University College Dublin

Efficient and accurate selection of optimal MPI collective algorithms using analytical performance modelling

Emin Nuriyev

This thesis is submitted to University College Dublin

in fulfilment of the requirements for the degree of

Doctor of Philosophy in Computer Science

March 2021

School of Computer Science

Head of School: Chris Bleakley

Research Supervisor: Alexey Lastovetsky

Acknowledgements

First, I would like to thank my deep and sincere gratitude to my supervisor Dr Alexey Lastovetsky, founding Director of the Heterogeneous Computing Laboratory, for giving me the opportunity to do my PHD in the Heterogeneous Computing Laboratory. I remember the first meeting we had back in June 2016, we discussed my background and MPI. Then he assigned me the task to implement a binomial tree broadcast algorithm and compare the performance of the algorithm to Open MPI MPI_Bcast implementation.

Second, I would like to thank my dear friend Khalid Hasanov who encouraged me to start my research journey and morally supported me through the journey. Special thanks for the support during the first semester when I was learning about MPI specification and HPC platforms.

Third, I would like to thank all my colleagues from the Heterogeneous Computing Laboratory, Arsalan Shahid, Muhammad Fahad, Tania Malik, Ravi Reddy Manumachu, Semen Khokhriakov, Hamidreza Khaleghzadeh. Special thanks to Egemen for his invaluable coffee chats. It was a pleasure working and studying with you all.

This thesis is the result of research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

The experiments were carried out on Grid'5000 developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies. Another part of the experiments was carried out using the resources of the Barcelona Supercomputing Center.

I am forever indebted to my parents, Yaqub and Mehriban for everything I am and everything I will ever be. Special thanks to my brothers, Mahir and Mehman, for taking care of our parents while I left the country and moved to Ireland to start my PhD education.

Finally, I would like to thank my wife Narmina for her understanding and continued support. Last but by no means least, I am thankful to my little angels Khadija and Ayla for bringing so much happiness into our life.

DEDICATION

To

My family

Abstract

The message passing interface (MPI) is the de facto standard in distributed-memory parallel programming. MPI collective operations provide optimized solutions to realize communication among groups of processes. The performance of collective operations has been a critical issue since the advent of MPI. Many algorithms have been proposed for each MPI collective operation but none of them proved optimal in all situations. Different algorithms demonstrate superior performance depending on the platform, the message size, the number of processes, etc. MPI implementations perform the selection of the collective algorithm empirically, executing a simple runtime decision function. Compiled decision function selects the same algorithm on all platforms for given parameters such as message size, number of processes and so forth. Therefore, while efficient, this approach does not guarantee the optimal selection. As a more accurate but equally efficient alternative, the use of analytical performance models of collective algorithms for the selection process was proposed and studied. Unfortunately, the previous attempts in this direction have not been successful.

In this thesis, we revisit the analytical model-based approach and propose two innovations that significantly improve the selective accuracy of analytical models: (a) We derive analytical models from the code implementing the

algorithms rather than from their high-level mathematical definitions. This results in more detailed models. (b) We estimate model parameters separately for each collective algorithm and include the execution of this algorithm in the corresponding communication experiment. Our approach takes the following steps to select the optimal collective algorithm: (1) We build analytical performance models for each collective algorithm taking into account implementation and platform details. (2) We execute each collective algorithm on the platform of interest and collect performance information. (3) We build a system of linear equations with unknown model parameters for each algorithm using collected performance information and built models. (4) We find the algorithm-specific values of the model parameters from the system of linear equations using linear regression. (5) We select the optimal collective algorithm at runtime by comparing the performance of algorithms using the built models and estimated model parameters.

The thesis presents a theoretical and experimental study of model-based selection. We present the proposed approach using broadcast and gather algorithms implemented in Open MPI 3.1. The models of the algorithms are built using the Hockney and τ-Lop communication performance models. We implement and experimentally validate our approach on three different platforms of multi-core processors, Grid5000, MareNostrum4, and Shaheen II. We have implemented a software tool supporting all five steps of the proposed method of selection of the optimal collective algorithms at runtime - collection of the performance information of the algorithms, estimation of the model parameters, and selection of the optimal algorithms at runtime.

Contents

List of Figures

List of Tables

Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree stated on the Title Page, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.

Chapter 1

Introduction

High-performance computing (HPC) is focused on aggregating computing power to deliver much higher performance than a general-purpose computer. HPC offers standards, techniques and tools to process data on large-scale systems and applications. Due to the large amount of data generated by the industry in recent years, HPC becomes essential for many different areas such as media, entertainment, finance, healthcare and so forth. With increasing the complexity of problems to be solved, the scale of the supercomputers and clusters are becoming larger. Increasing computing power on multi-core clusters, high bandwidth memory/IO, forces the HPC community to develop new algorithms and techniques to improve the performance of parallel applications and runtime systems.

The TOP500 project [1] was started in 1993 to provide ranks and details of the 500 most powerful supercomputers in the world. Based on the latest TOP500 list for 2020, the Japanese Fugaku [2] is the world's most powerful supercomputer. We see the remarkable growth in power and scale of supercomputers in the last 10 years. While as of November 2010 the entry-level into the TOP50 is at 126.5 Tflop/s, the entry-level into the TOP50 moves up to 1.32 petaflops in June 2020 rankings. Besides that, we can easily see how the scale of supercomputers changes over the years. Compared to the November 2010 statistics [3] where the number one supercomputer Tianhe-1 [4] was powered by 71,680 CPU cores, the number

one supercomputer FUGAKU in November 2020 was powered by 7,630,848 CPU cores. Mostly, the supercomputers in the TOP 500 list provides MPI library implemented by a supplier which is based on some mainstream open-source MPI implementation such as MPICH [5] or Open MPI [6, 7].

Developing high-performance applications for such types of parallel platforms as supercomputers and clusters is a challenging task. Main challenges come from the use of different network technologies and system memories in parallel systems each of which requires different approaches/technologies to have high-level parallelism in the application. I/O management on large scale systems for data-intensive scientific applications requires a special HPC I/O model to design optimal distributed-memory applications.

The message passing interface (MPI) [8] is the de-facto standard, which provides a reliable and portable environment for developing high-performance parallel applications on different platforms. MPI offers portable and scalable performance on HPC platforms. Therefore, it has been dominantly used since its invention in HPC applications. MPI proposes an execution model based on processes deployed on the hardware resources of the HPC platform and communicating using message passing primitives. MPI standard offers point-to-point and collective communication. Both types of communication are defined with different semantics, including non-blocking, buffered and persistent communication.

Collective routines in MPI involve a group of processes communicating in an isolated context, and those collectives rely on the semantics of *collective operations* such as *broadcast*, *gather*, *reduce* and so forth. A profiling study [9] reports that in average 80% of the total execution time of MPI applications is consumed by MPI collective operations. That is why significant research efforts have been invested in the design and implementation of efficient collective algorithms aimed to improve the performance of collective operations [10, 11, 12, 13, 14, 15, 16, 17, 18]. For example, MPICH [5, 19] employs three broadcast algorithms to implement *MPI_Bcast*. Open MPI 3.1 [6] employs six different algorithms to implement *MPI_Bcast* and five algorithms to implement *MPI_Allreduce*. On a given platform, different

algorithms will be optimal depending on many factors including the physical topology of the network, number of processes, message sizes and so forth. Unfortunately, there is no single collective algorithm optimal in all situations. Thus, there is a problem of selection of the optimal algorithm for each call of a collective routine, which normally depends on the platform, the number of processes, the message size and so forth.

The problem of selection of the optimal collective algorithm worsens for modern clusters, which are built up with multi-core CPUs with high-bandwidth memory I/O subsystems, connected by high performance networks. In those widespread architectures, processes deployed in the platform communicate using those uneven communication channels depending on their location. As a consequence, that scenario complicates the search for the more efficient collective algorithm given a mapping of processes to resources.

MPI Tools Information Interface (MPI_T) [8] is provided by the MPI standard and allows the MPI programmer to control internal variables of MPI implementations. For example, the MPI programmer can select the collective algorithm explicitly from the list of available algorithms for each collective call at runtime. However, it does not solve the problem of optimal selection delegating its solution to the programmer. Thus, finding the optimal algorithm is left to the MPI programmer.

MPI libraries provide modules and components allowing the MPI programmers to take advantage of multi-core CPUs and communication channels characteristics. For example, Open MPI offers a library communication component [20, 21], which employs a Linux utility KNEM [22] to enable direct inter-process memory copy. Remote direct memory access [23, 24, 25] through a network (e.g. Infiniband) is supported on some network software components. Such components reduce the time to transmit a message between two processes and this way contribute in acceleration of all collective algorithms. However, they do not solve the problem of selection between different collective algorithms implementing the same collective operation.

Several approaches to the selection of optimal MPI collective algorithms have been proposed and validated. The existing approaches to the selection

of optimal MPI collective algorithms can be classified into three main categories: (1) **Empirically Derived Decision Function;** (2) **Analytical Performance Modelling;** (3) **Machine Learning Algorithms**. We overview the three categories of approaches in the below sections.

1.1 Algorithm Selection of MPI Collectives using Empirically Derived Decision Functions

The use of empirically derived decision functions is the only approach for algorithm selection in mainstream MPI implementations. The main idea of the approach is based on extracting conditions for the selection from the performance data of the collective algorithms on a particular platform depending on the message size and the number of processes. For example, Listing 1.1 shows the Open MPI decision function for MPI_Bcast. The Open MPI decision function selects the broadcast algorithm based on the message size and number of processes. The shortcoming of this approach is that the conditions are extracted on a particular platform. It means the function built using this approach will be platform-specific.

Open MPI decision functions [26] are built on a cluster of AMD64 processors communicating across a Gigabit Ethernet Interconnect. An exhaustive benchmarking technique was used on the given platform to extract the rules to build the function in order to select the algorithm. The same approach is used for the MPICH decision functions. Decision functions in the MPICH implementation [27] are built using experimental performance data of the collective algorithms obtained on the Cray T3E 900 platform. Like Open MPI, the MPICH decision functions select the algorithm depending on the message size and number of processes. As we can see, both implementations use the performance of collective algorithms on a particular platform.

```
int bcast_intra_dec_fixed( void *buff,int count,
MPI_Datatype *datatype, int root,MPI_comm *comm) {
   const size_t small_message_size = 2048;
   const size_t intermediate_message_size = 370728;
   const double a_p16 = 3.2118e-6;
   const double b_p16 = 8.7936;
   const double a_p64 = 2.3679e-6;
   const double b_p64 = 1.1787;
   const double a_p128 = 1.6134e-6;
   const double b_p128 = 2.1102;

   int communicator_size;
   size_t message_size, dsize;

   communicator_size = MPI_comm_size(comm);
   MPI_Type_size(datatype, &dsize);
   message_size = dsize * (unsigned long)count;

   if ((message_size < small_message_size) || (count <= 1)){
      return binomial_tree_bcast(. . .);
   } else if (message_size < intermediate_message_size){
      return split_binary_tree_bcast(. . .);
   } else if (communicator_size < (a_p128 * message_size + b_p128)){
      return chain_bcast(. . . );
   } else if (communicator_size < 13) {
      return split_binary_tree_bcast(. . .);
   } else if (communicator_size < (a_p64 * message_size + b_p64)){
      return chain_bcast(. . .);
   } else if (communicator_size < (a_p16 * message_size + b_p16)){
      return chain_bcast(. . .);
   }
   return chain_bcast(. . .);
}
```

Listing 1.1: Open MPI decision function for MPI_Bcast.

The main advantage of this solution is its efficiency. The algorithm selection is very fast and does not affect the performance of the application. However, the MPI programmer has no control over the conditions built in the decision function. The main disadvantage of the existing decision functions is that they do not guarantee the optimal selection in all situations. On a particular platform, one subset of the implemented algorithms may always outperform the rest of the algorithms. Therefore, the decision functions constructed on a particular platform will only use that subset of the implemented algorithms. For example, due to better performance of the binary tree, split-binary tree and chain tree broadcast algorithms on the platform used for construction of its decision functions, the Open MPI 3.1 broadcast decision function does not use three out of six implemented broadcast algorithms, namely,the k-chain tree, binomial tree and linear tree broadcast algorithms.

1.2 Algorithm Selection of MPI Collectives using Analytical Performance Models

Analytical performance models offer us a low-cost efficient way to estimate the performance of collective algorithms. As an alternative approach, the use of analytical performance models of collective algorithms for the selection process has been proposed and studied. In the case of success, the analytical performance modelling approach, being as efficient as the existing decision functions approach, would guarantee the optimal selection in all situations.

Pjesivac-Grbovic et al. [28] first proposed algorithm selection of MPI collectives using analytical performance models. In this work, several communication performance models, such as Hockney [29], LogP [30], LogGP [31], PLogP [32], are used to build analytical performance models of collective algorithms. The performance models are built using the high-level definition of collective algorithms. The analytical performance models are then used in decision functions for selection of the optimal algorithm.

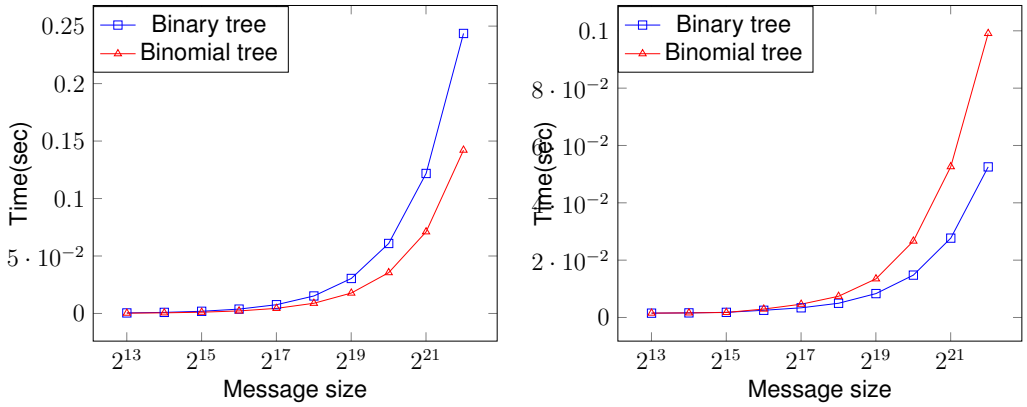


Figure 1.1: Performance estimation of the binary and binomial tree broadcast algorithms by the traditional analytical models in comparison with experimental curves. The experiments involve ninety processes (P=90). (a) Estimation by the existing analytical models. (b) Experimental performance curves.

Unfortunately, the analytical performance models proposed in this work could not reach the level of accuracy sufficient for selection of the optimal algorithm. Figure 1.1 shows the performance of the binary tree and binomial tree broadcast algorithms using: a) the estimation by the existing analytical models; b) the experimental results on the Grisou cluster of the Grid'5000 platform. It is evident that the existing models wrongly predict performance of collective algorithms.

1.3 Algorithm Selection of MPI Collectives using Machine Learning Techniques

Machine learning (ML) techniques as an alternative solution have also been tried to solve the problem of selection of optimal MPI algorithms. Supervised learning methods such as regression/classification trees (C4.5, SLIQ, SPRINT) [33], support vector machines [34], neural networks [35], are used to model the performance of collective algorithms.

Application of the quadtree encoding method [36] to the selection problem shows that the collection of detailed profiling data of collectives for all

message sizes and communication sizes is a very expensive procedure. Besides that, the constructed decision functions perform poorly on unseen data. Most recently Hunold et al. [37] studied the applicability of six different ML algorithms for selection of optimal MPI collective algorithms. The results show that the selection of the optimal algorithm without any information about the semantics of the algorithm yields inaccurate results. The main limitations of ML algorithms used as a solution to the selection problem are listed below.

- **Biased performance data** - A robust training set is crucial to the success of ML algorithms. The model built using biased performance data of the collective algorithms can overfit. Thus, it makes the model less usable for unseen data.
- **Training time** - Artificial Neural Network (ANN) takes very long training time to effectively train a model with traditional back propagation optimization techniques.
- **Runtime overhead** - Searching multidimensional data for optimal decision using regression tree predictors requires several iterations at application runtime for convergence. This technique at runtime can be a source of significant overhead.
- **Classification difficulty** - Large number of parameters ranging from algorithm index, segment size, mpi_affinity_alone, eager threshold, etc , can become increasingly hard problem to classify due to the explosion in connections to each of the output layers of an ANN.
- **Weak learner** - Decision trees tightly fit the given training data. Therefore, they are considered weak learners. Thus, the decision function built for a particular platform to select the optimal collective algorithm will perform poorly on unseen data.

1.4 Contributions

In this thesis, we revisit the model-based approach and propose a number of innovations that significantly improve the selective accuracy of analytical

models to the extent that allows them to be used for accurate selection of optimal collective algorithms. Our analytical modelling approach is based on the following innovations:

1. While previous attempts to build analytical performance models of collective algorithms only take into account their high-level mathematical definition, we derive our analytical models from the code implementing the algorithms. This results in much more detailed models, which are able to correctly compare the performance of different algorithms implementing the same collective operation.

2. We propose to estimate the model parameters separately for each collective algorithm and carefully design the communication experiments for their estimation.

   More specifically, we design a specific communication experiment for each collective algorithm, so that the algorithm itself would be involved in the execution of the experiment. Moreover, the execution time of this experiment must be dominated by the execution time of this collective algorithm. Then, we conduct a number of experiments on the target platform for a range of numbers of processors and message sizes and accurately measure their execution times. From these experiments, we derive a sufficiently large number of equations with the model parameters as unknowns. Finally, we use a solver to find the values of the model parameters.

