size improves the overall method performance for communicator and message sizes we considered.

6.1.2 Analysis of broadcast implementation

In this section, we attempt to determine analytically an *optimal* and the *best available* broadcast implementation on Grig cluster using FastEthernet. An optimal implementation is built based on the available algorithms and achieves the overall minimum duration. The best available implementation is built on top of the available algorithms and predetermined set of segment sizes.

In this section, the experimental results were collected using OCC benchmark on Grig cluster using FastEthernet. The tests covered communicator sizes between two and 28 processes, and message sizes in range from 1B to 384 KB.

Optimal broadcast implementation

We determine an experimentally optimal broadcast implementation built on top of the linear,binomial, binary, pipeline, and split-binary broadcast algorithms using the parallel performance models from Table 4.3 and corresponding optimal segment sizes from Table 5.1.

Figure 6.2 shows the optimal broadcast implementation based on the Hockney and LogGP model predictions. We considered only these two parallel performance models, since the analytical solution for the PLogP-based models does not exist in this case. In Figure 6.2 (a) and (c) we can see which algorithms achieve absolute minimum for the particular communicator and message size. Figure 6.2 (b) and (d) shows the logarithm of two of the optimal segment sizes rounded to the nearest multiple of eight. The color maps on both plots are the same.

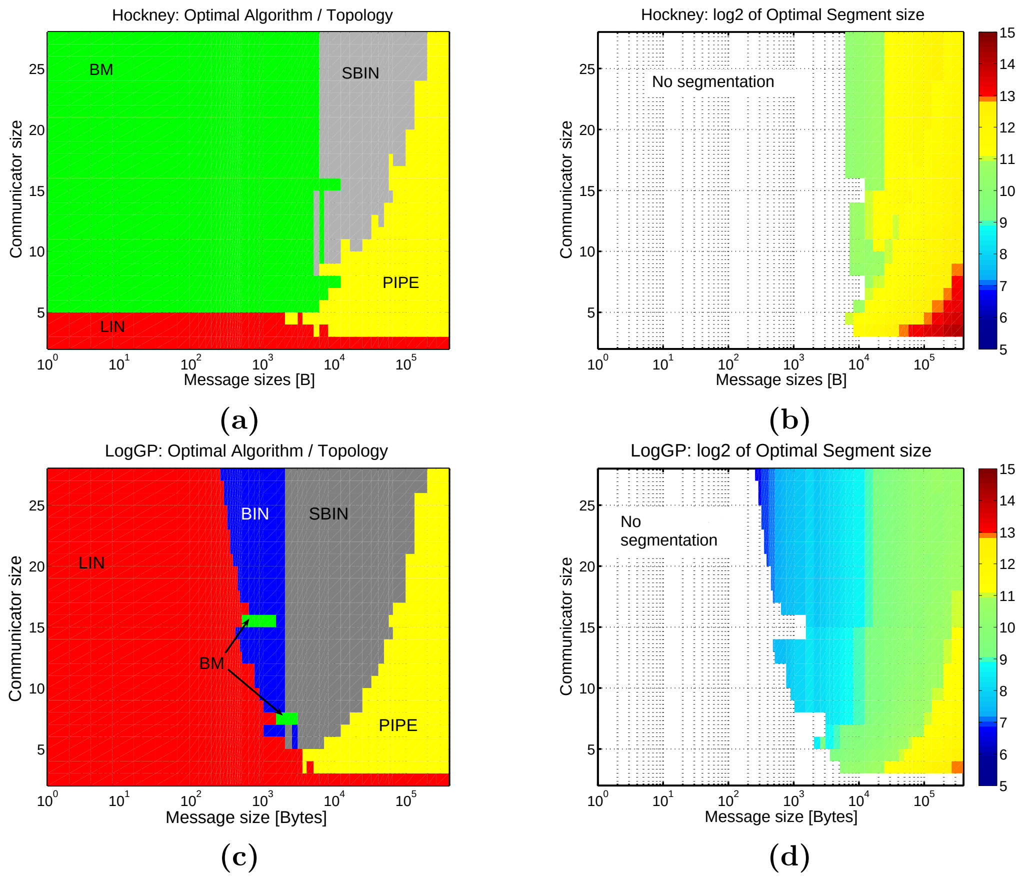


Figure 6.2: Optimal algorithms / topologies according to (a) Hockney and (c) LogGP models and $\log_2$ of the corresponding optimal segment sizes, (b) and (d), respectively.

The results are well aligned with our expectations: as the Hockney model pays higher overhead for sending multiple messages ($latency + m/bandwidth$), it prefers larger segment sizes than the LogGP model, which has to wait g before sending the next message. Also, Hockney models employ segmentation later, for messages larger than 10KB. The LogGP models in some cases suggests the message segmentation for messages even less than 1 KB. As such small message sizes are below the eager message size limit, physically, it makes no sense to use segmentation in this case.

Best broadcast implementation

Collecting the performance for the range of different segment sizes which need to be considered in the "optimal" case is not always feasible. Thus, it is hard to evaluate the exact properties of the decision functions based on these predictions. However, the data in Figure 6.2 indicates that the segment sizes of interest fall in the region between 64 B up to 32 KB. Both Hockney and LogGP models utilize no segmentation and segment sizes around 1 KB, 8 KB, and 16KB.

In this Section we consider the best broadcast implementation given the same set of basic algorithms and the set of segment sizes: none, 1 KB, 8 KB, 16 KB, and 32 KB. Figure 6.3 shows the four different decision maps: experimental, Hockney-, LogGP-, and PLogP-model based. Qualitatively, neither of the models capture the decision function completely.

The Hockney-model based decision function utilizes the larger segment sizes than the experiments suggest, and utilizes binomial broadcast without segmentation for small message sizes. However, the experiments show that the linear broadcast is still the algorithm of choice for small messages and communicator sizes we considered (up to 28 nodes). LogGP model captures the performance for the small and large message size, but for the message sizes between 1 KB and 10 KB, it utilizes split-binary algorithm without segmentation, while experimentally, binomial algorithm without and with 1 KB segments should be used. Similarly, the PLogP model predicts the optimal method for small message sizes well, and it is the only model that recommends using the binomial algorithm without segmentation for intermediate message sizes. However, it is unable to see the benefit of the split-binary algorithm, and recommends the binary algorithm with 1 KB segment sizes for intermediate-to-large message sizes on higher processes counts. Finally, PLogP does not switch from pipeline with 1 KB segments to larger segment sizes (8 KB) as in experimental and LogGP case.

Given the limitations of our models, it is reasonable to ask how useful are their predictions in building decision functions for real collective implementation. Additionally, what is the performance penalty the user will pay by using the model generated decision function instead of the measured one? Figure 6.4 addresses this question.

The Figure 6.4 shows the relative performance penalty incurred by selecting the method suggested by the performance models instead of the experimentally optimal one. The shade at point (m, P) corresponds to the relative performance slowdown of the model-suggested method for message size m and communicator size P. In this data set, the performance penalty for using the binomial algorithm without segmentation instead of the linear algorithm for small message sizes and all communicator sizes of interest can be as high as 1000%. This drove the mean performance penalty for Hockney based decision function

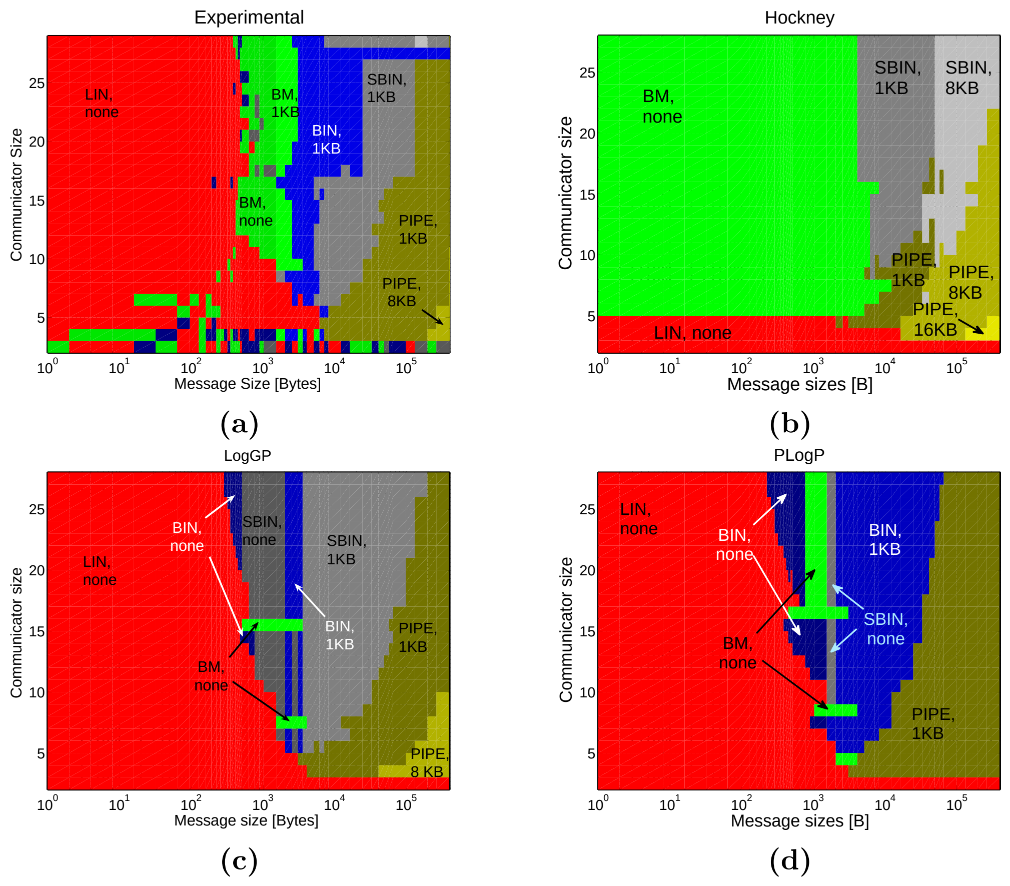


Figure 6.3: The best broadcast decision map according to (a) experiment, (b) Hockney, (c) LogGP, and (d) PLogP models.

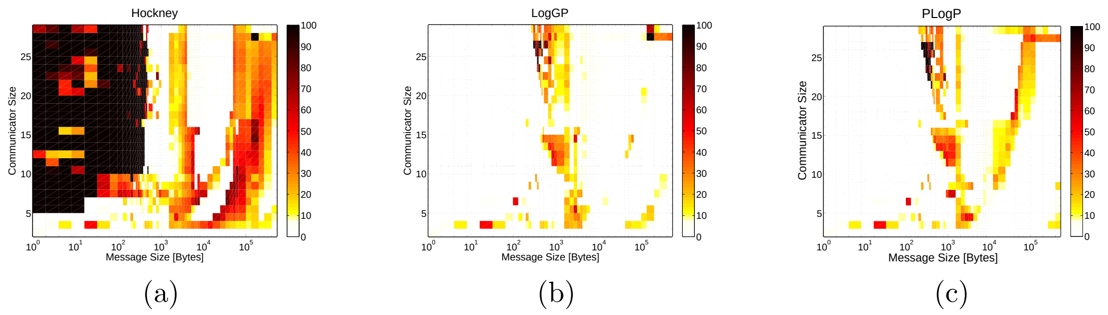


Figure 6.4: Performance penalty from using broadcast decision functions generated by models (a) Hockney, (b) LogGP, and (c) PLogP. White corresponds to less than 5%, black to 100% and higher performance penalty.

Model	Relative Performance Penalty [%]				Decision Time [μsec]
	Min	Max	Mean	Median	
Hockney	0.00	1061.34	85.61	22.89	759.28
LogP / LogGP	0.00	210.96	5.14	0.00	659.72
PLogP	0.00	246.06	7.03	0.00	2265.54

Table 6.2: Complete statistics for the performance of the model-generated decision maps, corresponding to Figures 6.3 and 6.4
.

to above 85% with median around 22%. The performance of LogGP and PLogP decision functions is much better: mean performance penalty is around 5% and 7% respectively, with median performance penalty 0 in both cases. Table 6.2 contains statistics about the relative performance penalty of these decision maps. specified message sizes.

In addition, one may be interested in time it took to make decision. Table 6.2, also reports the mean time it took to compute the best method for communicator and message size pair. The reported time is the time it took to the corresponding Matlab script to build decision map for this data set divided by the number of points (27 communicator sizes $\times$ 48 message sizes). The PLogP decision took much longer since it includes interpolation of PLogP parameters to specified message sizes. The measurements were taken on an AMD Athlon^{TM}64 Processor 3500+ at 2.2GHz with 512KB cache and 1GB RAM.

Finally, the interpretation of the relative performance of decision functions needs to be taken with caution, as the measured performance in this case was only the result of a micro-benchmark. Performance of real-world applications and their potential for performance losses or gains, varies greatly depending on application communication patterns.

6.1.3 Analysis of allgather implementation

In this section we analyze optimal allgather implementation on Grig cluster using MX interconnect. Data was collected using SKaMPI benchmark and Open MPI. Experiments covered all communicator sizes from two to 50 and 34 block sizes in range 1 B to 128 KB. We measured performance of bruck, recursive doubling, ring, and neighbor exchange algorithms (See Section 4.1.2.)

Figures 6.5 and 6.6 show the results of this analysis. Table 6.3 summarizes the performance penalty and the decision time statistics. Experimentally, the bruck algorithm is the algorithm of choice for block sizes up to 2 KB and non-power-of-two processes. Recursive doubling is preferred for power-of-two communicator sizes in almost all cases. The ring and neighbor exchange are used for intermediate and large block sizes. The neighbor exchange is only applicable for even number of processes, and tends to preform better than the ring in this case. Notable exception are the block sizes between 2 KB and 4 KB. In this range, the ring algorithm outperforms neighbor exchange for all communicator sizes we considered.

The Hockney model correctly predicts the bruck and recursive doubling algorithm for small block sizes. However, it does not switch to ring algorithm until 5 KB to 8 KB, thus incurring performance penalty between 10% and 50% in this range. In addition, the Hockney model fails to predict that the neighbor exchange algorithm can achieve better performance than the ring in most cases. This results in 10% to 40% performance penalty for even communicator sizes.

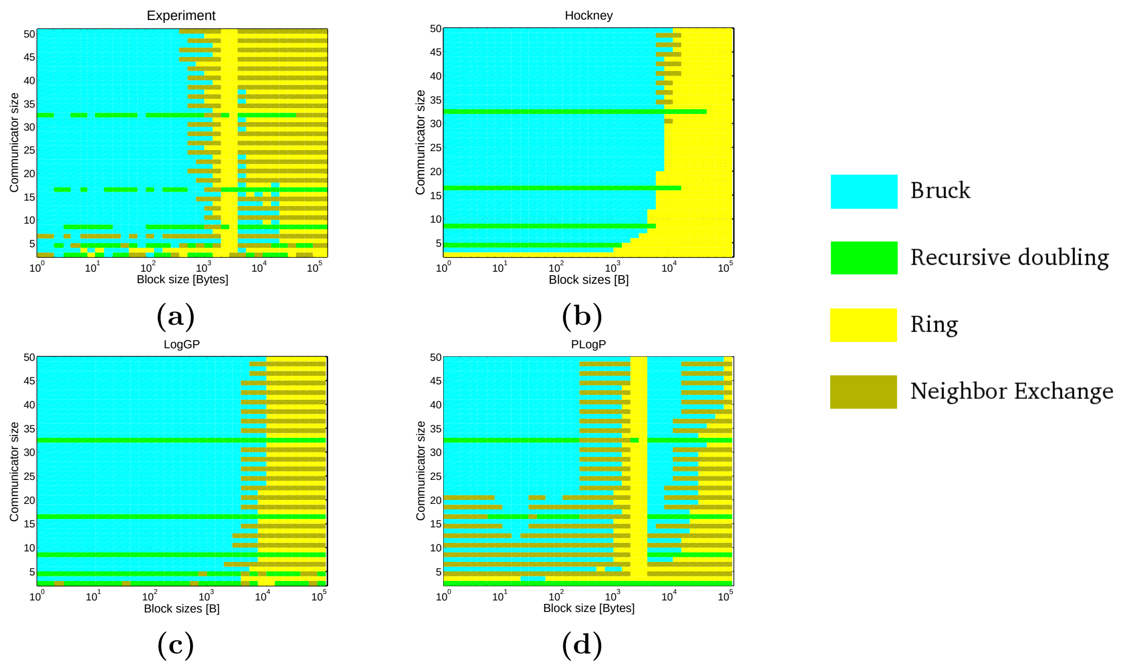


Figure 6.5: The allgather decision maps according to (a) experiment, (b) Hockney, (c) LogGP, and (d) PLogP models.

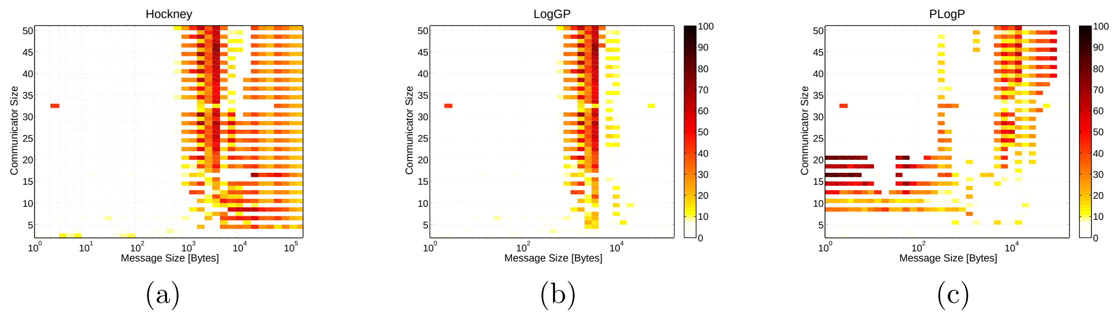


Figure 6.6: Performance penalty from using allgather decision functions generated by models (a) Hockney, (b) LogGP, and (c) PLogP. White corresponds to less than 5%, black to 100% and higher performance penalty.

Model	**Relative Performance Penalty [%]**				**Decision Time** [μsec]
	Min	**Max**	**Mean**	**Median**	
Hockney	0.00	72.37	8.29	0.00	327.89
LogP / LogGP	0.00	72.37	4.12	0.00	405.58
PLogP	0.00	85.31	5.64	0.00	2495.26

Table 6.3: Complete statistics for the performance of the model-generated allgather decision maps, corresponding to Figures 6.5 and 6.6
.

The LogGP model improves over the Hockney predictions by selecting the neighbor exchange algorithm for even communicator sizes and large block sizes. However, like Hockney decision, it switches from bruck to ring and neighbor exchange algorithms late. This results in performance penalty between 10% and 50% in the intermediate block sizes range.

PLogP model is the only one who captures the anomaly in 2 KB to 4KB block size range by selecting ring algorithm for all communicator sizes we considered. However, outside of this range, it incurs cost for using bruck for large block sizes, and neighbor exchange for small process counts and small message sizes. This region incurs between 10% and 85% performance penalty.

Overall statistics for this analysis (Table 6.3) are better than the ones reported for the broadcast implementation in Section 6.1.2 and Table 6.2. The mean performance penalty of all three models is well below 10%. However, this statistics can be misleading. As we can see in Hockney model case, even process counts can incur significant performance penalty for large block sizes, but the odd communicator sizes are not affected at all.

Finally, the time to decision for the Hockney and LogGP models are lower than in the broadcast case, mostly due to smaller number of models which need to be compared. However, the decision time for PLogP model increased because additional interpolations of model parameters (for message sizes $2^k \cdot m$.)

6.2 Quadtree encoding

In order to determine whether quadtrees are a feasible choice for encoding the automatic method selection process for MPI collective operations, we analyzed the accuracy and the performance of quadtrees built from the same experimental data but using different constraints.

Under the assumption that the collective operation's parameters are uniformly distributed across communicator size and message size space, the mean depth of the quadtree corresponds to the mean number of conditions that need to be evaluated before we can determine which method to use. In the worst case, we will follow the longest path in the tree to make the decision, and in the best case, the shortest.

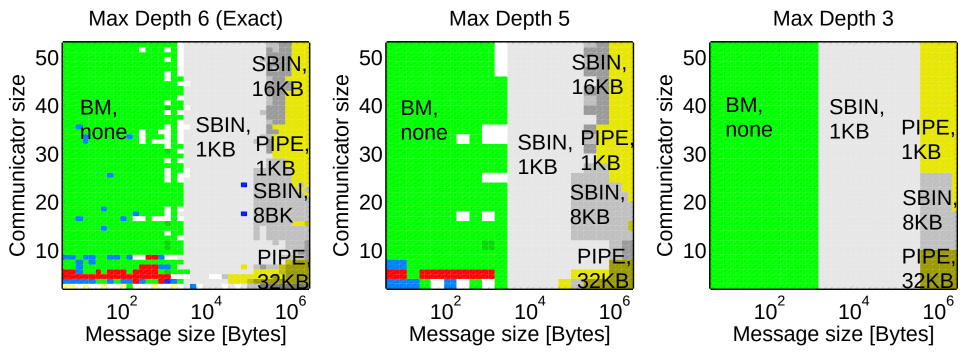
The performance data for broadcast and reduce collective algorithms was collected on the Frodo and Grig clusters located at the University of Tennessee, Knoxville. Measurements on Frodo in this paper were obtained using the MX library. On Grig, the Fast Ethernet network was used for reported measurements.

The measurements on the Frodo cluster were collected using the Open MPI version 1.3 release candidate and the SKaMPI [SKaMPI, 2005] benchmark, while the results from Grig were collected using MPICH-2 version 1.0.3 and the OCC [OCC, 2005] benchmark.

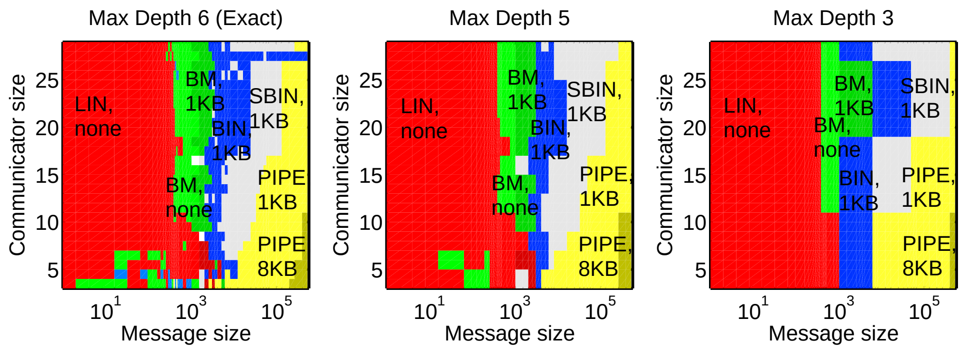
6.2.1 Broadcast decision maps

Figure 6.7 shows three different quadtree decision maps for a broadcast collective on the Frodo and Grig clusters, respectively.

We considered five different broadcast algorithms (Linear, Binary, Binomial, Split-Binary, and Pipeline), and seven different segment sizes (no segmentation, 1 KB, 8 KB, 16 KB, 32 KB, 64 KB, and 128 KB). The measurements on the Frodo cluster, Figure 6.7



(a)



(b)

Figure 6.7: Maximum-depth limited broadcast decision maps from the (a) Frodo and (b) Grig clusters. Different colors in the figures correspond to different method indexes. Both figures use the same color scheme. The trees were generated by limiting the maximum tree depth. The x-axis scale is logarithmic. The crossover line in these figures is not in the middle due to the "fill-in" points used to adjust the original size of the decision map from 49×38 and 25×48, respectively, to 64×64 form.

Tree Depth			**Performance Penalty [%]**				**Number of**	
Max	**Min**	**Mean**	**Min**	**Max**	**Mean**	**Median**	**Leaves**	**Nodes**
1	1	1.00	0.00	560.42	74.62	2.88	4	5
2	1	1.92	0.00	743.16	12.56	0.00	13	17
3	1	2.50	0.00	743.15	12.43	0.00	22	29
4	2	3.63	0.00	743.15	12.01	0.00	91	121
5	2	4.61	0.00	31.14	0.67	0.00	328	437
6	2	5.65	0.00	0.00	00.00	0.00	1153	1537

Table 6.4: Complete statistics for maximum-depth limited broadcast decision quadtrees on Frodo (Figure 6.7 (a)).

Tree Depth			**Performance Penalty [%]**				**Number of**	
Max	**Min**	**Mean**	**Min**	**Max**	**Mean**	**Median**	**Leaves**	**Nodes**
1	1	1.00	0.00	337.43	37.10	0.00	4	5
2	2	2.00	0.00	391.53	18.54	0.00	16	21
3	2	2.76	0.00	247.20	05.75	0.00	37	49
4	2	3.75	0.00	247.20	03.25	0.00	106	141
5	2	4.61	0.00	225.30	01.29	0.00	227	369
6	2	5.70	0.00	0.00	00.00	0.00	1024	1365

Table 6.5: Complete statistics for maximum-depth limited broadcast decision quadtrees on Grig (Figure 6.7 (b)).

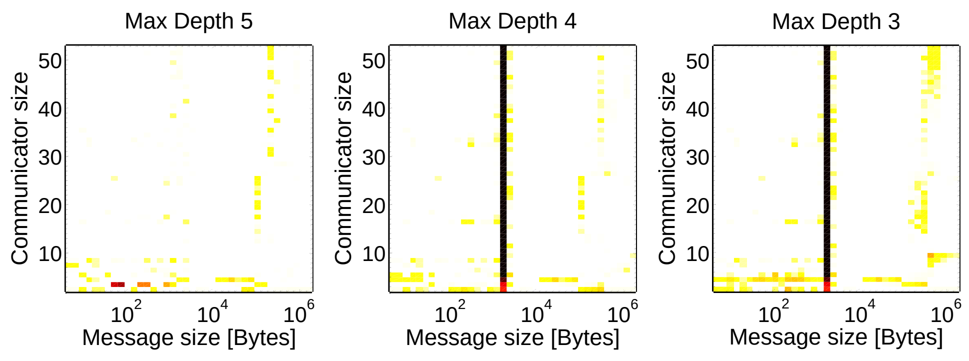
(a), covered all communicator sizes between two and 50 processes and message sizes in the 4 B to 2 MB range. On Grig, Figure 6.7 (b), results cover communicator sizes two to 28 and message sizes in the 1 B to 384 KB range.

The exact decision maps for the Frodo and Grig clusters in Figure 6.7 are quite different, mostly due to difference in characteristics of MX and FastEthernet interconnects (See Tables 4.12 and 4.13 and Figure 4.19.) Using the Myrinet interconnect (Figure 6.7 (a)), benchmarks achieve best performance using binomial tree and larger segment sizes, while over Fast Ethernet (Figure 6.7 (b)), a combination of a linear algorithm and methods with 1KB segments is a better choice. Exact decisions exhibit trends, however in both cases, there are regions with a high information density. Limiting the maximum tree depth smooths the decision map and subsequently decreases the size of the quadtree.

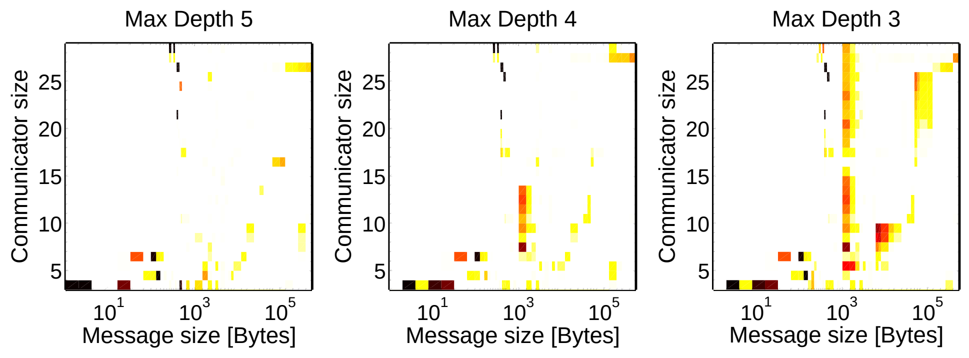
6.2.2 Performance penalty of decision quadtrees

In this section, we analyze the relative performance penalty of using a restricted quadtree instead of the exact one. Figure 6.8 shows the relative performance penalty of the decision quadtrees from Figure 6.7. Tables 6.4 and 6.5 summarize the properties and performance penalties for the same data.