We applied our approach to collective algorithms implemented in Open MPI. As a result, we managed to build a detailed analytical performance model for each collective algorithm and successfully use the models for selection of the optimal one. The accuracy of our solution has been validated on the Grid'5000 [38], MareNostrum4 [39] and Shaheen II [40] platforms.

The main contributions of this thesis can be summarized as follows:

- We propose and implement a new analytical performance modelling approach for MPI collective algorithms, which derives the models from the code implementing the algorithms.

- We propose and implement a novel approach to estimation of the parameters of analytical performance models of MPI collective algorithms, which estimates the parameters separately for each algorithm and includes the modelled collective algorithm in the communication experiment, which is used to estimate the model parameters.
- We experimentally validate the proposed approach to selection of optimal collective algorithms on two different clusters of the Grid'5000 platform.

1.5 Thesis Structure

The content of the thesis is organized as follows. Chapter 2 provides relevant information about the MPI standard and collective operations. Chapter 3 reviews the existing approaches to performance modelling, measurement of the model parameters, and algorithm selection problems. Chapter 4 introduces MPI collective algorithms implemented in Open MPI and describes our approach to construction of analytical performance models of MPI collective algorithms by deriving them from the MPI implementation. Chapter 5 presents our method to measure analytical model parameters. Chapter 6 presents experimental validation of the proposed approach. Chapter 7 concludes the thesis with a discussion of the results and an outline of the future work.

Chapter 2

Message Passing Interface

2.1 MPI Standard

The first version of MPI standard (MPI-1.0) was released in 1994 after collaboration of over forty organizations from academia and industry. The goal was to generate a standard which enables programmers to build portable, scalable and effective distributed memory applications. Due to the large scale collaboration, the first version of the standard delivered many essential features and specifications including point-to-point communication, collective operations, communication contexts (communicators), process topologies, bindings for FORTRAN 77 and C, and so forth. Therefore, starting from the first version the MPI became the de facto standard for distributed memory applications. Since 1994 several versions of MPI standard were delivered each of which was released with error corrections, new definitions, new routines and so forth. The latest version of the standard for today (MPI 3.1) was released in 2015.

MPI-2.0 [41] was approved in 1997. MPI-2.0 has been extended to the standard adding one-sided communications, extended collective communications, external interfaces and parallel I/O. Bindings for C++ have been added in MPI-2.0 as well.

MPI-3.0 was released on September 21, 2012. Collective operations introduced in MPI-1.0 are *blocking* collective operations. It means the

collective operations cannot take advantage of overlapping communication and computation. Thus, non-blocking collective operations were introduced in MPI-3.0 [42] for better performance. Non-blocking collectives return immediately to the running code and enable the user to utilise the CPU even during ongoing message transmissions.

MPI Tools Information Interface (MPI_T) was introduced in the 3.0 version of MPI in order to have control over internal environmental variables of MPI tools. The interface provides mechanisms for MPI implementers to expose internal variables. For example, we can set new broadcast algorithm for MPI_Bcast at runtime using *MPI_T_cvar_write()* function and *coll_tuned_bcast_algorithm* variable in Open MPI. MPI_T only defines standard specifications for API's, but the naming of variables is left to implementers.

We overview two types of MPI communications, point-to-point and collective, below in Section 2.2 and 2.3.

2.2 Point-to-Point Communication

A point-to-point communication is the basic mechanism in MPI designed to transmit a message between two processes. Two processes exchange a message using *send* and *receive* operations in context of point-to-point communication. In order to implement reliable communication between *sender* and *receiver*, four *communication modes* are defined in MPI: **standard**, **buffered**, **synchronous** and **ready**. The semantics of each communication mode has been provided in the MPI standard.

MPI provides *blocking* and *non-blocking* routines for send and receive operations. The definition of blocking send and receive operations are given below.

- Blocking send

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

The process calling MPI_Send sends *count* amount of message from *buf* to the receiver which is defined as *dest* in the routine. The routine blocks the process until the send operation is finished

- Blocking receive

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

The process calling MPI_Recv receives *count* amount of message from *sender* which is defined as *source* in the routine. The routine blocks the process until *buff* contains the received message.

The definition of non-blocking send and receive operations are given below.

- Non-blocking send

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
```

The process calling MPI_Isend initialises the send operation but does not complete it. The separate call can be called to check whether the send operation is completed or not.

- Non-blocking receive

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request)
```

The process calling MPI_Irecv initializes receive operation. The separate call can be called to check whether the receive operation is completed or not.

2.3 MPI Collective Operations

Unlike point-to-point communication where the communication is established between two processes, collective operations are designed as communication involves a group of processes to transmit a message. The key concept in MPI collectives is the *communicator* which defines a group of processes and provides the context for the operation. The definition of "MPI process" is up to the implementation which can be different from the operating system process. Most of the MPI collective operations defined in MPI-3.0 are listed below.

- Barrier

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

  The routine blocks the caller until all processes in the given communicator have called it. In other words, the function is used for synchronization purposes between MPI processes in the given communicator.

- Broadcast

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

  Root process broadcasts a message to all processes in the given communicator, itself included.

- Gather(v)

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

  The routine gathers constant size of *sendbuf* of each process in the given communicator into root including root itself.

```
int MPI_Gatherv(const void* sendbuf, int sendcount,
                MPI_Datatype sendtype, void* recvbuf,
                const int recvcounts[], const int displs[],
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_Gatherv allows root to receive a varying count of data from each process. This is done by providing the new argument, *displs*.

- Scatter(v)

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

MPI_Scatter is very similar to MPI_Bcast and inverse operation to MPI_Gather. The message is split into n equal chunks at root and chunks of an array are sent to processes in a given communicator.

```
int MPI_Scatterv(const void* sendbuf, const int sendcounts[],
                const int displs[], MPI_Datatype sendtype,
                void* recvbuf, int recvcount,
                MPI_Datatype recvtype, int root, MPI_Comm comm)
```

The routine sends the varying counts of elements to each process by providing new arguments, *sendcounts* and *displs*.

- Reduce

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

MPI_Reduce combines the elements provided in the input buffer of each process in the given communicator using the operation *op*. Predefined or user-defined operation can be used in MPI_Reduce. The value of output is a single n-element vector owned by root.

- Allreduce

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

MPI_Allreduce is very similar to MPI_Reduce. The difference is that all processes receive the same copy of the result vector in case of MPI_Allreduce.

- Allgather(v)

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

MPI_Allgather collects *sendbuf* from all processes in a given communicator and stores the data collected in the *recvbuf* of each process.

```
int MPI_Allgatherv(const void* sendbuf, int sendcount,
                  MPI_Datatype sendtype, void* recvbuf,
                  const int recvcounts[], const int displs[],
                  MPI_Datatype recvtype, MPI_Comm comm)
```

MPI_Allgatherv can be thought of as MPI_Gatherv, but where all processes receive the result, instead of just the root.

- Alltoall(v)

```
int MPI_Alltoall(const void *sendbuf, int sendcount,
              MPI_Datatype sendtype,void *recvbuf,
              int recvcount, MPI_Datatype recvtype,
              MPI_Comm comm)
```

MPI_Alltoall performs like MPI_Allgather where each process sends distinct data to each of the receivers.The j-th block sent from process i

is received by process j and is placed in the i-th block of recvbuf. The amount of data sent must be equal to the amount of data received.

```
int MPI_Alltoallv(const void* sendbuf, const int sendcounts[],
                  const int sdispls[], MPI_Datatype sendtype,
                  void* recvbuf, const int recvcounts[],
                  const int rdispls[], MPI_Datatype recvtype,
                  MPI_Comm comm)
```

MPI_Alltoallv is the flexible version of MPI_Alltoall. The flexibility is provided by new arguments, *sdispls* and *rdispls*. *Sdispls* and *rdispls* arguments specify the location of data for the send and data on the receive respectively.

- Reduce-Scatter(block)

```
int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,
                       int* counts, MPI_Datatype datatype,
                       MPI_Op operation, MPI_Comm communicator)
```

As the name implies, the routine first performs a global element-wise reduction of the vector then the result is scattered to the processes of the group.