The results on Grig (Figures 6.7 (b), 6.8 (b), and Table 6.5) show that a 3-level quadtree would have less than a 6% mean performance penalty. On Frodo (Figures 6.7 (a), 6.8 (a), and Table 6.4) 2-, 3-, and 4-level quadtrees have very similar performance penalty statistics, primarily due to the high performance penalty for a 1448B message on all communicator sizes greater than three. For this message size, the restricted trees use the Split-Binary algorithm with 1KB segments, instead of Binomial with no segmentation. The experimental data reveals a spike for the Split-Binary with 1KB segments for 1448B message size on all



(a)



(b)

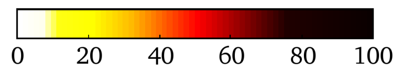


Figure 6.8: Performance penalty of maximum-depth limited broadcast decision function from (a) Frodo and (b) Grig (Figure 6.7). The colorbar represents the relative performance penalty in percentage: white means less than 5%, yellow is between 10% and 25%, red is 50% and above.

communicator sizes: measured time jumped to 300+ μs in comparison to 64+ μs for 1024B and 65+ μs for 2048B message. Moreover, all points with more than 40% performance penalty on 2-, 3-, and 4-level quadtrees were the ones for 1448B message. If we remove 1448B data points, the mean performance penalty for a 3-level tree on Frodo drops to 1.7%.

6.2.3 Quadtree accuracy threshold

In Section 5.3.2 we mentioned that an alternative way to limit the size of a quadtree is to specify the tree accuracy threshold. Figure 6.9 shows accuracy-threshold limited broadcast decision maps on the Frodo and Grig clusters. The data in these figures corresponds to the data in Figure 6.7.

Figure 6.10 shows the effect of varying the accuracy threshold on the mean quadtree depth and mean performance penalty of broadcast and reduce quadtree decision functions on the Frodo and Grig clusters. In all cases, the mean quadtree depth flattens out once a high enough accuracy threshold is achieved (from 45% for Reduce on Frodo to almost 70% for Reduce on Grig). Based on the mean performance penalty data, the trees generated with accuracy-threshold values higher than this limit are very similar to the exact tree.

6.2.4 Accuracy-threshold vs. Maximum-depth constrained trees

One of the objectives of this study is to determine which of the two methods for restricting the quadtree size gives higher-quality decision functions. Figure 6.11 shows the mean performance penalty of broadcast and reduce decisions as a function of the mean depth of the accuracy-threshold and maximum-depth constrained trees.

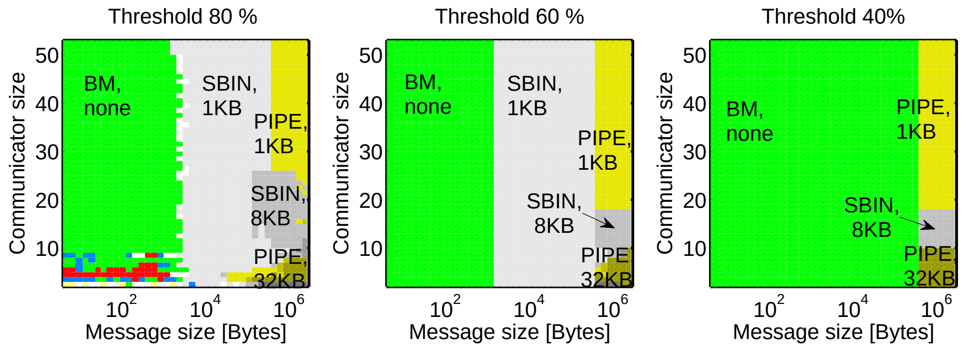
The results indicate that maximum-depth limited quadtrees achieve a lower mean performance penalty than similar accuracy-threshold limited trees. Moreover, for results with experimental data, which did not exhibit unexpected spikes (both collectives on Grig, and reduce on Frodo), maximum-depth limited trees improve their accuracy with each additional level. This is not the case for accuracy-threshold limited trees: we can see a range of mean quadtree depths corresponding to the same mean performance penalty.

6.2.5 In-memory quadtree-based decision system

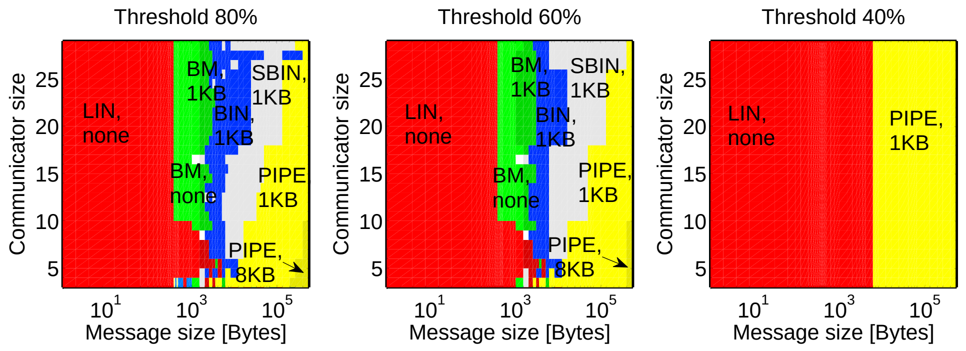
Table 6.6 contains the performance results for an in-memory quadtree-based decision system for reduce collective on the Grig cluster. The reported *decision time* value is the average time it took for the quadtree to make decisions for a random communicator and message size from the $2 - 64$ and $1 - 16MB$ ranges, respectively. All measurements were taken with the same seed value. The measurements were taken on an AMD Athlon$^{\text{TM}}$64 Processor 3500+ at 2.2GHz with 512KB cache and 1GB RAM.

In all cases, mean-time to decision of an in-memory 3-level quadtree decision system was between $50ns$ and $75ns$. The decision time for the corresponding compiled decision function was between $13ns$ and $19ns$.

In comparison, Open MPI broadcast and reduce decision functions took $18.90ns$ and $21.14ns$, respectively. The computed mean performance penalty of the Open MPI decision functions computed against the data collected on the Frodo cluster was below 20%, with a

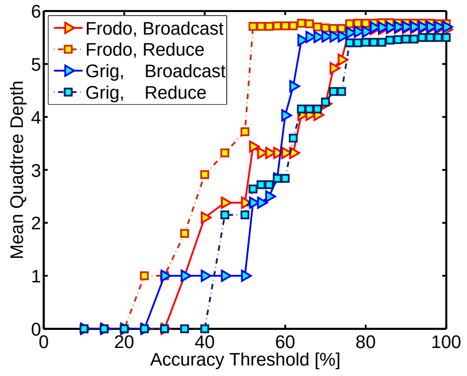


(a)



(b)

Figure 6.9: Accuracy-threshold limited broadcast decision maps from the (a) Frodo and (b) Grig clusters. In these images, the abbreviations are identical to the ones in Figure 6.7.



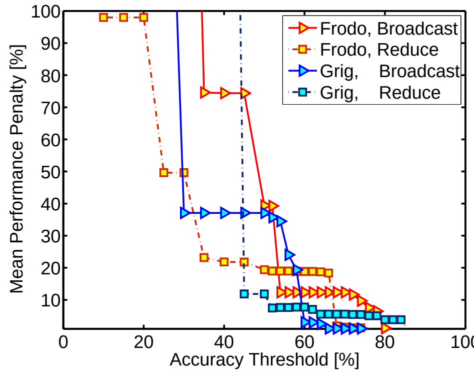


Figure 6.10: Effect of the accuracy threshold on mean quadtree depth and performance penalty.

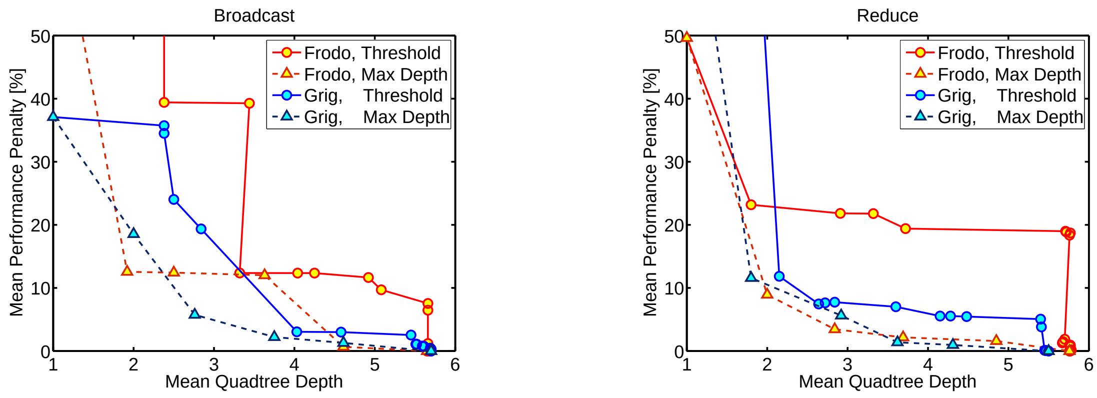
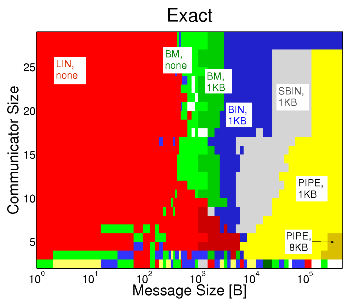


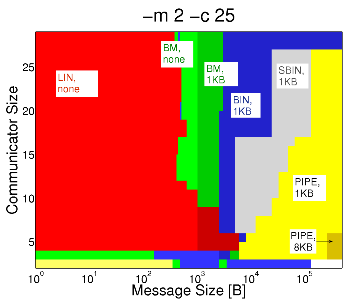
Figure 6.11: Comparison accuracy-threshold and maximum-depth quadtree constraints for broadcast and reduce collectives on Frodo and Grig cluster in terms of mean quadtree depth and mean performance penalty. Corresponds to data in Figures 6.7 through 6.10, and Tables 6.4 and 6.5.

Mean tree depth	**Mean performance penalty**	**In-memory quadtree**			**Decision function**	
		Memory size [Bytes]	**Decision time [ns]**	**Decision time per level [ns]**	**Size**	**Decision time [ns]**
1.00	81.10	712	35.61	17.81	4	11.68
1.80	11.58	1096	36.78	13.14	12	10.70
2.92	5.63	3784	54.53	13.91	68	13.68
3.62	1.39	7048	54.70	11.84	136	13.87
4.31	0.97	13192	55.02	10.36	264	13.88
5.50	0.00	42952	71.54	11.01	884	16.47

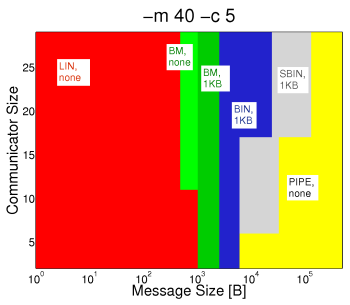
Table 6.6: Performance of the prototype implementation of an in-memory, quadtree-based decision system, and the corresponding decision function for the reduce collective on Grig. Time per level was computed as $\frac{\text{decision time}}{\text{mean depth+1}}$. Memory size includes size of the quadtree, space for communicator and message sizes, and method map. Function size is the total number of *if* and *else if* statements in the decision function source code.



(a)



(b)



(c)

Figure 6.12: Broadcast decision maps from the Grig cluster: (a) Measured (b) '-m 2 -c 25' (c) '-m 40 -c 5'. X-axis corresponds to message sizes, Y-axis represents the communicator sizes. Different colors correspond to different method indexes.

median value around 5%. This is similar to or higher than the mean performance penalty of the 3-level quadtree decision function on both systems.

6.3 C4.5 decision trees

In this work, we used release 8 of the C4.5 implementation by J.R. Quinlan [Quinlan, 2006] to construct decision trees based on existing performance data for broadcast and reduce collectives collected on the Grig cluster at the University of Tennessee, Knoxville. The performance data in this Section was collected using the MPICH-2 [MPICH2, 2002] version 1.0.3 and OCC library [OCC, 2005]. In order to use OCC results with C4.5, we had to manually convert the benchmark results to C4.5 input file format.

In our experiments, we tested decision trees constructed using different weight and confidence level constraints. We did not use windowing because our data was relatively sparse in comparison to the complete communicator - message size domain size, so we did not expect that there would be a benefit by not utilizing all available data points. Also, since communicator and message sizes were described as continuous attributes, we were not able to use the grouping functionality of C4.5.

We constructed decision trees both per-collective (e.g., just for broadcast or alltoall) and for the set of collectives that have similar or the same set of available implementations (e.g., both have Linear, Binary, and Pipeline algorithms) and for which we expected to have similar decision functions (e.g., broadcast and reduce).

6.3.1 Analysis of broadcast decision trees

Figure 6.12 shows three different decision maps for a broadcast collective on the Grig cluster using FastEthernet interconnect. We considered five different broadcast algorithms (Linear, Binomial, Binary, Split Binary, and Pipeline) and four different segment sizes (no segmentation, 1 KB, 8 KB, and 16 KB). The measurements covered all communicator sizes between two and 28 processes and message sizes in the 1 B to 384 KB range with total of 1248 data points. The original performance data set contains $1248 \times 4 \times 5$ data points.

Command line	Before pruning		After pruning			Performance penalty		
	Size	Errors	Size	Errors	Predicted Error	Min	Max	Mean
-m 2 -c 25	133	7.9%	127	7.9%	14.6%	0%	75.41%	0.66%
-m 4 -c 25	115	8.8%	95	9.4%	15.0%	0%	316.97%	1.16%
-m 6 -c 15	99	10.4%	65	11.5%	17.6%	0%	316.97%	3.24%
-m 8 -c 5	73	12.0%	47	12.8%	21.0%	0%	316.97%	1.66%
-m 40 -c 5	21	17.8%	21	17.8%	21.9%	0%	316.97%	2.08%

Table 6.7: C4.5 broadcast decision tree statistics corresponding to the data presented in Figure 6.12. Size refers to the number of leaf nodes in the tree. Errors are in terms of misclassified training cases. The data set had 1248 training cases. The median performance penalty was 0% in all cases.

Figure 6.12 (a) shows an exact decision map generated from experimental data. The subsequent maps were generated by C4.5 decision trees constructed by specifying different values for weight (“`-m`”) and confidence level (“`-c`”) parameters (See Section 5.4.1). The statistics about these and additional trees can be found in Table 6.7.

The exact decision map in Figure 6.12 (a) exhibits trends, but there is a considerable amount of information for intermediate size messages (between 1 KB and 10 KB) and small communicator sizes. The decision maps generated from different C4.5 trees capture general trends very well. The amount of captured detail depends on weight, which determines how the initial tree will be built, and confidence level, which affects the tree pruning process. “Heavier” trees require that branches contain more cases, thus limiting the number of fine-grained splits. A lower confidence level allows for more aggressive pruning, which also results in coarser decisions.