```
int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,
                         int recvcount, MPI_Datatype datatype,
                         MPI_Op op, MPI_Comm comm)
```

The routine is equivalent to an MPI_Reduce with count equal to *recvcount*n*, followed by an MPI_Scatter operation with *sendcount* equal to *recvcount*.

- Scan

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

MPI_Scan performs a prefix reduction across all processes in the given communicator.

- Exscan

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

Like MPI_Scan, MPI_Exscan performs a prefix reduction across all MPI processes in the given communicator but excludes the calling MPI process.

All collection operations listed above are *blocking* collective communications. Non-blocking collective operations are out of scope in this work.

2.4 MPI implementations

2.4.1 Open MPI

Open MPI [6, 7] is an open source high-performance MPI implementation. While the Open MPI project started as a collaborative effort of four different MPI implementation teams: FT-MPI [43], LA-MPI [44], LAM/MPI [45], and PACX-MPI [46], currently, the project has 57 contributors which consist of 16 individuals and 41 organizations.

Open MPI is implemented using a Modular Component Architecture (MCA) [47]. It is composed of three main components: the MPI layer (OMPI), the runtime environment (ORTE), and the portability layer (OPAL). The OMPI layer is designed to provide MPI API semantics such as point-to-point communication, collective algorithms, memory allocation so forth. The ORTE frameworks provide resource manager, error manager, RTE message layer, I/O forwarding and so forth. The last component, OPAL, provides many useful functions such as debugging call stack backtrace support, checkpoint and restart service, runtime memory checking, processor affinity and so forth.

MCA provides internal parameters to the MPI programmers to configure components and frameworks.

The framework called *COLL* in OpenMPI implements collective communication routines. The decision functions to select collective algorithms are implemented in this framework. As we presented in Chapter 1, Open MPI uses Empirically Derived Decision Functions. In this thesis, we use Open MPI 3.1 [48] to validate our approach.

Open MPI is one of the main-stream MPI implementations used in HPC. Many supercomputers and clusters are shipped with Open MPI. For example, SUMMIT [49] and SIERRA [50] supercomputers developed by IBM are shipped with IBM® Spectrum MPI [51], the high-performance MPI implementation based on the Open MPI.

2.4.2 MPICH

MPICH [5] is a high performance and portable implementation of the MPI standard. Version 3.4.1 [19] is the new stable version of MPICH.

For each collective operation one or more collective algorithms are implemented in MPICH. The popular algorithmic patterns, such as Linear tree, Bruck, Recursive doubling, Binomial tree, Ring, are used in implementation of various collectives, such as Allgahter, Alltoall, Allreduce and so forth [9, 27]. Several HPC vendors such as Intel [52], HP(Cray) [53] and Microsoft [54] implemented their own MPI library based on MPICH specification.

2.5 Alternatives To MPI

2.5.1 Partitioned Global Address Space

A partitioned global address space (PGAS) is a parallel programming model where a number of parallel processes execute an algorithm by communicating with one another via the global memory space that is conceptually partitioned among all processes. Starting from the late 1990s,

dozen PGAS languages have been designed and implemented. We overview several well-known PGAS languages below.

Co-Array Fortran (CAF) [55] emerged in 1998 and it is one of the earliest PGAS languages. When CAF was designed the main idea was to create a parallel programming language which allows converting Fortran 95 code into robust parallel code with small changes. The main idea of the language is that the cost of accessing remote data should be manageable. The WG5 committee [56] decided to include co-arrays in the next Fortran Standard in May 2005 [57].

Unified Parallel C (UPC) [58] is an extension of ISO C 99 based on PGAS with a number of extensions such as parallel execution model, communication and synchronization routines, shared address space support. Any C program implemented based on ISO C 99 is also a UPC program. UPC provides private and shared variables where the last one is achieved by extending C arrays and pointers. Many parallel applications are implemented using UPC to benefit from fast one-sided communication and avoid communication [59, 60, 61].

X10 [62] is a modern object-oriented programming PGAS language developed at IBM. Due to widespread adoption of the Java language, X10 developers have decided to use the Java programming language as the foundation. The language supports both task and data parallelism. Unlike other PGAS languages such as Unified Parallel C (UPC), X10 offers a uniform way to access shared memory. The library [63] provides large-scale graph analysis algorithms and efficient distributed computing framework using X10.

Chapel [64] is a parallel programming language developed for productive parallel computing on large-scale systems by Cray. The three concepts implemented in Chapel such as (1) the global view of computation; (2) the support for both task and data-driven parallelism; (3) the separation of algorithm and data structure, made it one of the powerful PGAS languages. Chapel was designed from scratch as an object-oriented language. The main advantage is the usability of the language. Learning the Chapel language is very easy for programmers of Python, Java and C++. Chapel has been implemented using GASNet [65] framework, the language and network

independent high-performance communication framework implemented by Lawrence Berkeley National Laboratory.

2.5.2 PVM

The Parallel Virtual Machine (PVM) [66] uses the message-passing techniques to allow users to exploit distributed computing across a network. As the name implies, the main idea of PVM is to abstract a collection of computers as a single virtual machine which comes with easy to use APIs. The parallel PVM tasks are dynamic and asynchronous. It means they do not have to start altogether. The programmers can define any number of subgroups of PVM tasks to perform collective operations like MPI_Bcast, MPI_Gather and so forth. PVM also provides fault tolerance by enabling the management of dynamic groups.

Having a simple intuitive user interface has made PVM popular as an educational tool to teach distributed memory computing. PVM was designed before the MPI standard. Therefore, PVM developers have invested notably effort in designing MPI APIs.

2.5.3 mpC

The mpC language [67, 68, 69, 70] was designed to implement distributed-memory applications on heterogeneous networks by supporting task and data parallelism. The network object is the basic notation in mpC language like MPI communicator. The language allows the programmer to specify dynamically application topology. Processes in mpC applications communicate by means of message passing. The main disadvantage of mpC is the lack of components for modern clusters and supercomputers support. Besides that, it is not easy to learn mpC language.

2.5.4 HeteroMPI

HeteroMPI [71] is an extension of MPI for distributed memory computations on heterogeneous networks. It is designed based on the features in the

specification of network types of the mpC language. The main idea of HeteroMPI is to provide an automated selection of such a group of processes that executes the heterogeneous algorithm faster than any other group. In order to automate the selection, it allows the programmers to describe a performance model of their heterogeneous algorithm.

2.5.5 Charm++

Charm++ [72] is a parallel programming system designed based on C++ at the University of Illinois. Computation in Charm++ is defined in terms of collections of objects called *chare*. The chares interact via asynchronous method invocations supported by the Charm++ runtime system. Unlike MPI, the runtime system provides dynamic load balancing strategies. The load balancing strategy allows to balance priorities and workloads during the execution time. Besides that, Charm++ provides support for fault tolerance, including application-level checkpointing. Charm++ is used in many different fields such as molecular dynamics, quantum chemistry and astronomy to implement parallel applications [73, 74].

Chapter 3

Background and Related Work

This chapter falls into the following major categories: Communication Performance Models; Analytical Performance Models of MPI Collective Algorithms; Measurement of Model Parameters; Selection of Collective Algorithms. We review some of the most notable related works in these four fields and provide a comprehensive discussion.

3.1 Communication Performance Models (CPM)

All analytical performance models of collective algorithms use communication models as building blocks. In this section we describe the most commonly used communication models in detail.

Hockney model

The Hockney model [29] estimates the time $T(m)$ of sending a message of size m between two nodes as $T(m) = \alpha + \beta \cdot m$, where α and β are the message latency and the reciprocal bandwidth respectively. The original paper presents the model using asymptotic values of α and β. While the model is very simple and effective to use, it cannot model network congestion. Different authors extended this model to address this issue [75, 76]. For example, Chan et al. [75] extend the Hockney model in the presence of network conflicts. With extension of the model, the time $T(m)$ of sending a message of size m

between two nodes becomes $T = \alpha + k \cdot \beta \cdot m$, where k is the maximum number of network conflicts.

Several tools have been implemented to measure the Hockney model parameters [77, 78]. These tools measure the Hockney model parameters using simple point-to-point communication tests.

LogP/LogGP models

The LogP model [30] was introduced in 1993 by Culler et al. The name is an acronym from its four parameters, L, o, g, and P. The parameters stand for network delay, overhead, gap per message, and the number of processors respectively. The time to transmit a short message in terms of LogP is estimated as $T = L + 2o$. In LogP model, the sender is allowed to initiate new send operation after g period of time. Parameter g is the minimum time interval needed by the network card between two send operations. It means that the network allows transmission of at most $\lfloor L/g \rfloor$ messages simultaneously. The key advance of LogP is that the model recognizes the contribution of the processor to the communication latency. As it is presented above none of the LogP model parameters depends on message size. This implies that only constant-size small messages are transmitted between the nodes.

Alexandrov et al. introduce the LogGP model in [31] as an extension of the LogP. The LogP model introduces a fifth parameter G, the gap per byte, which captures the cost of sending large messages across the network. The LogGP model estimates the time to send a message of size m between two processes as $T = L + 2o + (m - 1) \cdot G$. The reciprocal $1/G$ is the network bandwidth for long messages.

The LogP model became the foundation of many subsequent models such as LogGPS [79], LogGPO [80], LogGPG [81], LogGPC [82], LogGPH [83], LoPC [84], LogfP [85] that extend it by adding parameters for representing specific characteristics of the platform . For example, LogGPC [82] extends the LogGP model taking into account the impact of network contention C_n. The time $T(m)$ of sending a message size of m in terms of LogGPC is estimated

as $T(m) = o_s + L + (m-1) \cdot G + C_n + o_r$. LogGPH models communication considering the representation of hierarchical networks using the concept of "communication level".

PLogP model

Kielmann et al. introduce the Parameterized LogP, PLogP, model in [32]. Authors build the PLogP model by transforming constant LogP parameters into piecewise linear functions of the message size m. The PLogP model is defined in terms of latency L, $o_s(m)$ and $o_r(m)$, *sender* and *receiver* overheads, gap per message g(m), and the number of processes P involved in communication. The PLogP model estimates the time $T(m)$ of sending a message of size m as $T(m) = L + g(m)$. The notion of latency in PLogP slightly differs from those of LogP/LogGP, the gap and overhead parameters are equivalent in both models. The gap parameter in the model is defined as the minimum time interval between consecutive message transmissions. This implies that at all times $g(m) \geq o_s(m)$ and $g(m) \geq o_r(m)$. Authors present a transformation from PLogP parameters to LogP/LogGP parameters in [32] as well.

All communication models presented above are very first efforts in communication modelling. These communication models were designed for homogeneous platforms in mind and tested on such platforms. Nowadays, HPC platforms (clusters/supercomputers) are powered by several thousands of CPU cores, deep memory hierarchies and advanced network technologies. Distributed memory applications face the challenge of obtaining as much performance as possible from such complex platforms. The general adoption of large scale multi-core HPC platforms has seen a rise in new approaches to communication modeling. We overview some of them in the sections below.

$\log_n P$ model

The message transmissions in distributed memory applications are provided by middleware (library) using implicit communication mechanisms. Series of implicit communications performs by middleware has impact on point-to-point

communication. Thus, Cameron et al. [86, 87] propose the $\log_n P$ model taking into account the middleware costs of the communication. The main idea of the proposed model is that the message progresses as a succession of transfers through intermediate buffers between *sender* and *receiver* buffers. The model estimates the time $T(m)$ of sending a message size of m as $T(m) = \sum_{i=0}^{n-1}(\max\{g_i, o_i\} + l_i)$ where o: the per transfer time dedicated by the CPU without resource contention, g: o plus additional system delays, l: the length of time the processor is engaged in the transmission of a the message stored in non-contiguous or strided ways, n: the number of implicit transfers between sender and receiver, P: the number of processes. The main disadvantage of the model is that it is not easy to use it. While the model helps to understand the impact of implicit communications performed by middleware, due to complexity of the model parameters it is not feasible to measure them accurately in all cases.

τ-Lop model

Rico-Gallego et al. [88] introduce the τ-Lop model to measure the communication time between sender and receiver on shared memory. The authors extend this approach for multi-core clusters in [89]. The τ-Lop model considers the distinctive features of the communication channels to represent a point-to-point transmission as a sequence of transfers via shared memory or network. The model assumes that the cost of transmission of a message of size m is estimated as $T^c_{p2p}(m) = o^c(m) + \sum_{j=0}^{s-1} L^c_j(m, \tau_j)$, where c represents the communication channel, $o^c(m)$ is the overhead of protocols and software stack, $L^c_j(m, \tau_j)$ is the time to transfer a message of size m through channel c at the j-th step of the transmission, with τ_j contending transfers ($L^c_j(0, \tau_j) = 0$), and s is the number of steps of the message transmission. Hence, the τ-Lop model considers a different representation for a message transmitted through shared memory ($c = 0$) and network channel ($c = 1$). In the shared memory channel ($c = 0$ and $s = 2$) the time $T(m)$ of sending a message size of m is estimated as $T^0_{p2p}(m) = o^0(m) + 2L^0(m, 1)$. In an Ethernet network the model considers two shared memory transfers,

from sender memory to NIC and from receiver NIC to destination memory, and a network transfer between NICs, as $T^1_{p2p}(m) = o^1(m) + 2L^0(m,1) + L^1(m,1)$.

3.1.1 Conclusion

Although the Hockney model is one of the oldest and simplest communication models in terms of representation of communication and model parameters, it is widely used to design algorithms on a variety of platforms. As a matter of fact, more complex communication models can be difficult to use in practice. Too many parameters make it challenging to measure them accurately. Given that the goal of this thesis is to derive analytical performance models for collective algorithms, taking into account details of the implementing code and executing platform, the simple Hockney model is sufficient to express communication in a homogeneous platform where MPI programs use a one-process-per-CPU configuration. For analytical performance modelling of collective algorithms where shared memory is involved in communication and processes are mapped by CPU-Core, we use the τ-Lop model.

3.2 Analytical Performance Models of MPI Collective Algorithms

It is very important to understand the performance of collective algorithms to optimise them. This section surveys the current state of analytical performance models of collective algorithms. The list of the collective algorithms used in Open MPI is given in Table 3.1. Tables 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10 and 3.11 show the existing analytical performance models for broadcast, gather, scatter, allgather, alltoall, barrier, reduce, reduce-scatter and scan collective algorithms.

Thakur et al. [27] propose analytical performance models of several collective algorithms for *MPI_Allgather, MPI_Bcast, MPI_Alltoall, MPI_Reduce_scatter, MPI_Reduce*, and *MPI_Allreduce* routines using the Hockney model. The parameters of the models, α and β, are assumed to be

Collective routine	Collective algorithm
Allgather	Ring [27], Recursive doubling [27], Bruck [12], Neighbor exchange [90]
Broadcast	Flat tree [91], Chain tree [92], Binomial [27, 91], Binary [91], Split-binary [91], K-Chain tree [91]
Barrier	Flat tree [91], Double Ring [91], Recursive doubling [91], Bruck [12]
Scater	Linear [93], Binomial [91]
Gather	Linear [91], Linear with synchronisation [91], Binomial [91]
Alltoall	Linear [91], Pairwise exchange [91], Bruck [12]
Reduce	Flat tree [91], Chain [10, 91], Binomial [91], Binary [91], Rabenseifner [94, 11]
Reduce-scatter	Reduce-scatterv [91], Recursive halving [91], Ring [91]
Allreduce	Recursive doubling [91], Ring [91], Ring with segmentation [91], Rabenseifner [94, 11]
Scan	Linear [95, 91], Linear with segmentation [91], Binomial [91]

Table 3.1: List of collective algorithms used in Open MPI

the same for all algorithms, message sizes and numbers of processes. The authors find their models not accurate enough for the task of selection of optimal collective algorithms. They conclude that in order to improve the accuracy of their analytical models, we have to assume that α and β depend on the message size and the number of processes. They do not propose models improved this way though. In our work, we stick to the assumption of independence of model parameters on the message size and the number of processes. Instead, we improve the accuracy of our models by deriving them from the implementation of the modelled algorithms. In addition, we assume that α and β may depend on the algorithm. Thus, our approach to improving the accuracy of models of collective algorithms is to make them more algorithm and implementation specific.

Chan et al. [75] build analytical performance models of *Minimum-spanning tree* algorithms and *Bucket* algorithms for MPI_Bcast, MPI_Redcue, MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Reduce_scatter, MPI_Allreduce collectives and later extend this work for multidimensional mesh architecture in [96]. The proposed models are built using high-level theoretical descriptions of the algorithms. Therefore, the

authors conclude that while the models can be used for analysis of theoretical complexity of the algorithms, they are not accurate enough for the task of estimation and comparison of their practical performance.

An analytical performance model of a new *reduction* algorithm is proposed for a non-power-of-two number of processes by Rabenseifner et al. [11]. The model uses a traditional high-level mathematical description of the algorithm. The aim of the model is to understand and express the complexity of the algorithm. Like in all previous models, its level of abstraction is too high to reach the accuracy required for comparison of the practical performance of the proposed reduction algorithm with its counterparts.

A general analytical performance model for *tree-based* broadcast algorithms with message segmentation has been proposed by Patarasuk et al. [92]. Unlike traditional models, this model introduces a new parameter, *Maximum nodal degree* of the tree. The purpose of this model is restricted to theoretical comparison of different tree-based broadcast algorithms. Accurate prediction of the execution time of the broadcast algorithms and methods for measurement of the model parameters, including the maximal nodal degree of the tree, are out of the scope of their work.

Barchet-Estefanel et al. evaluate performance of inter-cluster and intra-cluster collectives using the PLogP model in [102]. The authors extend the same approach including *contention* and *supplemental* factors in [103]. The latest work validates the proposed approach in different network environments using All-to-All collective algorithms.

Hasanov et al. propose high-level hierarchical topology-oblivious optimization of MPI broadcast algorithms in [99]. P MPI processes executing the broadcast are splitted into G logical groups each of which has $\frac{P}{G}$ number of processes. The approach is designed as a composition of two steps: (1) Broadcast operation is performed among the first elements of groups. (2) The first element of each group broadcasts a message inside the group. Following this approach, the authors developed new hierarchical broadcast algorithms using hierarchical arrangement of MPI processes. While the approach is validated experimentally, theoretical analysis of the approach is done using the Hockney communication model as well. The performance models of

Broadcast	CPM	Performance model of the algorithm	Reference
Linear	Hockney	$T = (P-1)\cdot(\alpha + m\cdot\beta)$	[27], [75, 28]
Linear	LogP/LogGP	$T = L + 2\cdot o - g + n_s\cdot(P-1)\cdot((m_s-1)\cdot G + g)$	[28, 91]
Linear	PLogP	$T = L + n_s\cdot(P-1)\cdot g(m_s)$	[97, 98]
Chain	Hockney	$T = (P + n_s - 2)\cdot(\alpha + m\cdot\beta)$	[92, 91]
Chain	LogP/LogGP	$T = (P-1)\cdot(L + 2\cdot(m_s-1)\cdot G) + (n_s-1)\cdot(g + (m_s-1)\cdot G + o)$	[91]
Chain	PLogP	$T = (P-1)\cdot(L + g(m_s)) + (n_s-1)\cdot g(m_s)$	[98]
Binary	Hockney	$T = 2\cdot(\lceil\log_2(P+1)\rceil + n_s - 2)\cdot(\alpha + m\cdot\beta)$	[91]
Binary	LogP/LogGP	$T = (\lceil\log_2(P+1)\rceil - 1)\cdot(L + g + 2\cdot(o + (m_s-1)\cdot G)) + (n_s-1)\cdot(o + 2\cdot(g + (m_s-1)\cdot G))$	[30]
Binary	PLogP	$T = (\lceil\log_2(P+1)\rceil - 1)\cdot(L + 2\cdot g(m_s)) + (n_s-1)\cdot\max\{2\cdot g(m_s), o_r(m_s) + g(m_s) + o_s(m_s)\}$	[98, 97]
Split-binary	Hockney	$T = 2\cdot(\lfloor\log_2 P\rfloor + \frac{n_s}{2} - 1)\cdot(\alpha + m_s\cdot\beta) + (\alpha + \frac{m}{2}\cdot\beta)$	[91]
Split-binary	LogP/LogGP	$T = (\lceil\log_2(P+1)\rceil - 1)\cdot(L + g + 2\cdot o + (m_s-1)\cdot G) + (\frac{n_s}{2} - 1)\cdot(o + 2\cdot(g + (m_s-1)\cdot G)) + L + 2\cdot o + (\frac{m}{2} - 1)\cdot G$	[91]
Split-binary	PLogP	$T = (\lceil\log_2(P+1)\rceil - 1)\cdot(L + 2\cdot g(m_s)) + (\frac{n_s}{2} - 1)\cdot \max\{2\cdot g(m_s), o_r(m_s) + g(m_s) + o_s(m_s)\} + L + g(\frac{m}{2})$	[91]
Binomial	Hockney	$T = \lceil\log_2 P\rceil\cdot n_s\cdot(\alpha + m_s\cdot\beta)$	[91]
Binomial	LogP/LogGP	$T = \lceil\log_2 P\rceil\cdot(L + 2\cdot o + (m_s-1)\cdot G + (n_s-1)\cdot(g + (m_s-1)\cdot G))$	[30]
Binomial	PLogP	$T = \lceil\log_2 P\rceil\cdot(L + n_s\cdot g(m_s))$	[98, 97]

Table 3.2: Existing analytical performance models of broadcast algorithms.

Gather	CPM	Performance model of the algorithm	Reference
Linear without sync.	Hockney	$T = (P-1) \cdot (\alpha + m \cdot \beta)$	[27], [75]
Linear without sync.	LogP/LogGP	$T = L + 2 \cdot o + (P-2) \cdot ((m-1) \cdot G + g)$	[28, 91]
Linear without sync.	PLogP	$T = L + (P-1) \cdot g(m)$	[97, 98]
Linear with sync.	Hockney	$T = \alpha + (P-1) \cdot (2 \cdot \alpha + \beta \cdot m)$	[91]
Linear with sync.	LogP/LogGP	$T = \begin{cases} (P-1) \cdot (2 \cdot L + 4 + (m_s - 1) \cdot G) + g + (m - m_s - 1) \cdot G, \\ \qquad \text{if } g + m \cdot G < L + 2 \cdot (o + m_s \cdot G) \\ (P-1) \cdot (o + g + (m-1) \cdot G + L + o), \text{otherwise} \end{cases}$	[91]
Linear with sync.	PLogP	$T = \begin{cases} T = (P-1) \cdot (2 \cdot L + g(m_s) + g(0)) + g(m - m_s)), \\ \qquad \text{if } g(m - m_s) < L + g(0) \\ T = 2 \cdot L + g(0) + (P-1) \cdot (g(m_s) + g(m - m_s)), \text{otherwise} \end{cases}$	[91]
Binomial	Hockney	$T = \log_2 P \cdot \alpha + P \cdot \beta \cdot m$	[91]
Binomial	LogP/LogGP	$T = (\lfloor \log_2 P \rfloor - 1) \cdot (L + 2 \cdot o) + ((2 \cdot P - 1) \cdot m - \lfloor \log_2 P \rfloor) \cdot G$	[91]
Binomial	PLogP	$T = \sum_{k=0}^{\lfloor \log_2 P \rfloor - 1} (L + g(2^k \cdot m))$	[98]

Table 3.3: Existing analytical performance models of gather algorithms.

Scatter	CPM	Performance model of the algorithm	Reference
Linear	Hockney	$T = (P-1) \cdot (\alpha + m \cdot \beta)$	[27, 75]
Linear	LogP/LogGP	$T = L + 2 \cdot o + (P-2) \cdot ((m-1) \cdot G + g)$	[28, 91]
Linear	PLogP	$T = L + (P-1) \cdot g(m)$	[97, 98]
Binomial	Hockney	$T = \log_2 P \cdot \alpha + (P-1) \cdot \beta \cdot m$	[91]
Binomial	LogP/LogGP	$T = L + 2 \cdot o + (m \cdot (P-1) - \lceil \log_2 P \rceil) \cdot G + (\lceil \log_2 P \rceil - 1) \cdot \max\{L + 2 \cdot o, g\}$	[28, 91]
Binomial	PLogP	$T = \sum_{k=0}^{\lfloor \log_2 P \rfloor - 1} (L + g(2^k \cdot m))$	[98]

Table 3.4: Existing analytical performance models of scatter algorithms.

Allgather	CPM	Performance model of the algorithm	Reference
Bruck	Hockney	$T = \log_2 P \cdot \alpha + P \cdot \beta \cdot m + P \cdot \delta \cdot m$	[27, 75]
Bruck	LogP/LogGP	$T = \lfloor \log_2 P \rfloor \cdot (o + \max\{g, L + o\} - G) + (P-1) \cdot m \cdot G + P \cdot \delta \cdot m$	[91]
Bruck	PLogP	$T = \sum_{k=0}^{\lfloor \log_2 P \rfloor - 1} (L + g(2^k \cdot m)) + L + g \cdot ((P - \sum_{k=0}^{\lfloor \log_2 P \rfloor - 1} 2^k) \cdot m) + P \cdot \delta \cdot m$	[91]
Recursive doubling	Hockney	$T = \log_2 P \cdot \alpha + (P-1) \cdot \beta \cdot m$	[27, 91]
Recursive doubling	LogP/LogGP	$T = \log_2 P \cdot (o + \max\{g, L + o\} - G) + (P-1) \cdot m \cdot G$	[91]
Recursive doubling	PLogP	$T = \sum_{k=0}^{\log_2 P - 1} (L + g(2^k \cdot m))$	[91]
Ring	Hockney	$T = (P-1) \cdot (\alpha + \beta \cdot m)$	[27, 91]
Ring	LogP/LogGP	$T = (P-1) \cdot (L + 2 \cdot o + (m-1) \cdot G)$	[91]
Ring	PLogP	$T = (P-1) \cdot (L + g(m))$	[91]
Neighbor exchange	Hockney	$T = \alpha + \beta \cdot m + (\frac{P}{2} - 1) \cdot (\alpha + 2 \cdot \beta \cdot m)$	[75, 91]
Neighbor exchange	LogP/LogGP	$T = \frac{P}{2} \cdot (L + 2 \cdot o + (2 \cdot m - 1) \cdot G) - m \cdot G$	[91]
Neighbor exchange	PLogP	$T = L + g(m) + (\frac{P}{2} - 1) \cdot g(2 \cdot m)$	[91]

Table 3.5: Existing analytical performance models of allgather algorithms.

Alltoall	**CPM**	**Performance model og the algorithm**	**Reference**
Linear	Hockney	$T = (P-1)\cdot(\alpha+\beta\cdot m)$	[27]
Linear	LogP/LogGP	$T = L+2\cdot o+(m-1)\cdot G+2\cdot(P-1)\cdot g$	[30]
Linear	PLogP	$T = L+2\cdot(P-1)\cdot g(m)$	[91]
Pairwise exchange	Hockney	$T = (P-1)\cdot(\alpha+\beta\cdot m)$	[27, 91]
Pairwise exchange	LogP/LogGP	$T = (P-1)\cdot(L+o+(m-1)\cdot G+g)$	[91]
Pairwise exchange	PLogP	$T = (P-1)\cdot(L+g(m))$	[91]
Bruck	Hockney	$T = \lceil\log_2 P\rceil\cdot\alpha+\lfloor\log_2 P\rfloor\cdot(\beta+\delta)\cdot\frac{P}{2}\cdot m+\delta\cdot P\cdot m+ (\beta+\delta)\cdot(P-2^{\lfloor\log_2 P\rfloor})\cdot m$	[27, 12]
Bruck	LogP/LogGP	$T=\begin{cases}\log_2 P\cdot(o+(\frac{P}{2}\cdot m-1)\cdot G+\delta\cdot\frac{P}{2}\cdot m+\max\{g,L+o\})+ \\ \delta\cdot P\cdot m, \text{if } P=2^k \\ \\ \lfloor\log_2 P\rfloor\cdot(o+(\frac{P}{2}\cdot m-1)\cdot G+\delta\cdot\frac{P}{2}\cdot m+\max\{g,L+o\})+ \\ \delta\cdot P\cdot m+o+((P-2^{\lfloor\log_2 P\rfloor})\cdot m-1)\cdot G+ \\ \delta\cdot(P-2^{\lfloor\log_2 P\rfloor})\cdot m+\max\{g,L+o\}, \text{otherwise}\end{cases}$	[91]
Bruck	PLogP	$T = \lceil\log_2 P\rceil\cdot L+\lfloor\log_2 P\rfloor\cdot(g(\frac{P}{2}\cdot m)+\delta\cdot\frac{P}{2}\cdot m) +\delta\cdot P\cdot m+g\cdot((P-2^{\lfloor\log_2 P\rfloor})\cdot m)+ \delta\cdot(P-2^{\lfloor\log_2 P\rfloor})\cdot m$	[91]

Table 3.6: Existing analytical performance models of alltoall algorithms.

Barrier	CPM	Performance model of the algorithm	Reference
Linear	Hockney	$T = (P-1) \cdot \alpha$	[91]
Linear	LogP/LogGP	$T_{min} = (P-2) \cdot g + 2 \cdot (L + 2 \cdot o)$ $T_{max} = (P-2) \cdot (g+o) + 2 \cdot (L + 2 \cdot o)$	[91]
Linear	PLogP	$T_{min} = P \cdot g + 2 \cdot L$ $T_{max} = P \cdot (g + o_r) + 2 \cdot (L - o_r)$	[91]
Double Ring	Hockney	$T = 2 \cdot P \cdot \alpha$	[91]
Double Ring	LogP/LogGP	$T = 2 \cdot P \cdot (L + o + g)$	[91]
Double Ring	PLogP	$T = 2 \cdot P \cdot (L + g)$	[91]
Recursive Doubling	Hockney	$T = \begin{cases} \log_2 P \cdot \alpha, \text{if } P = 2^k \\ (\log_2 P + 2) \cdot \alpha, \text{if } P \neq 2^k \end{cases}$	[27]
Recursive Doubling	LogP/LogGP	$T = \begin{cases} \log_2 P \cdot (L + o + g), \text{if } P = 2^k \\ (\lfloor \log_2 P \rfloor + 2) \cdot (L + o + g), \text{if } P \neq 2^k \end{cases}$	[91]
Recursive Doubling	PLogP	$T = \begin{cases} \log_2 P \cdot (L + g), \text{if } P = 2^k \\ (\lfloor \log_2 P \rfloor + 2) \cdot (L + g), \text{if } P \neq 2^k \end{cases}$	[91]
Bruck	Hockney	$T = \lceil \log_2 P \rceil \cdot \alpha$	[27]
Bruck	LogP/LogGP	$T = \lceil \log_2 P \rceil \cdot (L + o + g)$	[91]
Bruck	PLogP	$T = \lceil \log_2 P \rceil \cdot (L + g)$	[91]

Table 3.7: Existing analytical performance models of barrier algorithms.

Reduce	CPM	Performance model of the algorithm	Reference
Flat tree	Hockney	$T = n_s \cdot (P-1) \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s)$	[27, 75]
Flat tree	LogP/LogGP	$T = o + (m_s - 1) \cdot G + L + n_s \cdot \max\{g, (P-1) \cdot (o + (m_s - 1) \cdot G + \gamma \cdot m_s)\}$	[91]
Flat tree	PLogP	$T = L + (P-1) \cdot n_s \cdot \max\{g(m_s), o_r(m_s) + \gamma \cdot m\}$	[97]
Chain	Hockney	$T = (P + n_s - 2) \cdot (\alpha + \beta \cdot m + \gamma \cdot m_s)$	[91]
Chain	LogP/LogGP	$T = (P-1) \cdot (L + 2 \cdot o + (m_s - 1) \cdot G + \gamma \cdot m_s) + (n_s - 1) \cdot \max\{g, 2 \cdot o + (m_s - 1) \cdot + \gamma \cdot m_s\}$	[91]
Chain	PLogP	$T = (P-1) \cdot (L + \max\{g(m_s), o_r(m_s) + \gamma \cdot m_s\}) + (n_s - 1) \cdot (\max\{g(m_s), o_r(m_s) + \gamma \cdot m_s\} + o_s(m_s))$	[91]
Binomial	Hockney	$T = n_s \cdot \lceil \log_2 P \rceil \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s)$	[27, 75]
Binomial	LogP/LogGP	$T = \lceil \log_2 P \rceil \cdot (o + L + n_s \cdot ((m_s - 1) \cdot G + \max\{g, o + \gamma \cdot m_s\}))$	[30, 31]
Binomial	PLogP	$T = \lceil \log_2 P \rceil \cdot (L + n_s \cdot ((m_s - 1) \cdot G + \max\{g(m_s), o_r(m_s) + \gamma \cdot m_s + o_s(m_s)\}))$	[91]
Binary	Hockney	$T = 2 \cdot (\lceil \log_2(P+1) \rceil + n_s - 2) \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s)$	[75, 27]
Binary	LogP/LogGP	$T = (\lceil \log_2(P+1) \rceil - 1) \cdot (L + 3 \cdot o + (m_s - 1) \cdot G + 2 \cdot \gamma \cdot m_s + (n_s - 1) \cdot ((m_s - 1) \cdot G + \max\{g, 3 \cdot o + 2 \cdot \gamma \cdot m_s\}))$	[30, 31]
Binary	PLogP	$T = (\lceil \log_2(P+1) \rceil - 1) \cdot (L + 2 \cdot \max\{g(m_s), o_r(m_s) + \gamma \cdot m_s\}) + (n_s - 1) \cdot (o_s(m_s) + 2 \cdot \max\{g(m_s), o_r(m_s) + \gamma \cdot m_s\})$	[91]
Rabenseifner	Hockney	$T = 2 \cdot \log_2 P \cdot \alpha + 2 \cdot (P-1) \cdot \beta \cdot m + (P-1) \cdot \gamma \cdot m$	[27, 11]
Rabenseifner	LogP/LogGP	$T = 2 \cdot \log_2 P \cdot (L + 2 \cdot o) + 2 \cdot ((P-1) \cdot m - \log_2 P) \cdot G + (P-1) \cdot \gamma \cdot m$	[91]
Rabenseifner	PLogP	$T = 2 \cdot \log_2 P \cdot L + (P-1) \cdot \gamma \cdot m + \sum_{k=1}^{\log_2 P} g(\frac{m}{2^k})$	[91]

Table 3.8: Existing analytical performance models of reduce algorithms.

Reduce-scatter	CPM	Performance model of the algorithm	Reference
Recursive halving	Hockney	$T = \begin{cases} \log_2 P \cdot \alpha + (P-1) \cdot (\beta + \delta) \cdot m, \text{if } P = 2^k \\ (\lfloor \log_2 P \rfloor + 2) \cdot \alpha + 2 \cdot P \cdot \beta \cdot m + (2 \cdot P - 1) \cdot \delta \cdot m, \text{if } P \neq 2^k \end{cases}$	[27]
Recursive halving	LogP/LogGP	$T = \begin{cases} \log_2 P \cdot (L + 2 \cdot o) + ((P-1) \cdot m - \log_2 P) \cdot G + \\ (P-1) \cdot \gamma \cdot m, \text{if } P = 2^k \\ (\lfloor \log_2 P \rfloor + 2) \cdot (L+2) + (2 \cdot (P \cdot m - 1) - \lfloor \log_2 P \rfloor) \cdot G + \\ (2 \cdot P - 1) \cdot \gamma \cdot m, \text{if } P \neq 2^k \end{cases}$	[91]
Recursive halving	PLogP	$T = \begin{cases} \log_2 P \cdot L + (P-1) \cdot \gamma \cdot m + \sum_{k=1}^{\log_2 P} g(\frac{P}{2^k}), \text{if } P = 2^k \\ (\lfloor \log_2 P \rfloor + 2) \cdot L + g(m) + (2 \cdot P - 1) \cdot \gamma \cdot m + \sum_{k=1}^{\log_2 P} g(\frac{P}{2^k}), \\ \text{if } P \neq 2^k \end{cases}$	[91]
Ring	Hockney	$T = (P-1) \cdot (\alpha + \beta \cdot m + \gamma \cdot m) + P \cdot \delta \cdot m$	[91]
Ring	LogP/LogGP	$T = (P-1) \cdot (L + 2 \cdot o + (m-1) \cdot G + \gamma \cdot m) + P \cdot \delta \cdot m$	[91]
Ring	PLogP	$T = (P-1) \cdot (L + g(m) + \gamma \cdot m) + P \cdot \delta \cdot m$	[91]

Table 3.9: Existing analytical performance models of reduce-scatter algorithms.

Scan	CPM	Performance model of the algorithm	Reference
Linear	Hockney	$T = (P-1) \cdot (\alpha + \beta \cdot m + \gamma \cdot m)$	[27]
Linear	LogP/LogGP	$T = (P-1) \cdot (L + 2 \cdot o + (m-1) \cdot G + \gamma \cdot m)$	[91]
Linear	PLogP	$T = (P-1) \cdot (L + g(m) + \gamma \cdot m)$	[91]
Linear with segment.	Hockney	$T = (P + n_s - 2) \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s)$	[91]
Linear with segment.	LogP/LogGP	$T = (P-1) \cdot (L + 2 \cdot o + (m_s - 1) \cdot G + \gamma \cdot m_s) + (n_s - 1) \cdot (\max\{g, 2 \cdot o + (m_s - 1) \cdot G + \gamma \cdot m_s\})$	[91]
Linear with segment.	PLogP	$T = (P-1) \cdot (L + g(m_s) + \gamma \cdot m_s) + (n_s - 1) \cdot (\max\{g(m_s), o(m_s) + o_r(m_s) + \gamma \cdot m_s\})$	[91]
Binomial	Hockney	$T = \lceil \log_2 P \rceil \cdot (\alpha + \beta \cdot m + \gamma \cdot m)$	[91]
Binomial	LogP/LogGP	$T = \lceil \log_2 P \rceil \cdot (L + 2 \cdot o + (m-1) \cdot G + \max\{g, o + \gamma \cdot m\})$	[91]
Binomial	PLogP	$T = \lceil \log_2 P \rceil \cdot (L + g(m) + \max\{g, o_s(m) + \gamma \cdot m\})$	[91]

Table 3.10: Existing analytical performance models of scan algorithms.

Allreduce	**CPM**	**Performance model of the algorithm**	**Reference**
Recursive doubling	Hockney	$T = \begin{cases} \log_2 P \cdot (\alpha + \beta \cdot m + \gamma \cdot m), \text{if } P = 2^k \\ (\lfloor \log_2 \rfloor + 2) \cdot (\alpha + \beta \cdot m + \gamma \cdot m) - \gamma \cdot m, \\ \text{if } P \neq 2^k \end{cases}$	[27]
Recursive doubling	LogP/LogGP	$T = \begin{cases} \log_2 P \cdot (L + 2 \cdot o + (m-1) \cdot G + \gamma \cdot m), \text{if } P = 2^k \\ (\lfloor \log_2 \rfloor + 2) \cdot (L + 2 \cdot o + (m-1) \cdot G + \gamma \cdot m) - \gamma \cdot m, \\ \text{if } P \neq 2^k \end{cases}$	[91]
Recursive doubling	PLogP	$T = \begin{cases} \log_2 P \cdot (L + g(m) + \gamma \cdot m), \text{if } P = 2^k \\ (\lfloor \log_2 \rfloor + 2) \cdot (L + g(m) + \gamma \cdot m) - \gamma \cdot m, \text{if } P \neq 2^k \end{cases}$	[91]
Ring	Hockney	$T = 2 \cdot (P-1) \cdot (\alpha + \beta \cdot \lceil \frac{m}{P} \rceil) + (P-1) \cdot \gamma \cdot \lceil \frac{m}{P} \rceil$	[91]
Ring	LogP/LogGP	$T = 2 \cdot (P-1) \cdot (L + 2 \cdot o + (\lceil \frac{m}{P} \rceil - 1) \cdot G) + (P-1) \cdot \gamma \cdot \lceil \frac{m}{P} \rceil$	[91]
Ring	PLogP	$T = 2 \cdot (P-1) \cdot (L + g(\lceil \frac{m}{P} \rceil)) + (P-1) \cdot \gamma \cdot \lceil \frac{m}{P} \rceil$	[91]
Ring with segment.	Hockney	$T = (P + n_s - 2) \cdot (\alpha + \beta \cdot m_s + \gamma \cdot m_s) + (P-1) \cdot (\alpha + \beta \cdot \lceil \frac{m}{P} \rceil)$	[91]
Ring with segment.	LogP/LogGP	$T = (P-1) \cdot (L + 2 \cdot o + (m_s - 1) \cdot G) + (n_s - 1) \cdot (\max\{g, (\gamma \cdot m_s + o)\} + (m_s - 1) \cdot G) + (P-1) \cdot (L + 2 \cdot o + (\lceil \frac{m}{P} \rceil - 1) \cdot G)$	[91]
Ring with segment.	PLogP	$T = (P-1) \cdot (L + g(m_s) + \gamma \cdot m_s) + (n_s - 1) \cdot (g(m_s) + \gamma \cdot m_s) + (P-1) \cdot (L + g(\lceil \frac{m}{P} \rceil))$	[91]
Rabenseifner	Hockney	$T = 2 \cdot \log_2 P \cdot \alpha + 2 \cdot \frac{P-1}{P} \cdot \beta \cdot m + \frac{P-1}{P} \cdot \gamma \cdot m$	[27]
Rabenseifner	LogP/LogGP	$T = 2 \cdot \log_2 P \cdot (L + 2 \cdot o) + 2 \cdot ((P-1) \cdot m - \log_2 P) \cdot G + (P-1) \cdot \gamma \cdot \frac{m}{P} + \log_2 P \cdot (o + \max\{g, L + o\} - G) + (P-1) \cdot \frac{m}{P} \cdot G$	[91]
Rabenseifner	PLogP	$T = 3 \cdot \log_2 P \cdot L + (P-1) \cdot \gamma \cdot \frac{m}{P} + 3 \cdot \sum_{k=1}^{\log_2 P - 1} g(\frac{m}{2^k})$	[91]

Table 3.11: Existing analytical performance models of allreduce algorithms.

Hierarchical broadcast	Performance model of the algorithm	Reference
Hierarchical Linear	$T(G) = (G + \frac{P}{G} - 2) \cdot (\alpha + m \cdot \beta)$	[99]
Hierarchical Chain	$T(G) = (2 \cdot n_s + G + \frac{P}{G} - 4) \cdot (\alpha + m_s \cdot \beta)$	[99]
Hierarchical Split-binary	$T(G) = 2 \cdot (\log_2(P + G) - 4) \cdot (\alpha + \beta \cdot \frac{m}{2}) + 2 \cdot (\alpha + \frac{m}{2} \cdot \beta)$	[99]
Hierarchical Scatter-Ring-Allgather	$T(G) = (\log_2 P + G + \frac{P}{G} - 2) \cdot \alpha + 2 \cdot m \cdot (2 - \frac{1}{G} - \frac{G}{P}) \cdot \beta$	[99]
Hierarchical Scatter-Recursive-Doubling-Allgather	$T(G) = 2 \cdot \log_2 P \cdot \alpha + 2 \cdot m \cdot (2 - \frac{1}{G} - \frac{G}{P}) \cdot \beta$	[99]

Table 3.12: Analytical performance models of the hierarchical broadcast algorithms.

Hierarchical reduce	Performance model of the algorithm	Reference
Hierarchical Flat	$T(G) = (G + \frac{P}{G} - 2) \cdot (\alpha + m \cdot \beta + m \cdot \gamma)$	[100]
Hierarchical Chain	$T(G) = (2 \cdot n_s + G + \frac{P}{G} - 4) \cdot (\alpha + m_s \cdot \beta + m_s \cdot \gamma)$	[99]
Hierarchical Rabenseifner's Reduce	$T(G) = 2 \cdot \log_2 P \cdot \alpha + 2 \cdot m\beta \cdot (2 - \frac{G}{P} - \frac{1}{G}) + m \cdot \gamma \cdot (2 - \frac{G}{P} - \frac{1}{G})$	[99]

Table 3.13: Analytical performance models of the hierarchical reduce algorithms.

Hierarchical scatter & gather	Performance model of the algorithm	Reference
Hierarchical Linear	$T(G) = (G + \frac{P}{G} - 2) \cdot (\alpha + m \cdot \beta)$	[101]
Hierarchical Linear with synch.	$T(G) = 2 \cdot \alpha + (G + \frac{P}{G} - 2) \cdot (2 \cdot \alpha + m \cdot \beta)$	[101]
Hierarchical Binomial	$T(G) = \log_2 P \cdot \alpha + (G - 1) \cdot m \cdot \beta + \log_2 \frac{P}{G} \cdot \alpha + (\frac{P}{G} - 1) \cdot m \cdot \beta$	[101]
Hierarchical Minimum Spanning	$T(G) = \log_2 P \cdot \alpha + (2 - \frac{1}{G} - \frac{G}{P}) \cdot m \cdot \beta$	[101]

Table 3.14: Analytical performance models of the hierarchical gather algorithms.

hierarchical broadcast algorithms are built using high-level mathematical description of the native broadcast algorithms and used only for theoretical analysis of the algorithms. The same approach is applied to the MPI reduce algorithms in [100], the MPI gather algorithms in [101]. All analytical performance models of the hierarchical collective algorithms are given in Table 3.12, 3.13 and 3.14. Hierarchical broadcast algorithms are applied to SUMMA [104] in [105, 106].

Pjevsivac-Grbovic et al. [28] study selection of optimal collective algorithms using analytical performance models for *barrier*, *broadcast*, *reduce* and *alltoall* collective operations. Analytical performance models are built using the Hockney, LogP/LogGP, and PLogP communication performance models. Additionally, the splitted-binary broadcast algorithm has been designed and analysed with different performance models in this work. The models are built up with the traditional approach using high-level mathematical definitions of the collective algorithms. In order to predict the cost of a collective algorithm by analytical formula, model parameters are measured using point-to-point communication experiments. After experimental validation of their modelling approach, the authors conclude that the proposed models are not accurate enough for selection of optimal algorithms.

Lastovetsky et al. [107] propose a communication performance model for heterogeneous clusters. The model assumes that time to transmit a message between two nodes in a heterogeneous cluster is composed of the network transmission delay, source and destination processing delays. The analytical performance model of the binomial broadcast algorithm is built up using this model taking into account the impact of message passing protocols. While the predicted execution time of the binomial broadcast algorithm was close to the experimentally measured time, its use for comparison of practical performance of broadcast algorithms has never been studied.

Lastovetsky and O'Flynn [108] propose a non-deterministic model of MPI_Gather for MPI platforms on a swithched Ethernet network.The model reflects a significant non-deterministic increase in the execution time for medium-sized messages, persistently observed for different parallel platforms

and MPI implementations and not reflected in traditional communication performance models.

3.3 Measurement of Model Parameters

One of the uses of analytical communication performance models is for theoretical analysis of the complexity of collective algorithms. In such purely theoretical studies, the authors do not pay much attention to methods of measurement of model parameters. However, if a model is intended for accurate prediction of the execution time of the communication algorithm on each particular platform, a well-defined experimental measurement method of the model parameters will be as important as the theoretical formulation of the model. Different measurement methods may give significantly different values of the model parameters and therefore either degrade or improve the model's prediction accuracy.

In general, a typical measurement method consists of a well-defined set of communication experiments, each of which is used to obtain an equation with model parameters as unknowns on one side of the equation and the measured execution time of the experiment on the other side. The full system of such equations is then solved to find the values of the model parameters for each particular platform. Existing measurement methods predominantly consist of *point-to-point* communication experiments, which are used to obtain a system of *linear* equations. In this subsection, we overview some notable works in this area.

Hockney [29] presents a measurement method to find the α and β parameters of the Hockney model. The set of communication experiments consists of point-to-point round-trips. The sender sends a message of size m to the receiver, which immediately returns the message to the sender upon its receipt. The time $RTT(m)$ of this experiment is measured on the sender side and estimated as $RTT(m) = 2 \cdot (\alpha + m \cdot \beta)$. These round-trip communication experiments for a wide range of message sizes m produce a system of linear equations with α and β as unknowns. To find α and β from this system, the linear least-squares regression is used.

Culler et al. [109] propose a method of measurement of parameters of the LogP model, namely, L, the upper bound on the latency, o_s, the overhead of processor involving sending a message, o_r, the overhead of processor involving receiving a message, and g, the gap between consecutive message transmission. The measurement method relies on the Active Messages (AM) protocol [110] and consists of the following four communication experiments:

- In the first experiment, the sender issues a small number of messages, N_s, consecutively without receiving any reply. The time of this experiment is measured on the sender side and estimated as $T_s = N_s \cdot o_s$. Thus, from this equation o_s can be found as $o_s = T_s/N_s$.

- In the second experiment, the sender issues a large number of messages, N_l ($N_l >> N_s$), consecutively. Time to send a message increases due to arriving replies during sending a message. When the capacity limit of the network is reached, the send request will eventually stall. Thus, the time to send N_l messages in one direction can be estimated as $T_l = N_l \cdot g$, and g is found from this linear equation as $g = T_l/N_l$. The time of this experiment is again measured on the sender side.