Looking at the decision tree statistics in Table 6.7, we can see that the default C4.5 tree (“-m 2 -c 25”) has 127 leaves and a predicted misclassification error of 14.6%. Using a slightly “heavier” tree “-m 4 -c 25” gives us a 25.20% decrease in tree size (95 leaves) and maintains almost the same predicted misclassification error. As we increase tree weight and decrease the confidence level, we produce the tree with only 21 leaves (83.46% reduction in size) with a 50% increase in predicted misclassifications (21.9%).

In this work, the goal is to construct reasonably small decision trees that will provide good run-time performance of an MPI collective of interest. Given this goal, the number of misclassified training examples is not the main figure of merit we need to consider. To determine the “quality” of the resulting tree in terms of collective operation performance, we consider the performance penalty of the tree. The performance penalty is the relative difference between the performance obtained using methods predicted by the decision tree and the experimentally optimal ones.

The last three columns in Table 6.7 provide the performance penalty statistics for the broadcast decision trees we are considering. The minimum, mean, and median performance penalty values are rather low - less than 4%, even as low as 0.66%, indicating that even the simplest tree we considered should provide good run-time performance. Moreover, the simplest tree, “-m 40 -c 5”, had a lower performance penalty than the “-m 6 -c 15,” which indicates that the percent of misclassified training cases does not translate directly into a performance penalty of the tree.

Command line	Before Pruning		After Pruning			Mean performance penalty	
	Size	Errors	Size	Errors	Predicted error	Broadcast	Reduce
-m 2 -c 25	239	137	221	142	12.6%	0.66%	0.41%
-m 6 -c 25	149	205	115	220	14.0%	1.62%	0.71%
-m 8 -c 25	127	225	103	235	14.4%	1.64%	0.72%
-m 20 -c 5	63	310	55	316	20.6%	2.40%	0.93%
-m 40 -c 25	33	392	33	392	19.6%	2.37%	1.53%

Table 6.8: Statistics for combined broadcast and reduce decision trees corresponding to the data presented in Figure 6.13. Size refers to the number of leaf nodes in the tree. Errors are in terms of misclassified training cases. The data set had 2286 training cases.

In all cases, the mean and median performance penalty values are excellent, but the maximum performance penalty of 316.97% requires explanation. At communicator size 25 and message size 480, the experimentally optimal method is Binary algorithm without segmentation (1.12 *ms*), but most decision trees select Binomial algorithm without segmentation (4.69 *ms*). However, the Binomial algorithm performance in the neighborhood of this data point is around and less than 1 *ms*, which implies that the 4.69 *ms* result is probably affected by external factors. Additionally, in the "-m 40 -c 5" tree, only six data points had a performance penalty above 50%.

6.3.2 Combined decision trees

It is reasonable to expect that similar MPI collective operations have similar decision functions on the same system. To test this hypothesis, we decided to analyze the decision trees generated from the experimental data collected for broadcast and reduce collectives on the Grig system. Our implementations of these collectives are symmetric; each of them has Linear, Binomial, Binary, and Pipeline based implementations. Broadcast supports the Split Binary algorithm for which we do not have an equivalent in reduce implementation, but we expect that C4.5 should be able to handle these cases correctly.

The training data for this experiment contains three attributes (collective name, communicator size, and message size) and the same set of predetermined classes as in the broadcast-only case.

Figure 6.13 shows the decision maps generated from the combined broadcast and reduce decision tree. The leftmost maps in both rows are the exact decisions for each of the collectives based on experimental data. The remaining maps are generated by querying the combined decision tree. Figures 6.13 (b) and (e) were generated using a "-m 2 -c 25" decision tree, while (c) and (f) were generated by a "-m 20 -c 5" decision tree. Table 6.8 provides the detailed information about the combined decision trees of interest including the mean performance penalty of the trees.

The structure of combined broadcast and reduce decision trees reveals that the test for the type collective occurs for the first time on the third level of the tree. This implies that the combined decision tree is able to capture the common structure of the optimal implementation for these collectives, as one would expect based on decision maps in Figure 6.13.

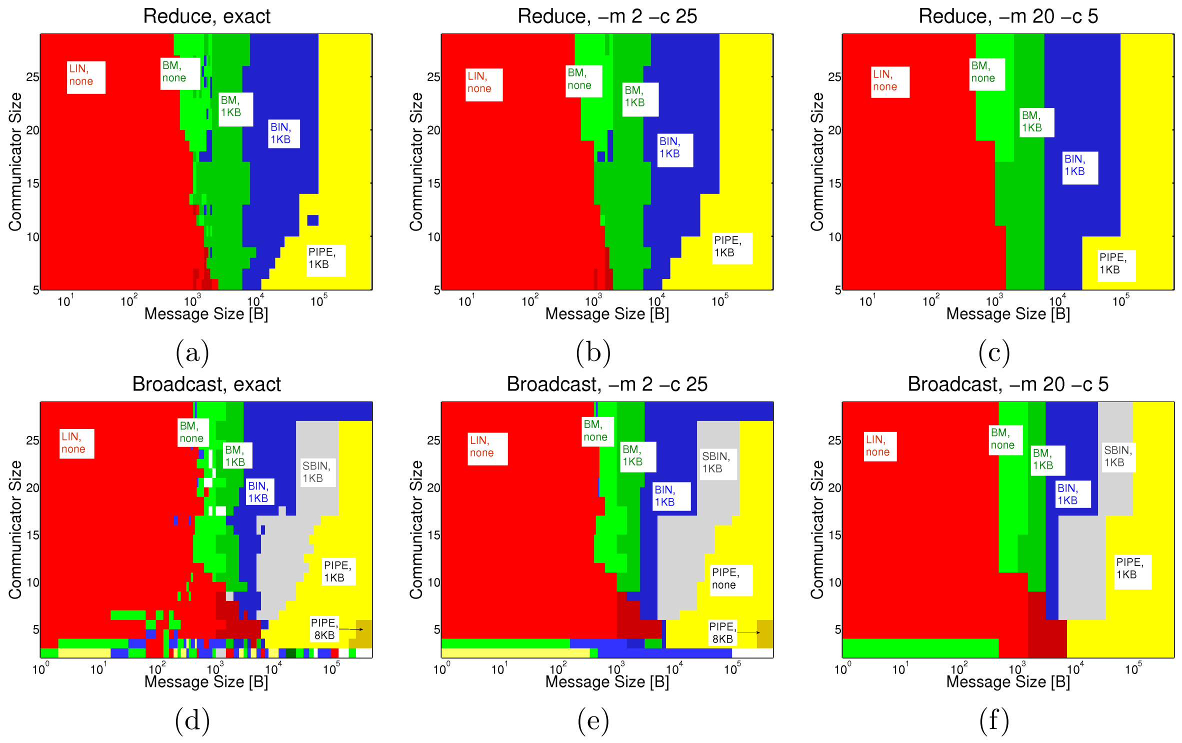
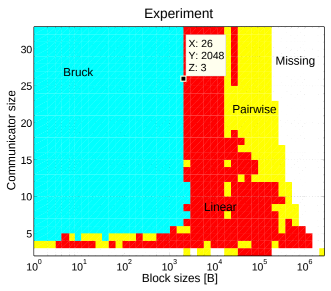
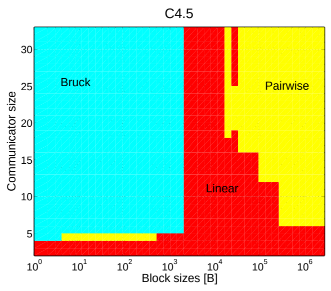


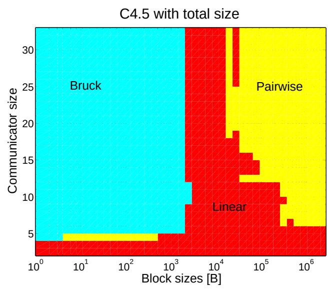
Figure 6.13: Combined broadcast and reduce decision maps from the Grig cluster: (a) reduce, Exact (b) reduce, '-m 2 -c 25' (c) reduce, '-m 20 -c 5' (d) broadcast, Exact (e) broadcast, '-m 2 -c25' (f) broadcast, '-m 20 -c 5'.



(a)



(b)



(c)

Figure 6.14: Effect of constructive induction on Alltoall decision function: (a) experiment (b) c4.5 (c) c4.5 with “total size” composite attribute.

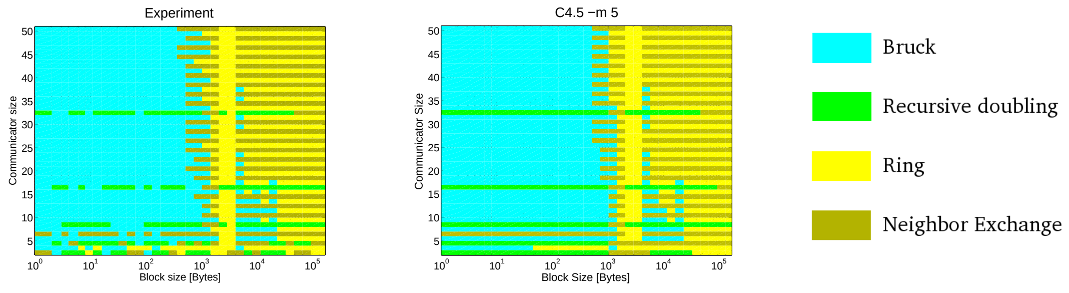
6.3.3 Constructive induction and composite attributes

The *constructive induction* is a machine learning technique in which a new attribute is formed as a function of the existing attributes. This allows learning algorithm to benefit from the domain knowledge and to capture a possibly complex dependence between the base attributes. The new attribute is referred to as *composite* attribute. In case of MPI collective operations, the composite attributes such as “total message size” for all-to-all type of algorithms are natural extension of communicator / message size attribute space. In addition, one can introduce additional attributes to describe “even / odd” or “power-of-two” communicator sizes. In this section we briefly explore the effect composite attributes have on accuracy of C4.5 decision trees for optimal collective implementation.

Composite attributes and alltoall decision trees

The results in this section were collected on Cisco cluster with Infiniband interconnect and quad processor nodes (8 nodes by 4 CPUs each). Figure 6.14 shows alltoall decision maps from this experiment.

The experimental decision map (Figure 6.14 (a)) has a well defined switching point between bruck and linear algorithm: 2 KB block size. Also, for communicator sizes above 20, linear algorithm should be replaced by pairwise exchange once the block size exceeds 16 KB. However, for smaller communicator sizes the switching point is not so clear: there appears to exist a linear dependence between the block and communicator size and the switching point between linear and pairwise exchange algorithm in this range. Applying C4.5 algorithm directly to this problem we obtain decision map in Figure 6.14 (b). The small communicator size - large message size region is divided in rectangular regions and while it does cover the original data well, the linear dependence became more like step function. However, using the “total message size” composite attribute (Figure 6.14 (c)), the linear dependence between 13 and 18 communicator sizes and 16 KB and 100 KB block sizes is captured.



(a) **(b)**

Figure 6.15: Effect of constructive induction on Allgather decision function: (a) experiment (b) c4.5 -m 5 with composite attributes.

Composite attributes and allgather decision trees

In order to further examine usefulness of composite attributes in this setting, we reconsider allgather implementation on Grig cluster using MX interconnect (See Section 6.1.3 for detailed analysis using parallel communication models.)

The experimental decision function (Figure 6.15 (a)) is especially complex to capture automatically since the notion of even or power-of-two communicator sizes is hard to discover as a pattern. The C4.5 decision tree built using only communicator and message size as attributes contains 179 leaves, has 9.3% misclassification rate on training set, and estimates the classification error on unseen cases to be 16.9%. Based on results from Section 6.3.1 this may be well performing tree, however, there is a better way to build decision tree in this case.

To address this issue, we introduce additional attributes to describe each data point: *total message size*, *even communicator size*, and *power-of-two communicator size*. The later two are binary attributes whose value can be "yes" or "no" depending on the value of the communicator size. Figure 6.15 shows (a) experimental and (b) c4.5 generated decision maps for allgather on the grig cluster. The c4.5 decision map was generated using "-m 5" switch.

The decision maps in Figure 6.15 are almost identical. The mean performance penalty of this map is 0.2% with maximum at 42.55% for communicator size 32 and block size 2 B. There is only six data points with relative performance penalty higher than 10%. The proposed decision tree consists of 59 leaves, has 4.9% misclassification error on training set, and estimates the error rate on unseen cases to be 7.78%. The internal structure of the tree reveals that after first split on message size of size 724 B, the next three levels of the tree are tests on either "even", "power-of-two", or "total message size" attributes. The maximum depth of the tree was 9 levels with average depth around 5.5.

6.4 Large scale results

This section provides the summary and analysis of the performance results for different barrier, broadcast, reduce, and allreduce algorithms available in Open MPI collected on

Thunderbird system at Sandia National Laboratory [Thunderbird, 2006]. The Thunderbird is a Dell PowerEdge 1850 system, with 4, 480 compute nodes each equipped with dual Intel^{TM}EM64T Xeon®3.6 GHz processors. Each of the compute nodes has 6 GB RAM. The system is connected using Infiniband high-performance interconnect. The system achieved 53TFlops when run with 9, 024 processes and ranked as high as number 6 on Top 500 list in 2006.

Jeff Squyres from Cisco, collected the performance of results using SKaMPI-5.0.1 [SKaMPI, 2005] benchmark. In addition, results of NetPIPE [NetPIPE, 2005] and LogP [Kielmann et al., 2000] benchmarks were obtained. Most of the algorithm performance data was collected on up to 1024 processes, although some of the measurements went up to 4096 processes.

The purpose of this section is to emphasize some of the issues that occur when running applications (including benchmarks) at large scale. The scalability of collective operations on this scale is an issue, but the stability of the system is even larger cause of concern. Even though the originally assigned time was theoretically enough to complete the measurements, not all values could be collected due to problems with Open MPI implementation, network issues, and system crashes. The detailed results and discussion of the issues are presented in [Pješivac-Grbović, 2007].

6.4.1 Point-to-point performance

The point-to-point communication performance was measured using NetPIPE [NetPIPE, 2005] and LogP [Kielmann et al., 2000] benchmarks.