- The third experiment is designed to find o_r. The sender issues N_l messages in one direction with $\triangle$ amount of time between messages. The delay $\triangle$ is introduced in order to make sure that the reply from the receiver has reached the sender side and therefore the time to process the reply by the sender can be accurately estimated as o_r. The time of this experiment is measured on the sender side and estimated as $T^{'} = N_l \cdot (o_s + \triangle + o_r)$. Since $\triangle$ and o_s are known, o_r can be found from this linear equation as $o_r = T^{'}/N_l - o_s - \triangle$.

- The fourth experiment performs a round-trip of a single message. The time of this experiment is measured on the sender side and estimated as $RTT = 2 \cdot (o_s + L + o_r)$. From this linear equation, L can be found as $L = RTT/2 - o_s - o_r$.

Kielmann et al. [32] propose a method of measurement of parameters of the PLogP (Parameterized LogP) model. PLogP defines its model parameters, except for latency L, as functions of message size. The method consists of the following four communication experiments:

- The first experiment is designed to measure $g(0)$. The sender sends N consecutive zero-byte messages followed by a single empty reply from the receiver. Network saturation is achieved by increasing the number of messages, N. It is assumed that when the network is saturated, the time T to send a large number of zero-byte messages can be estimated as $T = N \cdot g(0)$, and g can be found by solving this linear equation as $g(0) = T/N$. The time of this experiment is measured on the sender side.

- The second experiment is designed to measure $o_s(m)$. The sender starts the clock, sends a single message of size m, and then stops the clock. The time of this experiment is estimated as $T_s(m) = o_s(m)$.

- The third experiment is designed to measure $o_r(m)$. The sender sends a zero-byte message to the receiver, waits for $\triangle$ time ($\triangle > T_s(m)$), starts the clock, receives a message of size m and then stops the clock. The receiver receives the zero-byte message from the sender and sends back a message of size m. The time $T_r(m)$ measured on the sender side is estimated as $T_r(m) = o_r(m)$.

- The fourth experiment is designed to measure L and $g(m)$. It consists of two round-trips, with a zero-byte message and a message of size m respectively. The time of the first round-trip is estimated as $RTT(0) = 2(L + g(0))$, and the time of the second round-trip is estimated as $RTT(m) = 2 \cdot L + g(0) + g(m)$. Both times are measured on the sender side. L and $g(m)$ are then found from this system of two linear equations as $L = RTT(0)/2 - g(0)$ and $g(m) = RTT(m) - RTT(0) + g(0)$.

Hoefler et al. [111] developed a method to measure parameters of the LogGP model. LogGP extends the LogP model by adding a G parameter, the

gap per byte for long messages. The building block of the method is a *ping-ping* round-trip, where the sender sends N consecutive messages of size m with delay d to the receiver, the receiver first receives all these messages and then sends them back to the sender, which also receives them all. The execution time of each communication experiment of the method, $PRTT(N, d, m)$, depends on parameters N, d and m of the experiment and measured on the sender side (PRTT stands for Parameterized Round-Trip Time). Three particular ping-ping round-trip experiments are used to obtain equations involving the LogGP model parameters as unknowns:

- The first experiment executes a round-trip of a single message $(N = 1)$ of size m without delay $(d = 0)$. The time of this experiment, $PRTT(1, 0, m)$, is estimated as $PRTT(1, 0, m) = 2 \cdot (o_s + L + o_r + (m - 1) \cdot G)$.

- The second experiment executes a ping-ping round-trip that issues N consecutive messages of size m without delay $(d = 0)$. The time of this experiment, $PRTT(N, 0, m)$, is estimated as $PRTT(N, 0, m) = PRTT(1, 0, m) + (N - 1) \cdot G_{all}$, where G_{all} is a cumulative hardware gap, estimated as $G_{all} = G \cdot (m - 1) + g$.

- The third experiment executes a ping-ping round-trip that issues N consecutive messages of size m with delay $d > 0$. The time of this experiment, $PRTT(N, d, m)$, is estimated as $PRTT(N, d, m) = PRTT(1, 0, m) + (N - 1) \cdot \max\{o_s + d, G_{all}\}$.

Now model parameters g, G, L, o_r and o_s are found as follows:

- From equations obtained from the first and second experiments, the linear equation $G \cdot (m - 1) + g = \frac{PRTT(N,0,m) - PRTT(1,0,m)}{N-1}$, involving two unknown parameters g and G, can be derived. By repeating these experiments for a wide range of message size m, a system of m linear equations with g and G as unknowns is produced. To find g and G from this system, the linear least-squares regression can be used.

- From the first experiment with $m = 1$, the equation $PRTT(1,0,1) = 2 \cdot (o_s + L + o_r)$ can be derived, giving $L = PRTT(1,0,1)/2 - (o_s + o_r)$. However, the authors argue that due to the overlap of processor overheads and network latency, L should be more accurately estimated as $L = PRTT(1,0,1)/2$.

- In order to measure o_r, the measurement method proposed by Kielmann [32] is used.

- Finally, o_s is found from the linear equation $o_s + d_G = \frac{PRTT(N,d_G,m) - PRTT(1,0,m)}{N-1}$, which is derived from the first and third experiments, as $o_s = \frac{PRTT(N,d_G,m) - PRTT(1,0,m)}{N-1} - d_G$. Here, parameter $d = d_G$ of the third experiment is determined empirically to guarantee that $d_G > G_{all}$.

Rico-Gallego et al. [89] propose a detailed method for measurement of parameters of the τ-Lop model on a multi-core cluster. τ-Lop assumes that the cost of transmission of a message of size m is estimated as $T^c_{p2p}(m) = o^c(m) + \sum_{j=0}^{s-1} L^c_j(m, \tau_j)$, where $o^c(m)$ is the overhead of protocols and software stack, $L^c_j(m, \tau_j)$ is the time to transfer a message of size m through channel c at the j-th step of the transmission, with τ_j contending transfers ($L^c_j(0, \tau_j) = 0$), and s is the number of steps of the message transmission. For each communication channel, *shared memory* or *network*, experimental measurement of $o^c(m)$ is designed separately using the following round-trip experiments:

- The first experiment executes a round-trip of a message of size m under the *Eager* protocol for shared memory and network. The time of the experiment is estimated as $RTT^c(0) = 2 \cdot (o^c(m) + \sum_{j=0}^{s-1} L^c_j(0,1))$. For each channel, $o^c(m)$ is found as $o^c(m) = RTT^c(0)/2$.

- The second experiment executes a round-trip of a message of size m under the *Rendezvous* protocol for shared memory and network. The time of the experiment is estimated as $Ping^c(0) = o^c(m) + \sum_{j=0}^{s-1} L^c_j(0,1)$. Therefore, $o^c(m) = Ping^c(0)$.

- The third set of experiments exchange messages of size m between processes using *MPI_Sendrecv* routine in a ring shape. Process P_i sends a message to P_{i+1} and receives a message from P_{i-1}. Then, *MPI_Wait* is called to complete both transmissions. L^0 and L^1 are estimated by the execution of these experiments in different channels respectively.

From this overview, we can conclude that the state-of-the-art methods for measurement of parameters of communication performance models are all based on *point-to-point communication experiments*, which are used to derive a system of equations involving model parameters as unknowns. In this work, we propose to use *collective communication experiments* in the measurement method in order to improve the predictive accuracy of analytical models of collective algorithms.

The only exception from this rule is a method for measurement of parameters of the LMO heterogeneous communication model [112, 113, 114]. LMO is a communication model of heterogeneous clusters, and the total number of its parameters is significantly larger than the maximum number of independent point-to-point communication experiments that can be designed to derive a system of independent linear equations with the model parameters as unknowns. To address this problem and obtain the sufficient number of independent linear equations involving model parameters, the method additionally introduces simple collective communication experiments, each using three processors and consisting of a one-to-two communication operation (scatter) followed by a two-to-one communication operation (gather). The experiments are implemented using the MPIBlib library [77]. This method however is not designed to improve the accuracy of predictive analytical models of communication algorithms.

3.4 Selection of collective algorithms using machine learning algorithms

Machine learning (ML) techniques have been also tried to solve the problem of selection of optimal MPI algorithms.

In [36], applicability of the quadtree encoding method to this problem is studied. The goal of this work is to select the best performing algorithm and segment size for a particular collective on a particular platform. The approach is based on the following steps. (1) Collective algorithms are executed on a particular platform to collect detailed performance data. (2) The decision map is built for the collective on a particular platform by analyzing the performance data. It is assumed that the decision map covers all message and communicator sizes. (3) The quadtree is initialized using the decision map. (4) The decision function source code is generated from the initialised quadtree. For example, Linear tree, Binary tree, Binomial tree, Split-binary tree, and Chain tree broadcast algorithms are profiled with maximum 50 processes. The experimental results show that mean performance penalty reaches 74% and 37% and maximum performance penalty reaches 391% and 743% on different platforms respectively. While the study shows some level of applicability of the quadtree encoding algorithm to the problem, collection of detailed profiling data of collectives for all message sizes and communicator sizes is a very expensive procedure. Besides, for some message sizes and communicator sizes the penalty of the decision function is too high. Taking into account that decision trees are considered weak learners [115], the decision function will perform poorly on unseen data.

Applicability of the C4.5 algorithm to the MPI collective selection problem is explored in [116]. The C4.5 algorithm [117] is a decision tree classifier, which is employed to generate a decision function, based on a detailed profiling data of MPI collectives. The same steps are followed to build the decision tree using the C4.5 algorithm as in the quadtree encoding method presented above. The same weaknesses are shared by the decision trees built by the quadtree encoding algorithm and by the C4.5 algorithm. While the

accuracy of the decision function built by the C4.5 classification algorithm is higher than that of the decision function built by quadtree encoding algorithm, still, the performance penalty is higher than 50%.

Most recently Hunold et al. [37] studied the applicability of six different ML algorithms for selection of optimal MPI collective algorithms. The basic idea of their approach is to create a regression model for every collective algorithm that is available for a given collective operation, predicting the execution time of the collective algorithm. The constructed regression models are then used at run time to select the algorithm that minimizes the execution time for unseen configurations. The ML algorithms employed to build the regression models are Random Forests [118], Neural Networks [35], Linear Regressions [119], XGBoost [120], K-nearest Neighbor [121], and Generalized Additive Models (GAM) [122]. The configuration is characterised by the message size, the number of nodes, and the number of processes per node. The approach is evaluated using MPI_Bcast, MPI_Allreduce and MPI_Alltoall collectives. In the experimental evaluation, the number of nodes varies between 4 and 36, and the number of processes per node varies between 1 and 32. The experimental results show two things. First, it is very expensive and difficult to build a regression model even for a relatively small cluster. There is no clear guidance how to do it to achieve better results. Second, even the best regression models do not accurately predict the fastest collective algorithm in most of the reported cases. Moreover, in many cases the selected algorithm performs worse than the default algorithm, that is, the one selected by a simple native decision function.

To the best of the authors' knowledge, the works outlined in this subsection are the only research done in MPI collective algorithm selection using ML algorithms. The results show that the selection of the optimal algorithm without any information about the semantics of the algorithm yields inaccurate results. While the ML-based methods treat a collective algorithm as a black box, we derive its performance model from the implementation code and estimate the model parameters using statistical techniques. The limitations of the application of the statistical techniques (AI/ML) to collective performance modelling and selection problem can be found in a detailed

survey [123].

3.5 Summary

As presented in Section 3.3, most of the analytical performance models are used either to design the algorithms or for theoretical analysis of the algorithms. Applicability of the analytical models to the selection problem is studied in [28]. The analytical performance models in this work are built using the traditional high-level description of the algorithms. Unfortunately, the experimental validations show that this approach is failed to compare the performance of the algorithms accurately. Moreover, the model parameters are measured using simple point-to-point communication tests where the measured values are constant for all algorithms. Thus, in this thesis we propose two innovations that significantly improve the selective accuracy of analytical models: (1) We derive analytical models from the code implementing the algorithms rather than from their high-level mathematical definitions. (2) We estimate model parameters separately for each collective algorithm and include the execution of this algorithm in the corresponding communication experiment.

Chapter 4

Modelling of Collective Communication Algorithms

In this chapter, we build new analytical performance models for broadcast and gather algorithms. Section 4.1.1 and 4.1.2 provide descriptions of broadcast and gather algorithms used in Open MPI respectively. We build new analytical performance models for two different configurations: (1) MPI processes are mapped by CPU (One-Process-Per-CPU). (2) MPI processes are mapped by CPU-core (One-Process-Per-Core).

4.1 Collective Communication Algorithms

4.1.1 MPI Broadcast Algorithms

Open MPI architecture is based on software components, plugged into the library kernel. A component provides with a functionality with specific implementation features. For instance, collective component known as *Tuned* implements different algorithms for each collective operation defined in MPI as a sequence of point-to-point transmissions between the involved processes. A *communicator* provides with an isolated communication context for the group of processes executing the collective operation. Processes in a communicator are identified by an assigned *rank* integer number, starting at

0.

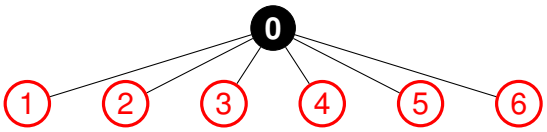
In the broadcast operation (MPI_Bcast) a process called *root* sends a message with the same data to all processes in the communicator. Messages can be segmented in transmissions. *Segmentation* of messages is a common technique used for enabling higher bandwidth utilization, and hence, improving the performance. It consists on dividing up the message into smaller fragments called segments and sending them in sequence. MPI libraries use two protocols to implement point-to-point communication: *eager* and *rendezvous*. The send primitive switches from eager to rendezvous when the message size reaches a threshold size E. Use of message segmentation where segment size smaller than eager threshold (E) avoids the *rendezvous* protocol.

Every algorithm implementing the broadcast in the *Tuned* component defines a communication graph with a specific topology between the P ranks in the communicator. Ranks are the nodes in the graph, and they are mapped to the processes of the parallel machine. The features and topology of the broadcast algorithms implemented in Open MPI *Tuned* component are listed below:

- **Linear (Flat) tree algorithm**. The algorithm employs a single level tree topology shown in Figure 4.1a where the root node has $P - 1$ children. The message is transmitted to child nodes without segmentation. In this thesis linear and flat tree broadcast algorithms are used interchangeably.

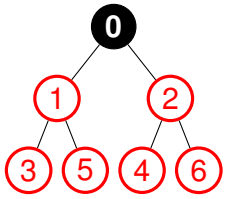
- **Chain tree algorithm**. Each internal node in the topology has one child (see Fig 4.1b). The message is split into segments and transmission of segments continues in a pipeline until last node gets the broadcast message. ith process receives the message from $(i-1)$th process, and sends to $(i+1)$th process.

- **Binary tree algorithm**. Unlike the chain tree, each internal process has two children, and hence data is transmitted from each node to both children (Figure 4.1c). Segmentation technique is employed in this algorithm. For simplicity we assume that binary tree is complete, then $P = 2^H - 1$ where H is the height of the tree, $H = \log_2(P+1)$.

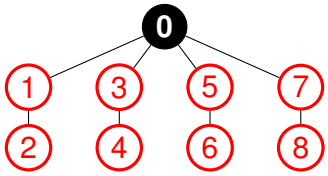


(a) Flat tree

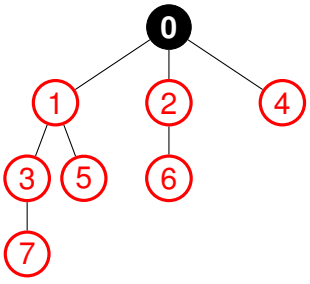
(b) Chain tree



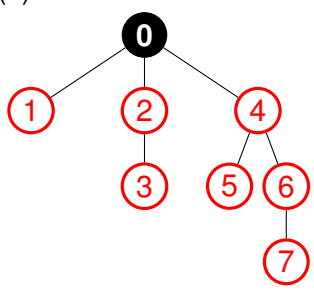
(c) Binary tree



(d) K-Chain tree (K = 4)



(e) Balanced binomial tree



(f) In-order binomial tree

Figure 4.1: *Virtual topologies for collective algorithms*

- **Split binary tree algorithm**. The split binary tree algorithm employs the same virtual topology as the binary tree (Figure 4.1c). As the name implies, the difference from the binary tree algorithm is that the message is split into two halves before transmission. After splitting the message, the right and left halves of the message are pushed down into the right and left sub-trees respectively. In an additional last phase, the left and right nodes exchange in pairs their halves of the message to complete the broadcast operation.

- **K-Chain tree algorithm**. The *K-Chain* virtual topology is employed in the algorithm (Figure 4.1d). The root broadcasts the message using segmentation to the child processes, and then the child processes broadcast the message to their children in parallel. As the name implies, the virtual topology consists of K number of *chain* tree virtual topology each of which is connected to *root*. The height of *K-chain tree* is estimated as $H = \lfloor \frac{P-1}{K} \rfloor$. Last process must wait for $H_{k-chain}$ steps until it gets the broadcast message. Rank of processes are mapped into K-Chain tree virtual topology using following formula, $\sum_{k=0}^{K-1} \sum_{i=0}^{H-1} (H \cdot k + i + 1)$

- **Binomial tree algorithm**.

  Definition 4.1.1 (Balanced binomial tree)**.** The balanced binomial tree of degree k with root R is the tree B_k defined as follows,

  1. If $k = 0$, $B_k = B_0 = \{R\}$, i.e., the binomial tree of degree zero consists of a single node, R.
  2. If $k > 0$, $B_k = \{B_{k-1}, ..., B_0, R\}$ i.e., the binomial tree of degree k comprises the root R, and k binomial subtrees, $B_{k-1}, ..., B_0$

  Figure 4.2 illustrates how the balanced binomial tree is built. The binomial tree broadcast algorithm employs balanced binomial tree. Unlike the binary tree, the maximum nodal degree of the binomial tree decreases from the root down to the leaves as follows:

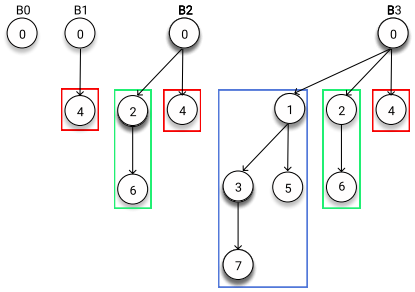


Figure 4.2: The balanced binomial tree of order 3. We highlighted subtrees of all lower ordered binomial trees. The order 3 binomial tree is connected to an order 2, 1, and 0 (highlighted as blue, green and red respectively) binomial tree.

$\lceil \log_2 P \rceil, \lceil \log_2 P \rceil - 1, \lceil \log_2 P \rceil - 2, ...$ The height of the binomial tree is the order of the tree, $H = \lfloor \log_2 P \rfloor$.

4.1.2 MPI Gather Algorithms

MPI_Gather is a *many-to-one* MPI operation. MPI_Gather takes data elements from all processes of the communicator and gathers them in one single process which is called *root*. MPI_Gather is used in many parallel applications such as parallel sorting and searching. The complete list of gather algorithms employed in Open MPI is as follows:

- **Linear algorithm without synchronisation**. This algorithm employs flat tree virtual topology (Figure 4.1a). In this algorithm, the non-root processes send their messages to the root, which posts receives from everyone. If the operation is not denoted as “in-place”, the root must perform a local copy of its own data.

- **Linear algorithm with synchronisation**. The algorithm uses flat tree virtual topology (Figure 4.1a) as well. This algorithm was introduced to prevent overloading of the root process using message segmentation technique. The message is split into two segments on each non-root

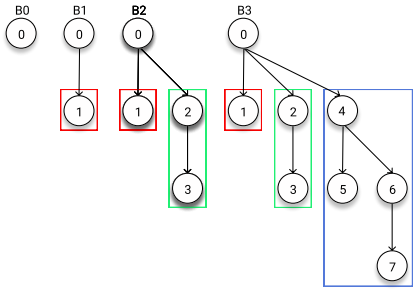


Figure 4.3: The in-order binomial tree of order 3. We highlighted subtrees of all lower ordered binomial trees. The order 3 binomial tree is connected to an order 0, 1, and 2 (highlighted as blue, green and red respectively) binomial tree.

process. To receive the message from non-root processes, the algorithm performs the following steps: (1) The root receives the first incoming segment of the message; (2) Then, the root sends a zero-byte message to non-root processes, signalling them to send the second segment of the message; (3) The root receives the second segment of the message.

- **Binomial algorithm**.

  Definition 4.1.2 (In-order binomial tree)**.** The in-order binomial tree of degree k with root R is the tree B_k defined as follows,

  1. If $k = 0$, $B_k = B_0 = \{R\}$, i.e., the binomial tree of degree zero consists of a single node, R.
  2. If $k > 0$, $B_k = \{R, B_0, B_1, ..., B_{k-1}\}$ i.e., the binomial tree of degree k comprises the root R, and k binomial subtrees, $B_0, B_1, ..., B_{k-1}$

  Figure 4.3 illustrates how the in-order binomial tree is built. The binomial algorithm employs in-order binomial tree topology (Figure 4.1f). The leaf nodes send their data to their parent processes immediately. Internal nodes in the tree wait to receive the data from all children before forwarding the message up the tree. Once the root node receives all messages, a local data shift operation may be necessary to put the data in the correct place.

Algorithm 1 Tree-based segmented broadcast algorithm