NetPIPE

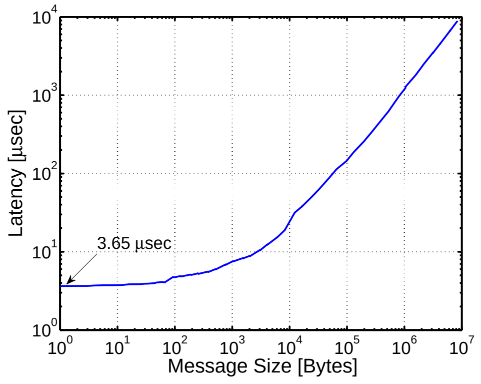
The NetPIPE [NetPIPE, 2005] benchmark is used to measure the latency and transfer time of the point-to-point communication. The results of this benchmark can be directly used as parameters for Hockney parallel communication model [Hockney, 1994]. Figure 6.16 shows the latency and transfer time measurements on this system. The latency was measured to be 3.65 μsec and the maximum bandwidth achieved was 7222.68 Mbps, which equates to transfer time of $0.0011 \frac{\mu sec}{Byte}$.

LogP benchmark

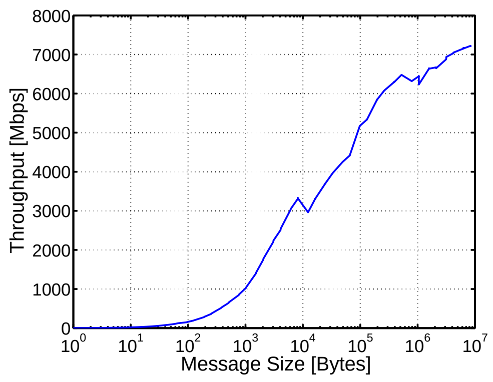
The LogP benchmark is used to directly measure parameters of PLogP parallel communication model [Kielmann et al., 2000]. The PLogP model can be used to predict parallel algorithm performance directly, or the values of its parameters can be used to estimate the values of LogP and LogGP model parameters [Culler et al., 1993, Alexandrov et al., 1995].

Figures 6.17 shows the measured values of PLogP model parameters and values of LogP and LogGP model computed from these values, respectively. In the original measurement, the latency value was not reported because benchmark output format rounded approximately 3×10^{-6} value to 0.0. Thus we used the NetPIPE results to estimate latency value.

In addition, we estimate computation-time-per-byte from “bogomips” statistics for Thunderbird's CPUs. Each of the two processors on the node achieves around 7200 (bogus) MIPS. This equates to approximately $6.96 \times 10^{-11} \frac{sec}{instr}$ which we turn to $1.75 \times 10^{-5} \frac{\mu sec}{byte}$ for integer operations.



(a)



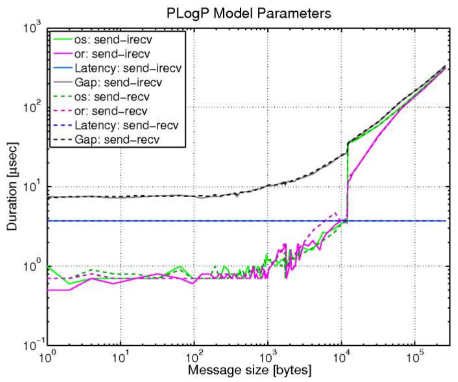
(b)

Figure 6.16: NetPIPE results on Thunderbird (a) Latency and (b) Bandwidth.

LogP / LogGP model parameters

Latency	3.65 μsec
Overhead	0.65 μsec
Gap	6.7 μsec
Gap per byte	0.0013 $\frac{\mu sec}{Byte}$
Computation time per Byte	1.75×10^{-5} $\frac{\mu sec}{Byte}$

(a)



(b)

Figure 6.17: (a) LogP / LogGP (b) PLogP model parameters on Thunderbird.

6.4.2 Collective operation performance

The collective operation algorithm performance was collected using SKaMPI 5.0.1 benchmark [SKaMPI, 2005].

With no previous information about the algorithm performance on large-scale systems, we decided to limit sets of measurements to the ones we believed would perform the best. In addition, we limited the maximum number of repetitions per data point to 35. We hoped this restriction will not cause large variability in measured data. These two restrictions allowed us to keep measurement time under control.

In addition, to support large-scale measurements, we had to modify SKaMPI benchmark to limit the verbosity of its output. By default, SKaMPI reports "measured time", "standard deviation of the measurement", "number of measurements", and individual times measured on each of the processes involved in the measurement. This amounts to rather large files even on 256 process test. Thus, we modified SKaMPI not to report individual timers.

Most of the tests were split in 9 different groups based on communicator and message size:

- communicator sizes
  - small: 64, 128
  - intermediate: 256, 512, 768, 1024
  - large: 1536, 2048, 2560, 3072, 3684, 4096, 4400
- message sizes
  - small: 8B, 128B, 1KB, 8KB
  - intermediate: 64KB, 256KB
  - large: 1MB, 8MB

We selected a separate set of methods (algorithm and segment size combinations) for each of the communicator × message size combinations. The following subsections detail the measurement results for barrier, reduce, and allreduce. The rest of the results are analyzed in [Pješivac-Grbović, 2007].

Barrier

In this test, we measured performance of linear, recursive doubling, and bruck algorithm. Figure 6.18 shows the absolute algorithm performance with error bars denoting the standard deviation of the measurement. As expected, the linear algorithm performance is drastically worse than the performance of bruck and recursive doubling algorithms. However, the performance of the recursive doubling algorithm is surprisingly better than the performance of the bruck algorithm for intermediate communicator sizes (350% for 1024 communicator size). For large communicator sizes (1536 and 2048), the performance of two algorithms is within the standard deviation of the measurement. Even so, the bruck algorithm seems to outperform recursive doubling on 2048 processes. Based on the results, it is hard to determine which algorithm would be better option.

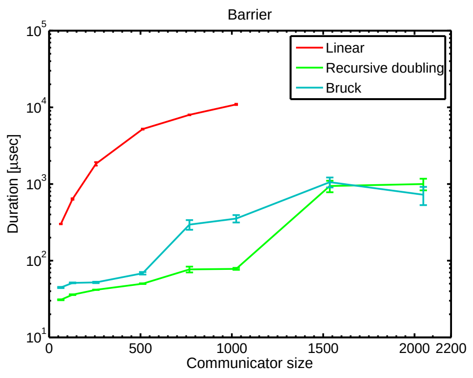


Figure 6.18: Performance of different barrier algorithms with error bar denoting standard deviation of measurement.

Reduce

The reduce performance was measured for small and intermediate communicator sizes (up to and including 1024 processes). We collected performance information for the linear, binary and binomial algorithms without segmentation, and binary, binomial, and pipeline algorithms with 32 KB, 128 KB and 512 KB segments. The attempt to measure performance of segmented algorithms with 8 KB segment sizes failed for most methods due to a previously undetected bug in the byte sending level of Open MPI. The only measurement with 8 KB segments are binary and binomial algorithms for 64 KB message. Apart from this, the reduce algorithm measurements were rather stable, only six data points out of 218 had standard deviation higher than 10%.

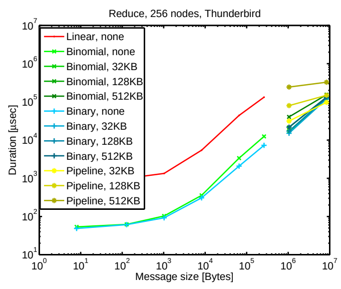
Figure 6.19 shows the absolute performance of different reduce methods we considered. Based on pipeline results, we can conclude that the smaller segment sizes yield better performance. The segmented pipeline algorithm has unique property that it asymptotically achieves constant running time for large enough message sizes. We can see this phenomenon occurring on 768 and 1024 processes, however, the level of the line is too high. Smaller segment size (such as 8KB) would possibly achieve better performance and become method of choice instead of binary algorithm with 32 KB segments for at least some cases.

Figure 6.20 shows the measured and model predicted decision maps for reduce on Thunderbird system. Models returned the best decision function given the algorithms experiments considered.

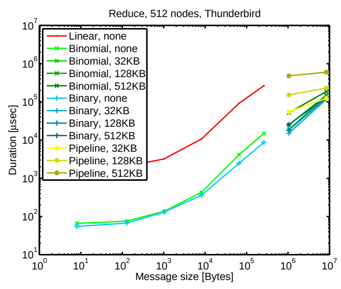
The model predictions are consistent with our segment size analysis based on absolute performance: using 8 KB segment size could improve overall performance of an algorithm. Moreover, based on the absolute performance of non-segmented binomial and binary algorithm, we can say that using binomial instead of binary algorithm does not affect reduce performance significantly (except in case of 1536 communicator size).

Allreduce

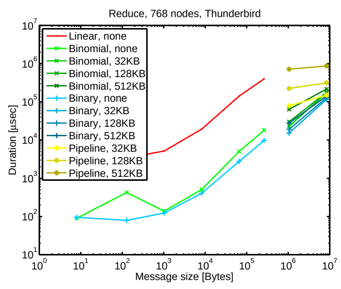
We measured performance of the recursive doubling allreduce algorithm on range of communicator and message sizes. The measurements covered all message sizes for small and



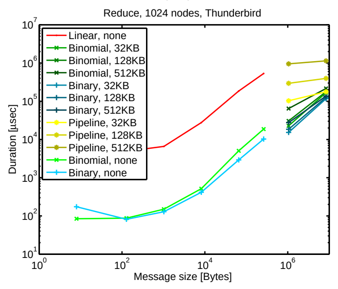
(a)



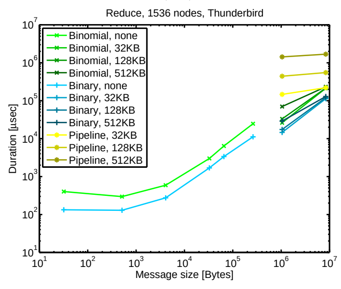
(b)



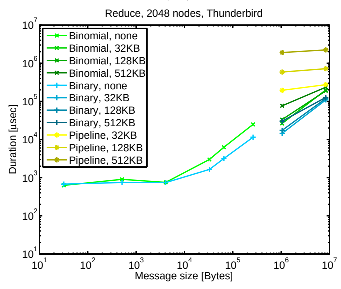
(c)



(d)

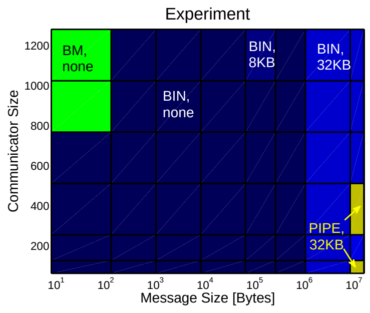


(e)

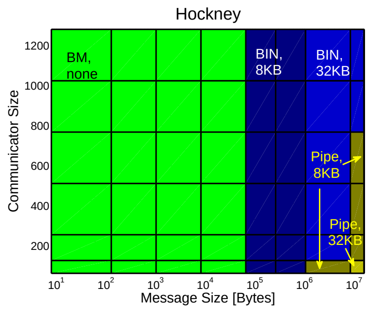


(f)

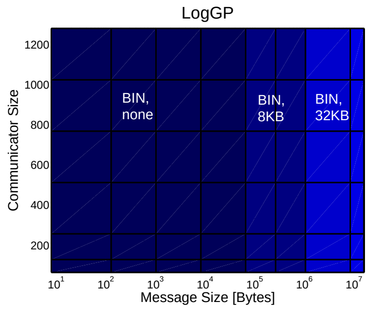
Figure 6.19: Absolute performance of reduce methods on Thunderbird on (a) 256, (b) 512, (c) 768, (d) 1024, (e) 1536, and (f) 2048 processes.



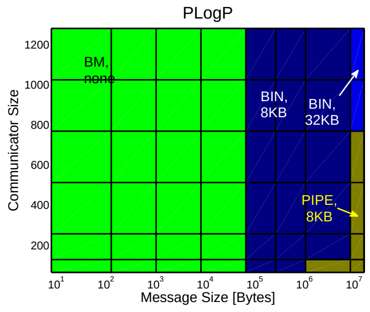
(a)



(b)

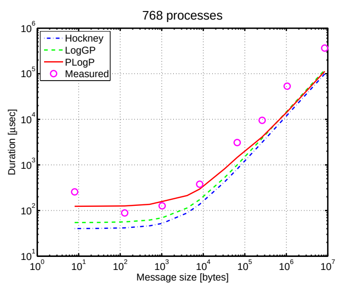


(c)

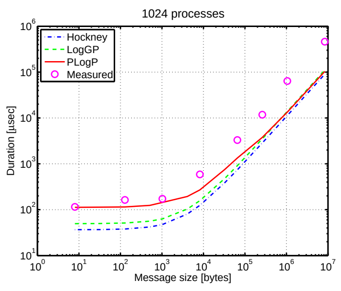


(d)

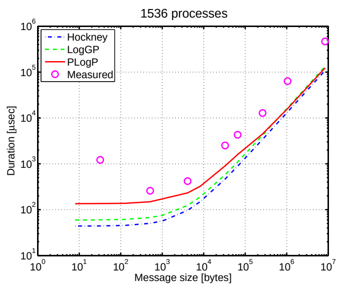
Figure 6.20: Reduce decision maps for Thunderbird system: (a) measured, and (b) Hockney, (c) LogGP, and (d) PLogP model based.



(a)



(b)



(c)

Figure 6.21: Performance of recursive doubling allreduce algorithm on (a) 768, (b) 1024, and (c) 1536 processes on Thunderbird.

intermediate communicator sizes, and only large message sizes for large communicator sizes. Figure 6.21 shows the absolute performance of recursive doubling allreduce on 768, 1024, and 1536 processes, along with model predictions.

The model accuracy in this case is disappointing and gets progressively worse as the process count increases. PLogP models is the only model which at least partially follows the measured performance for small message sizes. For large message sizes, the dominant factor in the performance model becomes bandwidth, and all three models have approximately the same value for this parameter, and thus predictions of all three models converge. The most probable reason for these result are unexpected overheads at lower levels when running such a large job and contention at the switch.

6.5 Comparison of three approaches

Sections 6.1, 6.2, and 6.3 present a detailed experimental analysis of the three different approaches to the algorithm selection optimizations: analytical performance models, graphical encoding methods (quadtree encoding), and statistical learning methods (decision trees), respectively. In addition, Section 6.4 presents results of the large scale performance tests and the data analysis using models.