```
if (rank == root) then
  // Send segments to all children
  for i ∈ 0..n_s − 1 do
    for child ∈ list of children do
      MPI_Isend(segment[i], child, ... )
    end for
    MPI_Waitall(. . .)
  end for
else if (intermediate nodes) then
  for i ∈ 0..n_s − 1 do
    // Post receive and wait
    MPI_Irecv(segment[i])
    MPI_Wait(. . .)
    // Send data to children
    for child ∈ list of children do
      MPI_Isend(segment[i], child, ... )
    end for
    MPI_Waitall(children)
  end for
else if (leaf nodes) then
  // Receive all segments from parent in a loop
  for i ∈ 0..n_s − 1 do
    MPI_Irecv(segment[i], ... )
    MPI_Wait(. . .)
  end for
end if
```

Section 4.2 and 4.3 introduce performance models of the algorithms described above for *one-process-per-cpu* and *one-process-per-core* configurations respectively.

4.2 Modelling of Collective Communication Algorithms: One-Process-Per-CPU

As stated in Chapter 1, we propose a new approach to analytical performance modelling of collective algorithms. While the traditional

approach only takes into account high-level mathematical definitions of the algorithms, we derive our models from their implementation. This way, our models take into account important details of their execution having a significant impact on their performance. In this section, we present our analytical modelling approach by applying it to broadcast and gather collective algorithms implemented in Open MPI. This approach could be similarly applied to other collective algorithms and MPI implementations such as MPICH. Analytical models of the broadcast and gather collective algorithms implemented in Open MPI are derived in Sections 4.2.1 and 4.2.2.

To model point-to-point communications, we use the Hockney model, which estimates the time $T_{p2p}(m)$ of sending a message of size m between two processes as $T_{p2p}(m) = \alpha + \beta \cdot m$, where α and β are the latency and the reciprocal bandwidth respectively. For segmented collective algorithms, we assume that $m = n_s \cdot m_s$, where n_s and m_s are the number of segments and the segment size respectively. We assume that each algorithm involves P processes ranked from 0 to $P - 1$.

4.2.1 Broadcast Algorithms

In this section, we build analytical performance models of broadcast algorithms implemented in Open MPI. All broadcast algorithms implemented in Open MPI, except for the linear tree broadcast algorithm, are implemented using message segmentation. As the main purpose of message segmentation is to avoid the *rendezvous* protocol, we only build analytical models of broadcast algorithms with message segmentation assuming the buffered mode of *send* operations. Models of segmented broadcast algorithms employing the *rendezvous* (synchronous) mode would have no practical application in Open MPI as they assume a configuration with a segment size being not small enough to avoid the *rendezvous* protocol, which does not make much sense.

Algorithm 2 Linear (Flat) tree broadcast algorithm

```
if (rank == root) then
  // Send whole message
  for child ∈ list of children do
    MPI_Send(message, child, ... )
  end for
else
  // Receive whole message from root
  MPI_Recv(message, ... )
end if
```

4.2.1.1 Linear (Flat) Tree Algorithm

In Open MPI, the linear broadcast algorithm is implemented using blocking *send* and *receive* operations (see Algorithm 2). The algorithm transmits the whole message from root to the leaves without message segmentation. Regardless of communication mode (buffered or not), because of blocking communication, each next *send* only starts after the previous one has been completed. Therefore, the execution time of the linear tree broadcast algorithm will be equal to the sum of execution times of $P - 1$ send operations:

$$T^{blocking}_{linear\_bcast}(P, m) = (P - 1) \cdot (\alpha + m \cdot \beta). \tag{4.1}$$

In Open MPI, this linear tree algorithm is one of the six algorithms available for implementation of the *MPI_Bcast* routine. There is another linear tree broadcast algorithm, which cannot be chosen to implement *MPI_Bcast*, but only used as a building block in other tree-based broadcast algorithms implementing *MPI_Bcast*, namely, in the *binomial tree*, *binary tree*, *k-chain tree*, and *chain tree* broadcast algorithms (see Algorithm 1 for more details). That linear tree algorithm is implemented using *non-blocking* send and receive operations.

In this latter case, $P - 1$ non-blocking *send*s will run on the *root* concurrently. Therefore, the execution time of the linear broadcast algorithm using non-blocking point-to-point communications and buffered mode,

$T^{nonblock}_{linear\_bcast}(P,m)$, can be bounded as follows:

$$T_{p2p}(m) \leq T^{nonblock}_{linear\_bcast}(P,m) \leq (P-1) \cdot T_{p2p}(m). \tag{4.2}$$

We will approximate $T^{nonblock}_{linear\_bcast}(P,m)$ as

$$T^{nonblock}_{linear\_bcast}(P,m) = \gamma(P,m) \cdot (\alpha + m \cdot \beta), \tag{4.3}$$

where

$$\gamma(P,m) = \frac{T^{nonblock}_{linear\_bcast}(P,m)}{T_{p2p}(m)}. \tag{4.4}$$

We will use this approximation when deriving analytical performance models of the remaining five broadcast algorithms implemented in Open MPI. As we can see from Algorithm 1, the non-blocking version of linear tree broadcast is used in these five algorithms for transmission of a single message segment. In this thesis, we assume the same fixed segment size in all segmented algorithms. Therefore, in the rest of the thesis we define γ as a function of P only, $\gamma(P)$. From Formula 4.2, we can derive that $T^{nonblock}_{linear\_bcast}(2,m) = T_{p2p}(m)$ and, hence, $\gamma(2) = 1$.

4.2.1.2 Binomial Tree Algorithm

In Open MPI, the binomial tree broadcast algorithm is segmentation-based and implemented as a combination of linear tree broadcast algorithms using non-blocking *send* and *receive* operations.

Figure 4.7 shows the stages of execution of the binomial tree broadcast algorithm. Each stage consists of parallel execution of a number of linear broadcast algorithms using non-blocking communication.The linear broadcast algorithms running in parallel have a different number of children. Therefore, the execution time of each stage will be equal to the execution time of the linear broadcast algorithm with the maximum number of children. The execution time of the whole binomial broadcast algorithm will be equal to the sum of the execution times of these stages.

In Open MPI, the binomial tree broadcast algorithm employs the balanced

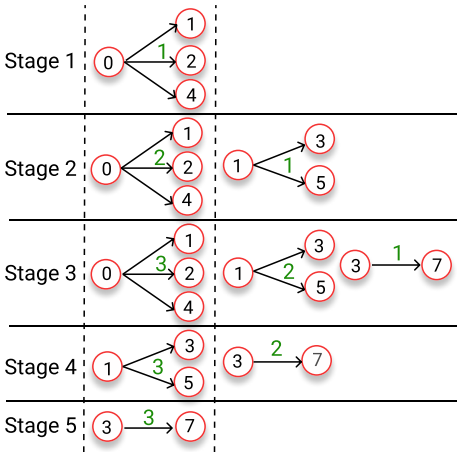


Figure 4.4: Execution stages of the binomial tree broadcast algorithm, employing the non-blocking linear broadcast ($P = 8$, $n_s = 3$). Nodes are labelled by the process ranks. Each arrow represents transmission of a segment. The number over the arrow gives the index of the broadcast segment.

binomial tree virtual topology. Therefore, the number of stages in the binomial broadcast algorithm can be calculated as

$$N_{steps} = \lfloor log_2 P \rfloor + n_s - 1. \tag{4.5}$$

Thus, the time to complete the binomial tree broadcast algorithm can be estimated as follows:

$$T_{binomial\_bcast}(P, m, n_s) = \sum_{i=1}^{\lfloor log_2 P \rfloor + n_s - 1} \max_{1 \le j \le \min(\lfloor log_2 P \rfloor, n_s)} T^{nonblock}_{linear\_bcast}(P_{ij}, \frac{m}{n_s}), \tag{4.6}$$

where P_{ij} denotes the number of nodes in the j-th linear tree of the i-th stage.

Using the property of the binomial tree and Formula 4.23, we have

$$T_{binomial\_bcast}(P, m, n_s) = (n_s \cdot \gamma(\lceil \log_2 P \rceil + 1) + \sum_{i=1}^{\lfloor \log_2 P \rfloor - 1} \gamma(\lceil \log_2 P \rceil - i + 1) - 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta). \tag{4.7}$$

4.2.1.3 Chain Tree Algorithm

In Open MPI, the chain tree algorithm is segmentation-based and implemented using non-blocking point-to-point communication. While the height of the chain tree equal to $P - 1$, the algorithm will be completed in $P + n_s - 2$ steps, each consisting of a varying number of concurrent non-blocking point-to-point communications (technically, Open MPI employs concurrent non-blocking linear tree broadcast algorithms, but in this case each linear broadcast will be equivalent to a point-to-point communication). Therefore, the execution time of the chain tree algorithm can be estimated as

$$T_{chain\_bcast}(P, m, n_s) = (P + n_s - 2) \cdot (\alpha + \frac{m}{n_s} \cdot \beta). \tag{4.8}$$

4.2.1.4 Split-Binary Tree Algorithm

In Open MPI, the split-binary tree algorithm is segmentation-based and implemented using blocking point-to-point communication. The algorithm consists of two phases – *forwarding* and *exchange*. In the first phase, the message of size m is split into two equal parts in the root, which are then sent to the left and right subtrees respectively using message segmentation. After completion of the first phase, each node in the left subtree contains the first half of the message and each node in the right subtree – the second half of the message. Because of segmentation, each node will receive $\frac{n_s}{2}$ segments during the first phase.

As the balanced binary tree virtual topology is employed in the split-binary tree algorithm, each node in the left subtree will have a matching pair in the right subtree and vice versa. In the second phase, each pair of matching nodes in the left and right subtrees exchange their halves of the message. The execution time of the split-binary tree broadcast will be equal to the sum of the execution times of the first and the second phases. As the heigh of the balanced binary tree is equal to $\lfloor \log_2 P \rfloor$, we have

$$T_{split\_binary\_bcast}(P, m, n_s) = 2 \cdot (\lfloor \log_2 P \rfloor + \frac{n_s}{2} - 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta) + (\alpha + \frac{m}{2} \cdot \beta) \quad (4.9)$$

4.2.1.5 Binary Tree Algorithm

In Open MPI, the binary tree broadcast algorithm is segmentation-based and uses the balanced binary tree topology (see Figure c). The root broadcasts each segment to its children using the non-blocking linear tree broadcast algorithm. Upon receipt of next segment, each internal node acts similarly. As the binary tree used in this algorithm is balanced, all the non-blocking linear broadcasts will have the same execution time, namely,

$$T^{nonblock}_{linear\_bcast}(3, m_s) = \gamma(3) \cdot (\alpha + \tfrac{m}{m_s} \cdot \beta).$$

As the height of the balanced binary tree is equal to $\lfloor \log_2 P \rfloor$, the algorithm will be completed in $(\lfloor \log_2 P \rfloor + n_s - 1)$ steps, each consisting of a varying number of concurrent non-blocking linear broadcasts, involving 3 processes. Therefore,

$$T_{binary\_bcast}(P, m, n_s) = \gamma(3) \cdot (\lfloor \log_2 P \rfloor + n_s - 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta). \quad (4.10)$$

4.2.1.6 K-chain Tree Algorithm

In Open MPI, the K-chain tree algorithm is implemented using non-blocking communication and message segmentation. In the K-chain tree, the root node has $K(K = 4)$ children, while the internal nodes have a single child each (Figure d). As the height of the tree is $\lfloor \frac{P-1}{K} \rfloor$, the algorithm takes $\lfloor \frac{P-1}{K} \rfloor + n_s - 1$ steps to complete. At each step, a varying number of non-blocking linear tree broadcast algorithms will be executed concurrently (one at the first step, K at the last step, and up to $K \times (\lfloor \frac{P-1}{K} \rfloor - 1) + 1$ algorithms for intermediate steps). Note, that while Open MPI employs concurrent non-blocking linear

tree broadcast algorithms, in this case the most of the linear broadcasts will be equivalent to non-blocking point-to-point communications.

The execution time of the K-chain tree algorithm will be equal to the sum of the execution times of its steps. The execution time of each step will be equal to the maximum execution time of the concurrently executed linear broadcasts. For the first n_s steps, this maximum time will be the time of the linear broadcast involving the root of the whole K-chain tree, which is estimated as $\gamma(K+1)\cdot(\alpha+\frac{m}{n_s}\cdot\beta)$ according to Formula 4.23. For each of the remaining $\lfloor\frac{P-1}{K}\rfloor-1$ steps, all concurrently executed linear broadcasts will be equivalent to non-blocking point-to-point communications, the time of which is $\alpha+\frac{m}{n_s}\cdot\beta$. Thus, the total execution time of the K-chain tree algorithm will be estimated as

$$T_{k\_chain\_bcast}(P,m,n_s) = (\lfloor\frac{P-1}{K}\rfloor+ \gamma(K+1)\cdot n_s - 1)\cdot(\alpha+\frac{m}{n_s}\cdot\beta). \quad (4.11)$$

4.2.2 Gather Algorithms

In this section, we derive analytical formulas of the gather algorithms implemented in Open MPI.

4.2.2.1 Linear Without Synchronisation

In the Open MPI implementation of the *linear without synchronisation* gather algorithm, the root receives messages from its $P-1$ children using blocking receive operations. Therefore, the execution time of this gather algorithm can be estimated as the sum of the execution times of $P-1$ blocking receive operations, that is,

$$T_{linear\_gather}(P,m) = (P-1)\cdot(\alpha+m\cdot\beta). \quad (4.12)$$

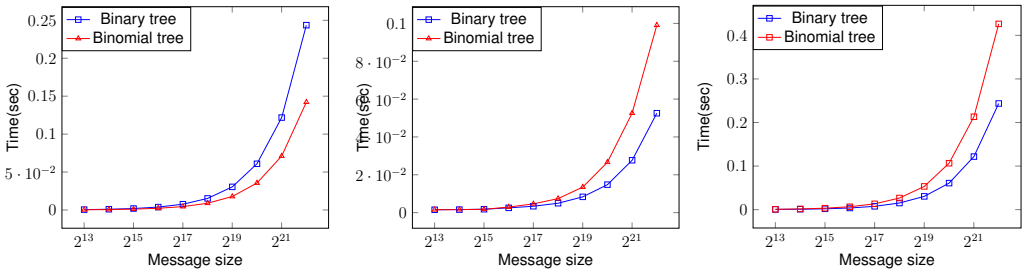
4.2.2.2 Linear With Synchronisation

The Open MPI implementation of the *linear with synchronisation* gather algorithm employs both blocking and non-blocking communications. The messages gathered from the children are all identically split into two equal parts. In order to receive all these parts from its $P-1$ children, the root executes a loop, at i-th iteration of which it receives both halves of the message from the i-th child by performing the following steps: 1) it first posts a non-blocking receive for the first part; 2) then it sends a zero-byte message using a blocking send, signalling the child to start sending the message parts; 3) then the root posts a non-blocking receive for the second half of the message; 4) finally, it blocks itself waiting for the completion of the previously posted non-blocking receives.

At the same time, upon receipt of a zero-byte signal message from the root, each child will perform two successive standard blocking sends for the first and the second parts of its message. When the size of these parts, $\frac{m}{2}$, is grater than the eager threshold, E, then the standard blocking sends will follow the *rendesvouz* protocol, that is, will be equivalent to *synchronous* sends. Otherwise, they will follow the *eager* protocol, that is, will be equivalent to *buffered* sends. In the first case, the execution of all point-to-point communications will be serialized, and, therefore, the execution time of the linear gather with synchronisation algorithm can be estimated as the sum of the execution times of the employed point-to-point communications:

$$\begin{aligned} T_{linear\_gather\_with\_synch}(P,m) &= (P-1)\cdot(2\cdot(\alpha+\frac{m}{2}\cdot\beta)) \\ &= (P-1)\cdot(2\cdot\alpha+m\cdot\beta). \end{aligned} \tag{4.13}$$

Otherwise, when $\frac{m}{2} \leq E$, each child will send its half-messages concurrently. Therefore, the execution time of the linear gather with synchronisation algorithm in this case can be estimated as



(a) (b) (c)

Figure 4.5: Performance estimation of the binary and binomial tree broadcast algorithms by the traditional and proposed analytical models in comparison with experimental curves. The experiments involve ninety processes (P=90). (a) Estimation by the existing analytical models. (b) Experimental performance curves. (c) Estimation by the proposed analytical models derived from the implementation codes.

$$T_{linear\_gather\_with\_synch}(P,m) = (P-1)\cdot(\alpha + \frac{m}{2}\cdot\beta). \tag{4.14}$$

4.2.2.3 Binomial Algorithm

In Open MPI, the binomial gather algorithm employs the *in-order* binomial tree virtual topology (Figure 4.1f). The leaf nodes and internal nodes use the standard blocking send to send the messages to their parents, which receive the message using the blocking receive. The algorithm will be completed in $\lfloor \log_2 P \rfloor$ steps, each performing a set of concurrent blocking receives.

At i-th step, the root will receive a message of size $2^{i-1}\cdot m$ from its i-th child, combining the messages gathered by the latter acting as the root of the i-th subtree during the previous $i-1$ steps ($i = 1, ..., \lfloor \log_2 P \rfloor$). Given this message size, $2^{i-1}\cdot m$, will be the largest communicated at the i-th step of the algorithm, its execution time can be estimated as

$$\begin{aligned} T_{binomial\_gahter}(P,m) &= \sum_{i=1}^{\lceil \log_2 P \rceil} (\alpha + 2^{i-1}\cdot m\cdot\beta) \\ &= \lceil \log_2 P \rceil \cdot \alpha + (P-1)\cdot m\cdot\beta. \end{aligned} \tag{4.15}$$

4.2.3 Comparison of The Proposed Analytical Models Against The State of The Art

In this section, we use the binomial and binary tree algorithms as an example to illustrate that unlike the traditional approaches, the approach based on the derivation of analytical models of collective algorithms from their implementation codes, yields models, which can be used for accurate pairwise comparison of the performance of collective algorithms implementing the same collective operation.

Existing analytical modelling approaches [91, 27, 100] estimate the execution time of the binary and binomial tree broadcast algorithms as follows:

$$T_{binomial\_bcast}(P,m) = \lceil \log_2 P \rceil \cdot (\alpha + m \cdot \beta),$$

$$T_{binary\_bcast}(P,m) = 2 \cdot (\lceil \log_2(P+1) \rceil - 1) \cdot (\alpha + m \cdot \beta).$$

Figure 4.5 shows the performance of the binary tree and binomial tree algorithms using: a) the estimation by the existing analytical models; b) the experimental results on the Grisou cluster of the Grid'5000 platform; c) the estimation by the analytical models presented in Section 4.2.1. It is evident that while the existing models wrongly predict that the binomial tree algorithm will outperform the binary tree algorithm on the target platform, our models correctly predict the relative performance of these algorithms.

4.3 Modelling of Collective Communication Algorithms: One-Process-Per-Core

We build new analytical performance models of broadcast algorithms described in Section 4.1.1 for multi-core clusters. For point-to-point communication modelling, we use the τ-Lop model.

τ-Lop takes into account the distinctive features of the communication channels to represent a point-to-point transmission as a sequence of

transfers via shared memory or network. In this approach, we use the τ-Lop model, that it estimates the time of sending a point-to-point message in s transfers as $T^c_{p2p}(m) = o^c(m) + \sum_{i=1}^{s} L^{c_i}(m)$, where c represents the communication channel, and the o and L parameters represent the overhead of the communication protocol and the transfer time respectively. Hence, τ-Lop considers a different representation for a message transmitted through shared memory ($c = 0$) and network channel ($c = 1$). For instance, through shared memory, MPI libraries default transmission is through a *shared intermediate buffer* between sender and receiver processes, hence two identical transfers ($s = 2$) are needed to transmit the message, and previous expression reduces to $T^0_{p2p}(m) = o^0(m) + 2L^0(m)$. While, through a network, representation depends on the network capabilities. For instance, in an Ethernet network we consider two shared memory transfers, from sender memory to NIC and from receiver NIC to destination memory, and a network transfer between NICs, as $T^1_{p2p}(m) = o^1(m) + 2L^0(m) + L^1(m)$.

Most of the Open MPI broadcast algorithms are implemented using message segmentation, except for the flat tree broadcast algorithm. For segmented broadcast algorithms, we assume that $m = n_s \cdot m_s$, where n_s and m_s are the number of segments and the segment size respectively. In this work, we assume the same fixed segment size in all segmented algorithms.

4.3.1 Flat Tree Algorithm

In Open MPI, the linear broadcast algorithm is implemented using blocking *send* and *receive* operations. The algorithm transmits the whole message from the root to the leaves without message segmentation. Regardless of communication mode (buffered or not), because of blocking communication and assuming the *rendezvous* protocol, each next *send* only starts after the previous one has been completed. Therefore, the execution time of the linear tree broadcast algorithm will be equal to the sum of execution times of $P - 1$ send operations:

$$T_{BFT}(P,m) = \sum_{i=1}^{P-1} T_{p2p}^{c_i}(m), \tag{4.16}$$

where $T_{p2p}^{c_i}$ is the point-to-point communication time through a channel c_i, connecting the root and the i-th process, estimated using the τ-Lop model. In the rest of the thesis, we use BFT to refer to the blocking flat tree broadcast algorithm.

Every time the Open MPI *MPI_Bcast* operation is invoked with a specific root, an internal tree with the specific virtual topology for the chosen algorithm is built, and then, the algorithm is executed. This internal tree is used as a building block in tree-based segmented broadcast algorithms implementing *MPI_Bcast*, namely, in the *binomial tree*, *binary tree*, *split binary tree*, *k-chain tree*, and *chain tree* broadcast algorithms (see Algorithm 1 for more details). That tree algorithm is composed of flat trees using *non-blocking* send and receive operations, hence we separately refer to any of them as *non-blocking flat tree* (NBFT).

As illustrated in Figure 4.6 and Figure 4.7, NBFT can use either one of the two available channels for all point-to-point communications or both of them. In general, the number of network point-to-point communications, C, in an NBFT can be calculated as follows,

$$C = \sum_{i=1}^{P-1} c_i \tag{4.17}$$

Obviously, $0 \leq C \leq P-1$. Then, the number of point-to-point communications through shared memory in the NBFT can be expressed as $P - C - 1$. The time of message transmission through network channel is longer than through shared memory. In our model, we assume that

$$T_{p2p}^{1}(m) = Q(m) \cdot T_{p2p}^{0}(m), \tag{4.18}$$