Parallel communication models provide an elegant and relatively simple way to predict a method performance. Selecting the best method in this case is equivalent to finding a method with the smallest predicted time among the group of the available methods. In addition, models can help us determine properties such as optimal segment size for an algorithm. Accuracy is the main limitation of the models. Among the three basic point-to-point communication models we considered, Hockney, LogGP, and PLogP, the Hockney model performed worst in terms of predicting both absolute and relative collective method performance. At the same time, this is the most frequently used model in MPI community. Both LogGP and PLogP models were, in general, capable of capturing performance of most algorithms. The LogGP models are more complex than the Hockney-based ones, but still provide intuitive description of an algorithm. The major limitation of LogGP approach is the fact that all model parameters are constants, and as such, this model is unable to capture non-linear behaviors. As we demonstrated in Section 4.4.2, this is the main reason it sometimes fails to predict that segmentation can improve an algorithm performance. The PLogP model addresses the "linearity" issue by using parameters that depend on the message size. As such, the PLogP model parameters are capable of capturing non-linear behavior of a collective operation. However, this means that PLogP models are harder to interpret than Hockney- and LogGP- based models, as the performance is a function of a parameter which depends on the message size (e.g. $g(m)$) instead of a value times m. In addition, the PLogP models are hardest to apply in decision function because the parameter values need to be estimated for the messages size in which they were not measured. This restriction results in the PLogP decision function having longest time-to-decision in comparison to Hockney and LogGP models. In terms of the mean performance penalty, either LogGP or PLogP models can produce decision functions which are accurate enough (less than 10% mean performance penalty with median of 0%.) As we implemented only prototype implementation in Matlab, the feasibility of including models in an MPI implementation directly is still an open question. The large scale testing showed that, while not perfect, models are the most functional tool for analyzing performance of sparse data. Detailed profiling of large scale system is often impossible, and thus only number of method performance data points can be collected. The sparsity of the performance data limits the applicability of both graphical encoding and statistical learning methods.

The results from Section 6.2 indicate that the quadtree encoding is a fast and feasible way to generate platform-specific decision functions automatically. In addition, the size of quadtree, and thus the size and complexity of resulting decision function, is easily manipulated. By limiting the maximum tree depth, one can set the maximum number of expressions that need to be evaluated in order to reach the decision. Using the accuracy threshold constraint the quadtree size can be dynamically adjusted such that information dense regions are covered with deeper branches, but flat regions are turned to leaves quickly. In the data sets we considered, quadtrees with both maximum and mean depth of three levels incurred less than 12% performance penalty. The complete data sets could be covered with six-level quadtree. While the results are promising, the main restriction of this approach is lack of flexibility. Expanding approach to multiple dimensions is possible, but the resulting data would be even more sparse and the resulting trees would contain superfluous tests. In addition, in order to construct quadtrees, the input data requires preprocessing step to convert it to a power-of-two square matrix. This step affects the efficiency of the encoding. Improving the efficiency of the preprocessing step to increase number of square regions is equivalent to searching for patterns in the original data, which is the problem equivalent to the one we are trying to solve using quadtrees in the first place.

The decision functions generated by C4.5 decision trees have the highest accuracy among the methods we considered. Similar to quadtrees, C4.5 decision trees split the input parameter space into (hyper-)rectangular regions. Unlike quadtree encoding, the C4.5 decision trees have no restrictions on shape and form of the input parameter space. In addition, the C4.5 decision trees can handle multi-dimensional data automatically. This property can be used to determine the common properties of decision functions of similar collectives, as we demonstrated in Section 6.3.2 where we combined the experimental results from broadcast and reduce operations on the same system. As expected, the test revealed that broadcast and reduce decision functions share significant amount of information. This implies that one can use the decision function for one of the collectives to construct reasonably good decision function for the other. Moreover, the additional attributes can allow C4.5 decision trees to capture rather complex patterns: such as methods which perform well only for power-of-two communicator sizes or methods whose efficiency depends on the total data size instead of just block size (See Section 6.3.3). However, unlike quadtrees whose size can be easily manipulated, the depth of the C4.5 decision tree is hard to estimate, making it impossible to set an a priori limit on the maximum number of expressions to be evaluated in the final decision function. Thus, run-time performance of this approach is harder to control.

6.6 Case study: Platform-specific collective tuning for FastEthernet

In this Section we analyze the potential for the collective operation performance improvement by exploring the platform-specific optimizations. Both MPICH 2 and Open MPI are by default, tuned for fast interconnects such as Myricom's MX or Infiniband. However, there are still systems with slower interconnects which are being used and both MPICH 2 and Open MPI achieve suboptimal performance by default. In this Section, we try to improve the broadcast performance on the Grig cluster using FastEthernet.

We approach this problem in the following manner: first, we measure the performance of different methods available in Open MPI using SKaMPI benchmark;* second, we measure the performance of the native MPI broadcast in MPICH 2 and Open MPI using different SKaMPI setup; third, we compute the experimentally optimal decision map and use it to build the C4.5- and quadtree- based decision functions; fourth, we replace the default broadcast decision function in Open MPI by the ones suggested by C4.5 and quadtrees; and finally, we repeat the SKaMPI benchmark measurements with the new decision functions. We could not test parallel communication models in this settings because they are implemented using Matlab.

We measured the performance of linear, binary, split-binary, and binomial algorithms without segmentation, and pipeline, binary, split-binary, and binomial algorithm with 1 KB, 3 KB, 8 KB, 16 KB, 32 KB, 64 KB, and 128 KB for message sizes in range 1 MB to 2 MB. The experimentally optimal decision function was almost identical to the one displayed in Sections 6.1.2, 6.2.1, 6.3.1. The native implementation performance was evaluated in two ways: on 24 and 64 processes for large number of message sizes in range from 1 B to 65 MB; and on all communicator sizes between two and 64 and message sizes 8B, 128B, 1000B, 60000B, 100000 B, and 10000000 B.

This data was used to construct the default C4.5 decision tree, the C4.5 decision tree with "total_data_distributed" composite attribute, and quadtrees with maximum depths 4 and 3. Figures 6.22 (a) and (d) show the absolute performance of MPICH 2 and Open MPI native broadcast implementation on 24 and 64 processes respectively. The remaining plots in this figure show the relative performance by using the platform-specific decision functions instead of default ones. Table 6.9 provides the performance improvement statistics about the runs. Results show that both MPICH 2 and Open MPI have similar performance for small

*As we measure the performance of methods available in Open MPI, we did not reuse broadcast performance information collected using OCC benchmark, but we repeated the measurements using SKaMPI.

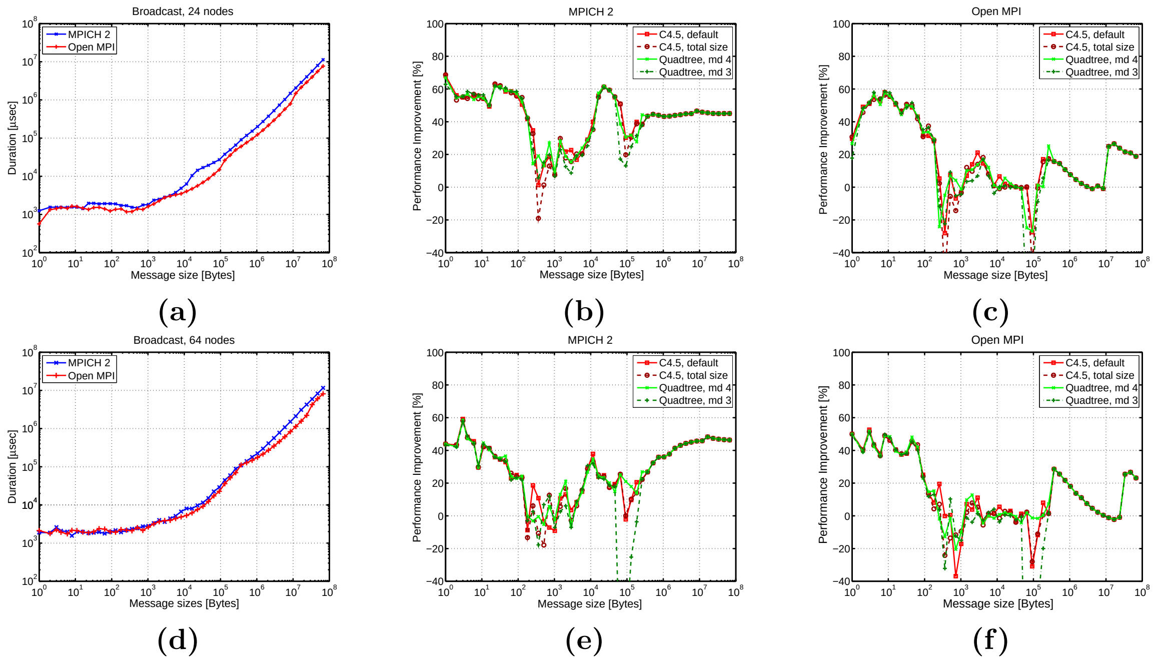


Figure 6.22: Performance improvement over native (b), (e) MPICH 2 and (c), (f) Open MPI Broadcast implementation on (a), (b), (c) 24 and (d), (e), (f) 64 processes.

message sizes. For large message sizes, Open MPI outperforms MPICH 2 by approximately 40%. Depending on the decision function, the mean performance improvement over MPICH 2 broadcast is between 22.59% and 42.85%. The performance improvement over Open MPI is less impressive but still significant: anywhere between 6.01% through 44.96%.

The most significant value of these results is that the platform-specific decision functions performed similarly in terms of mean performance improvement. Looking at the final version of decision functions, C4.5 decision functions had around 150 lines, while the 4-level quadtree had close to 600 lines. The 3-level quadtree had around 190 lines. The fact that 4-level quadtree had even better mean performance improvement over Open MPI than the C4.5 trees indicates that the evaluation of 600-line decision function is not significantly slower than evaluation the 150-line function.

Figure 6.23 shows broadcast results on all communicator sizes up to 64 and two message sizes: 128 B and 10 MB. Starting with the communicator size of 10, the platform-specific decision functions outperform MPICH 2 implementation between 5% and 55%. The trend is similar as far as Open MPI is concerned as well. For large message size, all four methods select the same algorithm and the performance improvement is uniform. Surprisingly, the measured duration for 10 MB message size exhibits a strong non-linear behavior both using MPICH 2 and Open MPI.

Decision	Comm Size	Performance improvement over Open MPI			Performance improvement over MPICH 2		
		Min	Max	Mean	Min	Max	Mean
C4.5	24	−28.06%	58.05%	17.80%	1.30%	68.17%	42.85%
C4.5 w/ts	24	−54.44%	56.30%	16.35%	−19.03%	68.70%	41.70%
QT md 4	24	−26.85%	57.54%	17.27%	9.28%	66.81%	42.56%
QT md 3	24	−68.43%	58.49%	14.68%	5.69%	62.84%	41.00%
C4.5	64	−36.85%	52.63%	14.17%	−9.13%	59.22%	28.40%
C4.5 w/ts	64	−27.99%	51.42%	13.36%	−17.87%	58.17%	27.58%
QT md 4	64	−20.42%	51.66%	14.61%	−6.16%	58.38%	28.71%
QT md 3	64	−161.34%	50.70%	6.92%	−99.46%	57.56%	22.59%

Table 6.9: Performance improvement of platform-specific broadcast implementation over MPICH 2 and Open MPI on 24 and 64 processes. Positive is better.

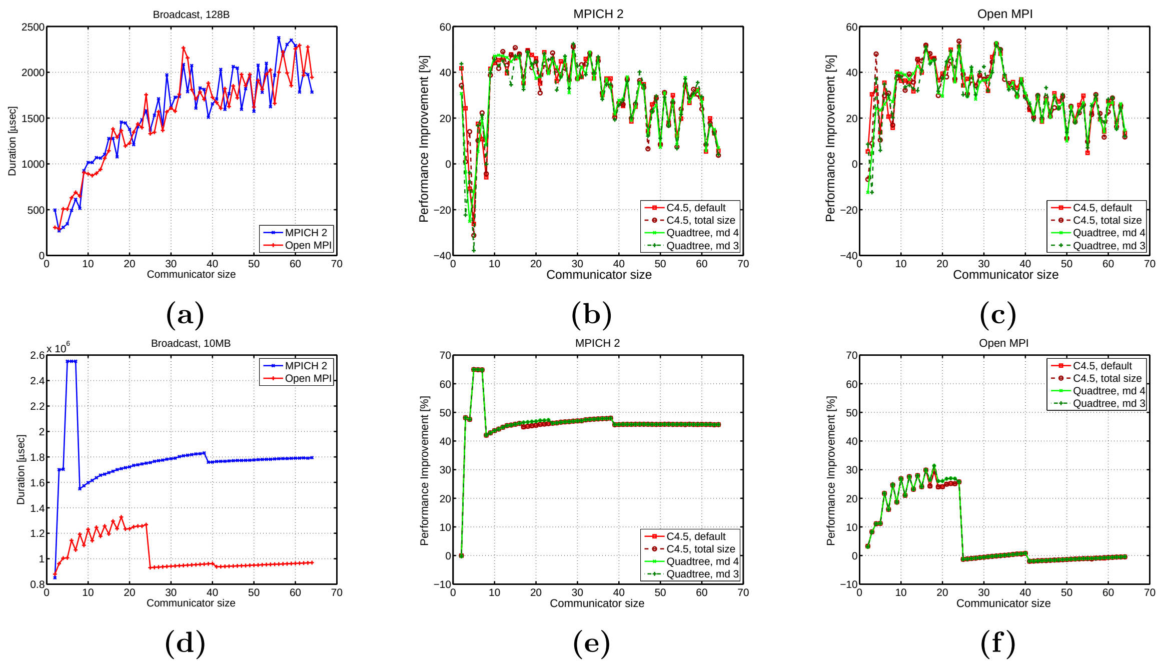


Figure 6.23: Performance improvement over native (b), (e) MPICH 2 and (c), (f) Open MPI Broadcast using (a), (b), (c) 128 B and (d), (e), (f) 10 MB message sizes.

6.7 Case study: Parallel Ocean Program

The application performance is the ultimate yardstick of a tuning process. The work presented in this dissertation helped improve the performance of Parallel Ocean Program [Dukowicz et al., 1993] on the Jaguar cluster at NCCS [Graham et al., 2007]. The measurements in this section were collected by external collaborators, we performed the result analysis and suggested changes.

The Parallel Ocean Program (*POP*) is a parallel application which models ocean behavior as a part of the Community Climate System Model. The Community Climate System Model is used to provide input to the Intergovernmental Panel on Climate Change. POP is dominated by a large number of small 24 B (three doubles) allreduce operations.

Graham et al. in [Graham et al., 2007] analyzed the performance of POP on NCCS's Jaguar cluster using Open MPI version 1.3 and Cray MPI. Jaguar has total of $11,508$ dual socket 2.6 GHz dual-core AMD Opteron chips and is interconnected with a 3-D torus with SeaStar communication processor and network router, which offloads the network communication from the main processor.

Cray MPI is based on MPICH [MPICH, 1994] and includes optimizations for Cray-specific hardware for point-to-point communication. Open MPI supports two point-to-point management layers (PMLs) for communication over Cray's Portal interconnect: CM and OB1. The CM PML is designed to provide the direct access to the MPI-sematics aware network library. The OB1 PML is designed to provide support for multiple networks concurrently, and in general incurs higher latency than CM. In direct comparison of point-to-point communication, Cray MPI latency was $4.75\mu sec$, while Open MPI's CM and OB1 PMLs achieve $4.91\mu sec$ and $6.16\mu sec$, respectively. In terms of bandwidth, all three versions achieve similar asymptotic bandwidth: between $1980\frac{Bytes}{sec}$ and $1990\frac{Bytes}{sec}$, however Cray MPI and CM PML achieve better bandwidth for intermediate size messages than OB1 PML.

The default Open MPI decision function for allreduce and MPICH select the recursive doubling algorithm for message containing three doubles. However, mostly due to better latency, for this application, Cray MPI outperformed Open MPI on a number of communicator sizes. We measured performance of different reduce, broadcast, and allreduce algorithms using SKaMPI benchmark up to 91 process. We compared the measured results to the default Open MPI decision function for allreduce (Figure 6.24) and discovered that for large process counts (60 and above) the basic allreduce implementation (reduce + broadcast) outperformed the recursive doubling algorithm. Selecting the reduce + broadcast over recursive doubling for 24 B message on 91 processes achieves 25% and 11% performance improvement using CM and OB1, respectively. For 24 B message size, the default reduce function selects the binary tree algorithm, and broadcast utilizes the binomial tree.

Open MPI allows user to specify the method to use for the particular program run via command line parameters (Section 3.1.3). Selecting the reduce + broadcast implementation for allreduce in this case improved overall performance of the step time to run one degree resolution POP application using Open MPI and CM PML on 256 processes from 184.32 [*sec*] to 164.4 [*sec*], a 10.8% improvement. The same step took 165.59 [*sec*] using Cray MPI. By switching to a more appropriate reduce method, we were able to both compensate for the difference in latency and slightly improve the application performance.

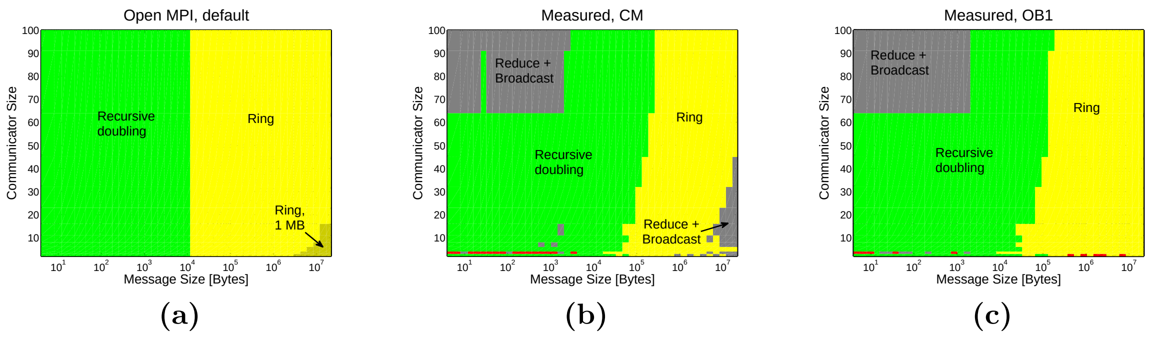


Figure 6.24: Allreduce decision maps on Cray XT4 (a) default Open MPI, and measured using (b) CM PML (c) OB1 PML.

Chapter 7

Summary and Conclusions

This dissertation focuses on investigating different approaches to platform-specific tuning of the MPI collective algorithm selection process. We employ modeling, graphical encoding, and machine learning techniques to interpret and store MPI collective algorithm performance information in order to automatically generate a run-time decision / algorithm-selection function.

Modeling techniques provide an elegant and fast way to determine the best available method for the problem instance at hand. The models are also invaluable for quick understanding of expected algorithm performance. However, the accuracy of the predictions varies significantly from algorithm to algorithm and for different ranges of collective input parameters. Inclusion of internal message segmentation into an equation exposes additional model limitations as neither of the considered point-to-point models is designed to handle communication / computation overlap well. Hockney- and LogGP-based performance models are amenable to numerical analysis and interpretation since their parameters are constants. This allows us to determine the optimal segment size for a segmented algorithm easily. At the same time, this is the main reason why neither of the two models is able to capture some of the experimentally observed, non-linear, behaviors. PLogP-based performance models are harder to analyze because parameters such as gap and overhead are a function of message size. It is often impossible to analytically determine properties of PLogP-based functions (such as optimal segment size). In general, both LogGP- and PLogP-based collective performance models had better accuracy than the corresponding Hockney models. However, the MPI community uses the Hockney model almost exclusively for analyzing algorithm performance. Parallel communication models are the only approach that can help for optimizations of large scale systems as the sparse performance data limits the applicability of the graphical encoding and statistical learning methods. Looking at applicability of models to run-time decision functions, the model based decision function selects a method that achieves the minimum (predicted) duration for the particular set of input parameters. The run-time efficiency of this function depends greatly on model implementations. In our tests, the run-time cost of the PLogP model was significantly higher than the Hockney and LogGP case - mainly due to the time it takes to determine gap and overhead parameters in cases when the measured value is missing.

If we only consider communicator and message size input parameters, the information about the best available collective implementation can be visualized using decision maps.

Quadtree encoding is a well-understood image encoding algorithm and provides a direct way to store algorithm performance information in terms of communicator and message size. The amount of stored information is manipulated by specifying either the maximum tree depth or region accuracy threshold at the time when the tree is built. Between the constraints, limiting the maximum tree depth achieves a lower performance penalty for the tree with the same average depth. Starting from a quadtree, an algorithm selection function can be created by converting internal nodes of the quadtree to appropriate conditional statements. In addition, the maximum quadtree depth sets an upper bound on the maximum number of statements, which need to be evaluated to reach a decision. Finally, under the assumption that input parameter values are evenly distributed, the average tree depth corresponds to the average number of statements to be evaluated. The major limitations of quadtree approach are the restriction to two-dimensional input parameter space and that efficiency of encoding depends on the shape of initial data.

In order to expand input parameters space from two dimensions, we consider the C4.5 decision tree algorithm. C4.5 is a supervised learning algorithm that works with a set of pre-defined categories/classes and can handle variable number of attributes (input parameters). Each of the internal nodes in a C4.5 decision tree corresponds to a single attribute test. Thus, used directly, C4.5 trees can only capture rectangular regions in multidimensional input parameter space. However, using constructive induction, we can create composite attributes that can handle non-trivial relationships between attributes. For example, in case of the all-to-all type of algorithms, we consider "total message size", "even communicator size", and "power-of-two communicator size" composite attributes. As in the case of quadtree, the algorithm selection function generation process is equivalent to replacing internal nodes of the decision tree with equivalent statements. Unlike quadtrees, the size of C4.5 decision tree cannot be manipulated easily. Even trees with just a few leaf nodes can have a significant depth. In addition, the C4.5 algorithm is copyrighted, so it is not clear if it can be used for purposes other than academic exploration.

In direct comparison, C4.5 trees are able to construct the most accurate decision functions. The mean performance penalty of these trees was below 3% with the median value at 0%. The maximum-depth limited quadtrees of depth three were suffering less than 10% performance penalty in comparison to the complete quadtree, which had six levels in all cases we considered. The performance of analytical models was least consistent with the mean performance penalty varying between 4% and 85% in some cases. However, based on platform-specific tuning results on the Grig cluster, we can conclude that both quadtree and C4.5 can produce similar decision functions that perform almost the same and can achieve as much as 42.85% mean performance improvement over native broadcast implementations.

The techniques discussed in this dissertation were applied to determine the most appropriate collective implementation for a POP application on a Cray XT4 system. Selecting the platform-specific best method in Open MPI decreased the execution time of application step by approximately 10%, allowing Open MPI to beat the Cray MPI implementation overall by a slight margin. Selecting an appropriate, platform-specific method at run-time can lead to a significant performance improvement. Methods we proposed and examined are a step towards the ultimate goal of achieving a fully tunable, platform-specific, collective implementation.

In order to support this work, we implemented the Optimized Collective Communication library, OCC. The OCC consists of four modules: methods, performance measurement, verification testing, datatypes, and decision module. The methods module contains a number of different collective algorithms discussed in Section 4.1. The performance measurement and functional verification tests are sets of micro benchmarks that measure the performance and verify correctness of implemented algorithms and can support user defined datatypes from the datatype module. Finally, the decision module provides the necessary functionality for analysis of experimental data, decision map and function generation, and computation of performance penalty and run time cost. OCC is described in detail in Appendix A.1.1.

In addition, all of the algorithms were implemented as a part tuned collective module in an Open MPI distribution. Tuned Open MPI module provides a good framework for experimenting with different algorithm selection choices. We spent extensive time to tune the Open MPI collectives for a general "high-performance system" case. Thus, the default Open MPI collective performance should achieve good performance. But as we have seen in the case of the POP application on the Cray XT4, when the best possible performance is necessary, platform-specific optimizations are needed. Current design of Open MPI collectives allows user to select an appropriate algorithm for a single collective. This is sufficient if the application relies on a single input parameter set for a collective. If an application uses different sets of input parameters for a collective (for example both small and large message sizes), selecting a single method can lead to performance degradation. The dynamic rules which are currently implemented in Open MPI are a very flexible alternative, but recent measurements showed that the overhead of using dynamic rules can be significant, annulling the benefits of selecting the right algorithm for small message sizes.

The time-to-decision is critical for high performance collective operations. Based on our results, we conclude that a pre-compiled decision function is the way to achieve best time-to-decision and offers the most flexibility in terms of what type of rules can be included. All the methods we explored can be used to generate source code for a decision function. The newly proposed design for Open MPI collectives moves the tuned algorithm implementations to a common place, and the different decision functions become collective frameworks themselves. Thus, an advanced user will be able generate their own platform-specific decision functions and use them automatically as a part of Open MPI. The work done in this dissertation can help them build accurate and efficient platform-specific decision functions automatically.

Bibliography

Bibliography

[Alexandrov et al., 1995] Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. (1995). LogGP: Incorporating long messages into the LogP model. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM Press.

[Almasi et al., 2005] Almasi, G., Archer, C., Castanos, J., Gunnels, J., Erway, C., Heidelberger, P., Martorell, X., Moreira, J., Pinnow, K., Ratterman, J., et al. (2005). Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2):393–406.

[Barchet-Estefanel and Mounié, 2004] Barchet-Estefanel, L. A. and Mounié, G. (2004). Fast tuning of intra-cluster collective communications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*, pages 28–35. Springer. Proceedings, 11th European PVM/MPI Users' Group Meeting.

[Barchet-Steffenel and Mounié, 2005] Barchet-Steffenel, L. and Mounié, G. (2005). Total exchange performance modelling under network contention. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, volume 3911 of *LNCS*, pages 100–107.

[Bell et al., 2003] Bell, C., Bonachea, D., Cote, Y., Duell, J., Hargrove, P., Husbands, P., Iancu, C., Welcome, M., and Yelick, K. (2003). An evaluation of current high-performance networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 28.1. IEEE Computer Society.

[Bernaschi et al., 2003] Bernaschi, M., Iannello, G., and Lauria, M. (2003). Efficient implementation of reduce-scatter in MPI. *J. Syst. Archit.*, 49(3):89–108.

[Bhowmick et al., 2007] Bhowmick, S., Eijkhout, V., Freund, Y., and Fuentes, E. Keyes, D. (2007). Application of Machine Learning in Selecting Sparse Linear Solvers. *For Publication on the International Journal of High Performance Computing Applications.*

[Bouteiller et al., 2006] Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., and Cappello, F. (2006). MPICH-V: a multiprotocol fault tolerant MPI. *International Journal of High Performance Computing and Applications*, 20(3):319–333.

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

[Chan et al., 2006a] Chan, E., Heimlich, M., Purkayastha, A., and van de Geijn, R. (2006a). Collective communication: Theory, practice, and experience. Technical Report FLAME Working Note No. 22, Department of Computer Sciences, The University of Texas at Austin.

[Chan et al., 2006b] Chan, E., van de Geijn, R., Gropp, W., and Thakur, R. (2006b). Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the seventeenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press.

[Chan et al., 2004] Chan, E. W., Heimlich, M. F., Purkayastha, A., and van de Geijn, R. A. (2004). On optimizing collective communication. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 145–155, Washington, DC, USA. IEEE Computer Society.

[Chen et al., 2005] Chen, J., Zhang, L., Zhang, Y., and Yuan, W. (2005). Performance evaluation of allgather algorithms on terascale linux cluster with fast ethernet. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 437, Washington, DC, USA. IEEE Computer Society.

[Culler et al., 1993] 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*, pages 1–12. ACM Press.

[Dukowicz et al., 1993] Dukowicz, J. K., Smith, R. D., and Malone, R. C. (1993). A reformulation and implementation of the Bryan-Cox-Semter Ocean Model on the connection machine. *Journal of Atmospheric and Oceanic Technologies*, 10:195–208.

[Earth Simulator, 2002] Earth Simulator (2002). The Earth Simulator - supercomputing site. `http://www.top500.org/system/5628`. Last Accessed on August 2007.

[Eijkhout et al., 2005] Eijkhout, V., Fuentes, E., Edison, T., and Dongarra, J. J. (2005). The component structure of a self-adapting numerical software system. *International Journal of Parallel Programming*, 33(2).

[Fagg et al., 2006] Fagg, G., Bosilca, G., Pješivac-Grbović, J., Angskun, T., and Dongarra, J. (2006). Tuned: A flexible high performance collective communication component developed for open mpi. In *Proceedings of 6th Austrian-Hungarian workshop on distributed and parallel systems (DAPSYS)*, Innsbruck, Austria. Springer-Verlag.

[Fagg et al., 2004] Fagg, G. E., Gabriel, E., Bosilca, G., Angskun, T., Chen, Z., Pjesivac-Grbovic, J., London, K., and Dongarra, J. (2004). Extending the mpi specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference (ICS) 2004*. Primeur.

[Fagg et al., 2003] Fagg, G. E., Gabriel, E., Chen, Z., Angskun, T., Bosilca, G., Bukovsky, A., and Dongarra, J. J. (2003). Fault tolerant communication library and applications for high performance computing. In *LACSI Symposium*.

[Faraj et al., 2006] Faraj, A., Yuan, X., and Lowenthal, D. (2006). Star-mpi: Self tuned adaptive routines for mpi collective operations. In *The 20th ACM International Conference on Supercomputing (ICS06)*, Queensland, AUstralia.

[Finkel and Bentley, 1974] Finkel, R. and Bentley, J. (1974). Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9.

[FT-MPI, 2003] FT-MPI (2003). Fault tolerant MPI implementation. `http://icl.cs.utk.edu/ftmpi/`. Last accessed on May 2007.

[Gabriel et al., 2004] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[Graham et al., 2007] Graham, R. L., Bosilca, G., and Pješivac-Grbović, J. (2007). A comparison of application performance using Open MPI and Cray MPI. In *Cray User's Group meeting, CUG 2007*.

[Grama et al., 2003] Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Pearson Education Limited, second edition edition.

[Gropp and Lusk, 1999] Gropp, W. and Lusk, E. (1999). Reproducible measurements of MPI performance characteristics. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18. Springer-Verlag.

[Harness, 1999] Harness (1999). Harness, Parallel Virtual Machine Project. `http://www.csm.ornl.gov/harness/`. Last accessed on June 2007.

[Hartmann et al., 2006] Hartmann, O., Kühnemann, M., Rauber, T., and Rünger, G. (2006). Adaptive selection of communication methods to optimize collective MPI operations. In *Proceedings of International Conference ParCo 2005)*, Malaga, Spain.

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

[Kielmann et al., 2001] Kielmann, T., Bal, H. E., Gorlatch, S., Verstoep, K., and Hofman, R. F. (2001). Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing*, 27(11):1431–1456.

[Kielmann et al., 2000] Kielmann, T., Bal, H. E., and Verstoep, K. (2000). Fast measurement of LogP parameters for message passing platforms. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1176–1183, London, UK. Springer-Verlag.

[Kielmann et al., 1999] 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. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140. ACM Press.

[Knuth, 1998] Knuth, D. E. (1998). *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 2 edition.

[LA-MPI, 2002] LA-MPI (2002). The Los Alamos Message Passing Interface. `http://public.lanl.gov/lampi/`. Last accessed on June 2007.

[Lagoudakis and Littman, 2000] Lagoudakis, M. G. and Littman, M. L. (2000). Algorithm selection using reinforcement learning. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[LAM/MPI, 2002] LAM/MPI (2002). Local Area Multicomputer/Message Passing Interface. `http://www.lam-mpi.org/`. Last accessed on June 2007.

[Martinasso and Méhaut, 2006] Martinasso, M. and Méhaut, J.-F. (2006). Model of concurrent mpi communications over smp clusters. Research Report 5910, INRIA.

[Moody et al., 2003] Moody, A., Fernandez, J., Petrini, F., and Panda, D. K. (2003). Scalable nic-based reduction on large-scale clusters. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 59, Washington, DC, USA. IEEE Computer Society.

[MPI Forum, 1995] MPI Forum (1995). MPI: A message-passing interface standard. `http://www.mpi-forum.org/docs/docs.html`.

[MPICH, 1994] MPICH (1994). Implementation of MPI standard. `http://www-unix.mcs.anl.gov/mpi/mpich1/`. Last accessed on May 2007.

[MPICH-GM, 2007] MPICH-GM (2007). Myricom MPICH-GM software. `http://www.myri.com/scs/download-mpichgm.html`. Last accessed on August 2007.

[MPICH-MX, 2007] MPICH-MX (2007). Myricom MPICH-MX software. `http://www.myri.com/scs/download-mpichmx.html`. Last accessed on August 2007.

[MPICH-V, 2007] MPICH-V (2007). MPI implementation for volatile resources. `http://mpich-v.lri.fr/`. Last accessed on May 2007.

[MPICH2, 2002] MPICH2 (2002). Implementation of MPI 2 standard. `http://www-unix.mcs.anl.gov/mpi/mpich/`. Last accessed on May 2007.

[MVAPICH, 2007] MVAPICH (2007). MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/. Last accessed on August 2007.

[NetPIPE, 2005] NetPIPE (2005). A Network Protocol Independent Performance Evaluator. `http://www.scl.ameslab.gov/netpipe/`.

[OCC, 2005] OCC (2005). Optimized Collective Communication Library. `http://www.cs.utk.edu/~pjesa/projects/occ/`. Accessed on March 2006.

[Open MPI, 2005] Open MPI (2005). Open Source High Performance Computing. `http://www.open-mpi.org/`. Last accessed on May 2007.

[PACX-MPI, 2003] PACX-MPI (2003). PArallel Computer eXtension Message Passing Interface. `http://www.hlrs.de/organization/amt/projects/pacx-mpi`. Last accessed on June 2007.

[PAPI, 2005] PAPI (2005). Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[Petrini et al., 2001] Petrini, F., Coll, S., Frachtenberg, E., and Hoisie, A. (2001). Hardware- and software-based collective communication on the quadrics network. *nca*, 00:0024.

[Petrini et al., 2003] Petrini, F., Fernandez, J., Frachtenberg, E., and Coll, S. (2003). Scalable collective communication on the asci q machine. In *Proceedings of 11th Symposium on High Performance Interconnects*, pages 54 – 59. IEEE Computer Society.

[Pješivac-Grbović, 2007] Pješivac-Grbović, J. (2007). Open MPI collective operation performance on thunderbird. Technical Report UT-CS-07-594, The University of Tennessee, Computer Science Department, Knoxville, TN. `http://www.cs.utk.edu/~library/2007.html`.

[Pješivac-Grbović et al., 2007a] 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.

[Pješivac-Grbović et al., 2007b] 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*, number 4641 in LNCS, pages 105–115. Springer-Verlag, Berlin Heidelberg.

[Pješivac-Grbović et al., 2007c] Pješivac-Grbović, J., Bosilca, G., Fagg, G. E., Angskun, T., and Dongarra, J. J. (2007c). MPI collective algorithm selection and quadtree encoding. 33/9:613–623.

[Püschel et al., 2005] Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B. W., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. (2005). SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, San Mateo, California.

[Quinlan, 2006] Quinlan, J. R. (2006). C4.5 source code. `http://www.rulequest.com/Personal/`.

[Rabenseifner, 1997] Rabenseifner, R. (1997). A new optimized mpi reduce algorithm. Technical report, High-Performance Computing Center, University of Stuttgart.

[Rabenseifner, 1999] Rabenseifner, R. (1999). Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the Message Passing Interface Developer's and User's Conference (MPIDC '99)*, pages 77–85.

[Rabenseifner, 2004] Rabenseifner, R. (2004). Optimization of collective reduction operations. In *ICCS 2004*, number 3036 in LNCS, pages 1–9. Springer Verlag. 4th International Conference on Computational Sciences, Krakow, Poland.

[Rabenseifner and Träff, 2004] Rabenseifner, R. and Träff, J. L. (2004). More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *Proceedings of EuroPVM/MPI*, Lecture Notes in Computer Science. Springer-Verlag.

[Sanders and Träff, 2006] Sanders, P. and Träff, J. L. (2006). Parallel prefix (scan) algorithms for MPI. In Mohr, B., Träff, J. L., Worringen, J., and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 4192 in LNCS, pages 49–57. Springer Belin / Heidelberg. 13th European PVM/MPI User's Group Meeting, Bonn, Germany.

[SKaMPI, 2005] SKaMPI (2005). Special Karlsruher MPI - benchmark. `http://liinwww.ira.uka.de/~skampi/`.

[Snir et al., 1998] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1998). *MPI: The Complete Reference*, volume 1. The MIT Press, 2nd edition.

[Stewart, 1995] Stewart, J. (1995). *Calculus*. Brooks/Cole Publishing Company, third edition edition.

[Thakur and Gropp, 2003] Thakur, R. and Gropp, W. (2003). Improving the performance of collective operations in MPICH. In Dongarra, J., Laforenza, D., and Orlando, S., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 2840 in LNCS, pages 257–267. Springer Verlag. 10th European PVM/MPI User's Group Meeting, Venice, Italy.

[Thakur et al., 2005] Thakur, R., Rabenseifner, R., and Gropp, W. (2005). Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66.

[Thunderbird, 2006] Thunderbird (2006). Thunderbird - supercomputing site. `http://www.top500.org/system/8114`. Last Accessed on August 2007.

[Top 500, 2007] Top 500 (2007). TOP500 supercomputer sites. `http://www.top500.org`. Published biannually since 1993.

[Torgo, 1999] Torgo, L. (1999). *Inductive learning of tree-based regression models.* PhD thesis, Department of Computer Science, Faculty of Sciences, University of Porto.

[Vadhiyar et al., 2000] Vadhiyar, S. S., Fagg, G. E., and Dongarra, J. J. (2000). Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 3. IEEE Computer Society.

[Vadhiyar et al., 2004] Vadhiyar, S. S., Fagg, G. E., and Dongarra, J. J. (2004). Towards an Accurate Model for Collective Communications. *International Journal of High Performance Computing Applications*, 18(1):159–167.

[Vapnik, 1998] Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley, New York, NY.

[Vuduc et al., 2004] Vuduc, R., Demmel, J. W., and Bilmes, J. A. (2004). Statistical Models for Empirical Search-Based Performance Tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94.

[Whaley et al., 2001] Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.

Appendix

Appendix A

Appendix

A.1 Implementation

This chapter provides a short overview of the software developed for the purpose of this dissertation and related projects.

A.1.1 Optimized collective communication

Optimized collective communication library (*OCC*) is a framework for functional collective method verification and performance testing. OCC is an MPI collective library built on top of the MPI point-to-point operations. OCC consists of four modules: methods, verification and performance-testing, datatypes, and decision modules. The methods module provides a simple interface for addition of new collective algorithms. Currently it provides multiple algorithms for barrier, broadcast, reduce, allgather, alltoall, allreduce, scatter, and wrappers around native implementations for all remaining collectives. The verification module provides the basic verification tools for the existing methods. The performance module provides a set of micro-benchmarks for the collective operations defined in the library. Both verification and performance testing capabilities work automatically with user defined datatypes from datatypes module. The datatypes module provides standardized interface for the user to add more complex datatypes, which can then be used in verification and performance testing. The decision module provides functionality for managing performance data, automatic experimental decision map generation, and quadtree and run-length encoding related functionality necessary for analysis of decision maps.

A.1.2 Tuned collective component in Open MPI

Our group at Innovative Computing Laboratory at the University of Tennessee is a major developer of the tuned collective component in Open MPI. Many of the developments related to this project are part of this dissertation.

The tuned collective component [Fagg et al., 2006] is implemented on top of the point-to-point communication routines. Since Open MPI can automatically select the most appropriate communication device for two processes, this implementation can automatically utilize the most appropriate way to send a message between two processes, e.g. shared memory when processes reside on the same node, and appropriate interconnect for communication between distinct nodes.

The tuned component is currently under active development. At the time of this writing the following collective operations and corresponding algorithms are available:

- barrier: fan-in-fan-out, double ring, recursive doubling, bruck.
- broadcast: linear, k-chain, binary, binomial, split-binary, and pipeline. All algorithms except the linear one support explicit internal message segmentation.
- gather: linear with and without synchronization, binomial.
- allgather: linear, bruck, recursive doubling, neighbor exchange.
- alltoall: linear with and without synchronization, bruck, pairwise.
- scatter: linear and binomail.
- reduce: generalized reduce with pipeline, k-chain, binary, and binomial topology for commutative operations, and linear and in-order binary tree generalized reduce for non-commutative operations.
- allreduce: linear, recursive doubling, ring with and without segmentation.
- reduce_scatter: reduce + scatterv, recursive halving, ring.

Section 4.1 provides descirption of the algorithms listed above.

The algorithm selection in this component can be done in either of the following three ways: via a compiled decision function; via user-specified command line flags; or using a rule-based run-length encoding scheme that can be tuned for a particular system. Currently, the decision functions in Open MPI are predetermined by the implementers based on the results collected using Myricom's MX interconnect on the Grig and Frodo clusters at the University of Tennessee, Knoxville. The user-specified command line flags allow user to set the collective method to be used for the particular program run. It is useful for applications that are dominated by a single type of collective, e.g. large message size allreduce, when the default decision function selects suboptimal algorithm.

Vita

Jelena Pješivac-Grbović was born in Belgrade, Serbia on October 31st 1978. She attended Jovan Sterija Popović elementary, followed by the Mathematical High School. After high school, she attended the School of Electrical Engineering at the University of Belgrade. She transferred to Ramapo College of New Jersey in August 1999 and started the program in Computer Science and Physics. In May 2003, she graduated from Ramapo College as the second in her class, receiving the Outstanding student awards in both majors. As an undergraduate, she actively participated in research projects. She completed three summer internships at Los Alamos National Laboratory working on avascular tumor models under supervision of Dr. Yi Jiang. In August 2003, she started Ph.D. program in Computer Science at the University of Tennessee, Knoxville. She was a recipient of merit-based Hilton A. Smith Graduate Fellowship for the first-year graduate students. At UT, she worked as a Graduate Research Assistant in Distributed and Parallel Computing group at the Innovative Computing Laboratory at the Computer Science department. Her dissertation advisor was Dr. Jack Dongarra and her project leaders were Dr. Graham Fagg and Dr. George Bosilca. Her main area of research were parallel communication libraries, such as MPI, with focus on collective operations. While at ICL, she was an active developer in FT-MPI/Harness and Open MPI projects. Together with her colleagues from the Distributed and Parallel Computing group, she was a recipient of the AMD Award for Requirements for HPC Systems Software at International Supercomputer Conference in 2004. In 2006, she received the Citation for Extraordinary Professional Promise from the University of Tennessee, Knoxville. She is co-author of five journal papers and more than 10 conference proceeding and workshop papers. She defended her dissertation "Towards Automatic and Adaptive Optimizations of MPI Collective Operations" in September 2007. Her committee consisted of Dr. Jack Dongarra, Dr. James Plank, Dr. Itamar Elhanany, and Dr. George Bosilca. After graduation, she joined Google Inc.