where $Q(m)$ is a platform-dependent parameter representing the ratio of delays of the communication channels ($Q(m) \geq 2$). We denote $T_{NBFT}^{c}(P,m)$ the execution time of an NBFT, which uses only one channel, c, for all

message transmissions. The execution time of an arbitrary NBFT, $T_{NBFT}(P, C, m)$, is modelled as follows,

$$T_{NBFT}(P, C, m) = \begin{cases} T^0_{NBFT}(P, m), \text{ if C=0} \\ T^1_{NBFT}(C + \lfloor \frac{P-C-1}{Q(m)} \rfloor + 1, m), \text{otherwise.} \end{cases} \tag{4.19}$$

Thus, in our model the execution time of any NBFT **A**, which uses two channels, will be calculated as the execution time of the NBFT **B**, only using the network channel and obtained from **A** by formally replacing each group of $Q(m)$ shared-memory transmissions by one network transmission.

It is evident from Algorithm 1 that NBFTs are only used in Open MPI to transmit segments of the same fixed size, m_s, which is therefore not a variable in our model.

The execution time of the NBFT broadcasting a message of size m_s through channel c, $T^c_{NBFT}(P, m_s)$, can be bounded as follows,

$$T^c_{p2p}(m_s) \leq T^c_{NBFT}(P, m_s) \leq T^c_{BFT}(P, m_s). \tag{4.20}$$

From formula (4.16), we can derive

$$T^c_{BFT}(P, m_s) = \sum_{i=1}^{P-1} T^c_{p2p}(m_s) = (P-1) \cdot T^c_{p2p}(m_s). \tag{4.21}$$

Hence,

$$T^c_{p2p}(m_s) \leq T^c_{NBFT}(P, m_s) \leq (P-1) \cdot T^c_{p2p}(m_s). \tag{4.22}$$

Therefore, we approximate $T^c_{NBFT}(P, m_s)$ as follows,

$$T^c_{NBFT}(P, m_s) = \gamma^c(P) \cdot T^c_{p2p}(m_s), \tag{4.23}$$

where $\gamma^c(P)$ is a *parallelisation factor*, representing the increase in the cost of the overlapping $P-1$ non-blocking transmissions of a segment of size m_s through the channel c in the NBFT with respect to a single point-to-point

transmission.

4.3.2 Binomial Tree Algorithm

In Open MPI, the binomial tree broadcast algorithm is segmentation-based and implemented as a combination of flat tree broadcast algorithms using non-blocking *send* and *receive* operations. Figure 4.6 shows the topology of a binomial tree with $P = 8$ and the NBFTs executing at different stages of the algorithm (see Algorithm 1). Figure 4.7 illustrates the stages of execution of the binomial tree algorithm illustrated in Figure 4.6. Each stage consists of parallel execution of a number of NBFTs. Therefore, the execution time of a stage will be equal to the maximum execution time of its NBFTs and the execution time of the algorithm will be equal to the sum of the execution times of the stages.

Open MPI employs the *balanced* binomial tree for the binomial tree broadcast algorithm. The height of the balanced binomial tree is equal to $\lfloor log_2 P \rfloor$. The number of parallel running NBFTs, j, at the i-th stage can be bound by $2 + (\lfloor log_2 P \rfloor - 1) \cdot (\lfloor log_2 P \rfloor - 2)$. The algorithm will be completed in $\lfloor log_2 P \rfloor + \frac{m}{m_s} - 1$ stages. Thus, the time to complete the binomial tree broadcast algorithm will be estimated as follows,

$$T_{binomial}(P, m, m_s) = \sum_{i=1}^{\lfloor log_2 P \rfloor + \frac{m}{m_s} - 1} \max_j T_{NBFT}\left(P_{ij}, C_{ij}, m_s\right), \tag{4.24}$$

where P_{ij} $(1 \leq P_{ij} \leq \lceil log_2 P \rceil)$ is the number of nodes in the j-th $NBFT$ running at i-th stage.

4.3.3 Chain Tree Algorithm

In Open MPI, the chain tree algorithm is segmentation-based and implemented using non-blocking point-to-point communication. While the height of the chain tree equal to $P - 1$, the algorithm will be completed in $P + \frac{m}{m_s} - 2$ steps, each consisting of a varying number of concurrent NBFTs.

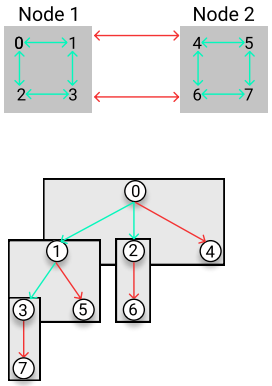


Figure 4.6: A binomial tree made of eight processes. Rectangles above present nodes on the cluster each of which consist of a quad-core processor. Ranks are mapped using sequential mapping algorithm. Below, green and red arrows in the binomial tree represent shared memory and network message transmission respectively. Rectangles represent the non-blocking flat trees (NBFTs) executed by Algorithm 1.

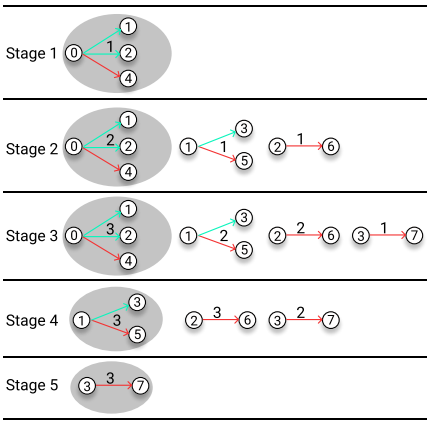


Figure 4.7: Execution stages of the binomial broadcast algorithm employing the non-blocking flat trees (NBFTs) broadcasts on the cluster of two nodes each of which composes of a quad-core processor described in Figure 4.6. The message is split up into $n_s = 3$ segments. Each arrow represents transmission of a segment through shared memory (green) and network (red) channels. The number over the arrow gives the index of the broadcast segment. The execution time of the stage is equal to the execution time of the grey coloured NBFT.

The number of parallel running NBFTs at the ith execution stage of the chain tree algorithm, j, varies as $1 \leq j \leq P-1$. Therefore, the execution time of the chain tree algorithm can be estimated as

$$T_{chain}(P, m, m_s) = \sum_{i=1}^{P+\frac{m}{m_s}-2} \max_j T_{NBFT}\left(P_{ij}, C_{ij}, m_s\right). \tag{4.25}$$

where $P_{ij} = 2$.

4.3.4 Binary Tree Algorithm

In Open MPI, the binary tree broadcast algorithm is segmentation-based and uses the balanced binary tree topology (see Figure c). The root broadcasts each segment to its children using the NBFT. Upon receipt of next segment, each internal node acts similarly. As the height of the balanced binary tree is equal to $\lfloor \log_2 P \rfloor$, the algorithm will be completed in $\lfloor \log_2 P \rfloor + \frac{m}{m_s} - 1$ steps, each consisting of a varying number of concurrent non-blocking flat tree broadcasts, involving 3 processes. The number of parallel running NBFTs at the ith execution stage of the binary tree algorithm, j, varies as $1 \leq j \leq 2^{\lfloor log_2 P \rfloor}$. Therefore,

$$T_{binary}(P, m, m_s) = \sum_{i=1}^{\lfloor \log_2 P \rfloor + \frac{m}{m_s} - 1} \max_j T_{NBFT}\left(P_{ij}, C_{ij}, m_s\right), \tag{4.26}$$

where $P_{ij} = 3$

4.3.5 K-chain Tree Algorithm

In Open MPI, the K-chain tree algorithm is implemented using non-blocking communication and message segmentation. In the K-chain tree, the root node has $K (K > 1)$ children, while the internal nodes have a single child each (Figure d). As the height of the tree is $\lfloor \frac{P-1}{K} \rfloor$, the algorithm takes $\lfloor \frac{P-1}{K} \rfloor + \frac{m}{m_s} - 1$ steps to complete. The number of parallel running NBFT at the ith execution stage of the k-chain tree algorithm, j, varies as $1 \leq j \leq (P - K - 1)$. Note,

that while Open MPI employs concurrent NBFTs, in this case the most of the NBFTs will be equivalent to non-blocking point-to-point communications.

$$T_{kchain}(P, m, m_s) = \sum_{i=1}^{\lfloor \frac{P-1}{K} \rfloor + \frac{m}{m_s} - 1} \max_j T_{NBFT}\left(P_{ij}, C_{ij}, m_s\right), \tag{4.27}$$

where $P_{ij} \in \{1, K\}$.

4.3.6 Split-Binary Tree Algorithm

In Open MPI, the split-binary tree algorithm is segmentation-based and implemented using blocking *send* and non-blocking *receive* routines. This is the difference from other segmented tree-based broadcast algorithms which all use non-blocking standard-mode send. However, because in Open MPI the segment size m_s is selected so that the blocking sends in the split binary tree will be executed in the buffered mode, we approximate the execution time of all flat tree broadcast algorithms in the split binary by the execution time of an NBFT.

The split binary algorithm consists of two phases – *forwarding* and *exchange*. In the first phase, the message of size m is split into two equal parts in the root, which are then sent to the left and right subtrees respectively using message segmentation.

After completion of the first phase, each node in the left subtree contains the first half of the message and each node in the right subtree – the second half of the message. Because of segmentation, each node will receive $\frac{m}{2 \cdot m_s}$ segments during the first phase.

As the balanced binary tree virtual topology is employed in the split-binary tree algorithm, each node in the left subtree will have a matching pair in the right subtree and vice versa. In the second phase, each pair of matching nodes in the left and right subtrees exchange their halves of the message. The execution time of the split-binary tree broadcast will be equal to the sum of the execution times of the first and the second phases. As the height of the balanced binary tree is equal to $\lfloor \log_2 P \rfloor$, the first phase will be completed in

$\lfloor log_2 P \rfloor + \frac{m}{2m_s} - 1$ steps. While in the second phase each pair of matching nodes send/receive message at the same time, the execution time of the second phase will be equal to $\max_{c_k} T_{p2p}^{c_k}\left(\frac{m}{2}\right)$ where $1 \leq k \leq \lfloor \frac{P-1}{2} \rfloor$ and $c_k \in \{0, 1\}$. Thus, the time to complete the split-binary tree algorithm will be estimated as follows:

$$T_{split\_binary}(P, m, m_s) = \sum_{i=1}^{\lfloor \log_2 P \rfloor + \frac{m}{2m_s} - 1} \max_j T_{NBFT}\left(P_{ij}, C_{ij}, m_s\right) + \max_{c_k} \left\{ T_{p2p}^{c_k}\left(\frac{m}{2}\right) \right\}, \quad (4.28)$$

where $P_{ij} = 3$.

Chapter 5

Estimation of Model Parameters

This section presents the proposed approach to estimation of the parameters of analytical performance models of MPI collective algorithms. We estimate the parameters separately for each algorithm and include the modelled collective algorithm in the communication experiment, which is used to estimate the model parameters. The same fixed segment size is used in the estimation of model parameters for all segmented collective algorithms. The segment size used in this thesis is less than Open MPI eager threshold. Thus, only the eager protocol is employed for segmented collective algorithms in our communication experiments. As stated in Chapter 4, the approach to estimation of the model parameters are presented for two different configurations as well: *One-Processes-Per-CPU* and *One-Processes-Per-Core*.

5.1 One-Processes-Per-CPU

As presented in Section 4.2, the analytical model of an Open MPI collective algorithm uses three platform parameters – α, β, and $\gamma(p)$. The traditional state-of-the-art approach to estimation of α and β would be to find these parameters from a number of point-to-point communication experiments. Namely, the time of a round-trip of a message of size m, $RTT(m)$, is measured for a wide range of m. From these experiments, a system of linear

equations with α and β as unknowns is derived. Then, linear regression is applied to find α and β. The found values of α and β would be then used in all analytical predictive formulas.

This approach yields a unique single pair of (α, β) for each target platform. Unfortunately, with α and β found this way, not all our analytical formulas will be accurate enough to be used for accurate selection of the best performing collective algorithm. Using non-linear regression does not improve the situation as the function $RTT(m)$ is typically near linear. Therefore, we propose to estimate the model parameters separately for each collective algorithm. More specifically, we propose to design a specific communication experiment for each collective algorithm, so that the algorithm itself would be involved in the execution of the experiment. Moreover, the execution time of this experiment must be dominated by the execution time of this collective algorithm. Then, we conduct a number of experiments on the target platform for a range of numbers of processors, p, and message sizes, m. From those experiments, we can derive a sufficiently large number of equations with α, β, and $\gamma(p)$ as unknowns, and then use an appropriate solver to find their values.

Unfortunately, when applied straightforwardly, this approach yields a system of *non-linear* equations like the one shown in Figure 5.1. This nonlinearity makes the task of estimation of the parameters mathematically very difficult, because we need to solve a large system of nonlinear equations.

Our approach to this problem is the following. As the non-linearity is caused by multiplicative terms involving $\gamma(p)$, we separate the estimation of $\gamma(p)$ from the estimation of α and β. Namely, we assume that $\gamma(p)$ is algorithm-independent and design a separate communication experiment for its estimation. The values of $\gamma(p)$ found from this experiment are then used as known coefficients in the algorithm-specific systems of equations for α and β. We present this approach in Sections 5.2.1 and 5.2.2.

$$\begin{cases} (\frac{P-1}{K} + \gamma(K+1) \cdot n_{s_1} - 1) \cdot (\alpha + \frac{m_1}{n_{s_1}} \cdot \beta) + (P-1) \cdot (\alpha + m_{g_1} \cdot \beta) = T_1 \\ (\frac{P-1}{K} + \gamma(K+1) \cdot n_{s_2} - 1) \cdot (\alpha + \frac{m_2}{n_{s_2}} \cdot \beta) + (P-1) \cdot (\alpha + m_{g_2} \cdot \beta) = T_2 \\ \dots \\ (\frac{P-1}{K} + \gamma(K+1) \cdot n_{s_M} - 1) \cdot (\alpha + \frac{m_M}{n_{s_2}} \cdot \beta) + (P-1) \cdot (\alpha + m_{g_M} \cdot \beta) = T_M \end{cases}$$

$$\Downarrow$$

$$\begin{cases} \alpha + \frac{(\frac{P-1}{K} + \gamma(K+1) \cdot n_{s_1} - 1) \cdot \frac{m_1}{n_{s_1}} + (P-1) \cdot m_{g_1}}{\frac{(P-1) \cdot (1+K)}{K} + \gamma(K+1) \cdot n_{s_1} - 1} \cdot \beta = \frac{T_1}{\frac{(P-1) \cdot (1+K)}{K} + \gamma(K+1) \cdot n_{s_1} - 1} \\ \alpha + \frac{(\frac{P-1}{K} + \gamma(K+1) \cdot n_{s_2} - 1) \cdot \frac{m_2}{n_{s_2}} + (P-1) \cdot m_{g_2}}{\frac{(P-1) \cdot (1+K)}{K} + \gamma(K+1) \cdot n_{s_2} - 1} \cdot \beta = \frac{T_2}{\frac{(P-1) \cdot (1+K)}{K} + \gamma(K+1) \cdot n_{s_2} - 1} \\ \dots \\ \alpha + \frac{(\frac{P-1}{K} + \gamma(K+1) \cdot n_{s_M} - 1) \cdot \frac{m_M}{n_{s_M}} + (P-1) \cdot m_{g_M}}{\frac{(P-1) \cdot (1+K)}{K} + \gamma(K+1) \cdot n_{s_M} - 1} \cdot \beta = \frac{T_M}{\frac{(P-1) \cdot (1+K)}{K} + \gamma(K+1) \cdot n_{s_M} - 1} \end{cases}$$

Figure 5.1: A system of M non-linear equations with α, β, and $\gamma(K+1)$ as unknowns, derived from M communication experiments, each consisting of the execution of the K-Chain tree broadcast algorithm, broadcasting a message of size m_i ($i = 1, ..., M$) from the root to the remaining $P-1$ processes, followed by the linear gather algorithm without synchronisation, gathering messages of size m_{g_i} ($m_{g_i} < E$ and $m_{g_i} \neq m_s$) on the root. The execution times, T_i, of these experiments are measured on the root.
Given $\gamma(K+1)$ is evaluated separately, the system becomes a system of M linear equations with α and β as unknowns.

5.1.1 Estimation of $\gamma(p)$

The model parameter $\gamma(p)$ appears in the formula estimating the execution time of the linear tree broadcast algorithm with non-blocking communication, which is only used for broadcasting of a segment in the tree-based segmented broadcast algorithms. Thus, in the context of Open MPI, the linear tree broadcast algorithm with non-blocking communication will always broadcast a message of size m_s to a relatively small number of processes.

According to Formula 4.4,

$$\gamma(p) = \frac{T^{nonblock}_{linear\_bcast}(p,m_s)}{T_{p2p}(m_s)} = \frac{T^{nonblock}_{linear\_bcast}(p,m_s)}{T^{nonblock}_{linear\_bcast}(2,m_s)} .$$

Therefore, in order to estimate $\gamma(p)$ for a given range of the number of processes, $p \in \{2, ..., P\}$, we need a method for estimation of $T^{nonblock}_{linear\_bcast}(p, m_s)$. We use the following method:

- For each $2 \leq p \leq P$, we measure on the root the execution time $T_1(p, N)$ of N successive calls to the *linear tree with non-blocking communication* broadcast routine separated by barriers. The routine broadcasts a message of size m_s.
- We estimate $T^{nonblock}_{linear\_bcast}(p, m_s)$ as $T_2(p) = \frac{T_1(p,N)}{N}$.

The experimentally obtained discrete function $\frac{T_2(p)}{T_2(2)}$ is used as a platform-specific but algorithm-independent estimation of $\gamma(p)$.

From our experiments, we observed that the discrete estimation of $\gamma(p)$ is near linear. Therefore, as an alternative for platforms with very large numbers of processors, we can build by linear regression a linear approximation of the discrete function $\frac{T_2(p)}{T_2(2)}$, obtained for a representative subset of the full range of p, and use this linear approximation as an analytical estimation of $\gamma(p)$.

5.1.2 Estimation of Algorithm Specific α And β

To estimate the model parameters α and β for a given collective algorithm, we design a communication experiment, which starts and finishes on the root (in order to accurately measure its execution time using the root clock), and

involves the execution of the modelled collective algorithm so that the total time of the experiment would be dominated by the time of its execution.

For example, for all broadcast algorithms, the communication experiment consists of a broadcast of a message of size m (where m is a multiple of segment size m_s), using the modelled broadcast algorithm, followed by a *linear-without-synchronisation* gather algorithm, gathering messages of size m_g ($m_g < E$) on the root. The execution time of this experiment on P nodes, $T_{bcast\_exp}(P, m)$, can be estimated as follows:

$$T_{bcast\_exp}(P, m) = T_{bcast\_alg}(P, m) + T_{linear\_gather}(P, m_g) \tag{5.1}$$

Using analytical formulas from Section 4.2 for $T_{bcast\_alg}(P, m)$ and $T_{linear\_gather}(P, m_g)$, for each combination of P and m this experiment will yield one linear equation with α and β as unknowns. By repeating this experiment with different P and m, we obtain a system of linear equations for α and β. Each equation in this system can be represented in the canonical form, $\alpha + \beta \times m_i = T_i$ ($i = 1, ..., M$). Finally, we use the least-square regression to find α and β, giving us the best linear approximation $\alpha + \beta \times m$ of the discrete function $f(m_i) = T_i$ ($i = 1, ..., M$).

Figure 5.1 shows a system of linear equations built for the K-Chain tree broadcast algorithm for our experimental platform. To build this system, we used the same P nodes in all experiments but varied the message size $m \in \{m_1, ..., m_M\}$. With M different message sizes, we obtained a system of M equations. The number of nodes, P, was approximately equal to the half of the total number of nodes. We observed that the use of larger numbers of nodes in the experiments will not change the estimation of α and β.

5.2 One-Process-Per-Core

As presented in Section 4.3, the analytical models of the Open MPI broadcast algorithms use the τ-Lop model parameters, the ratio of delays of the communication channels $Q(m)$, and the parallelisation factor $\gamma^c(P)$ as the model parameters. The traditional state-of-the-art approach to estimation

of model parameters would be to find these parameters from a number of point-to-point communication experiments. Namely, the time of a round-trip of a message of size m, $RTT(m)$, is measured for a wide range of m. From these experiments, a system of linear equations with unknown model parameters is derived. Then, linear regression is applied to find model parameters.

This approach yields a unique single set of parameters for each target platform. Unfortunately, with model parameters found this way, not all our analytical formulas will be accurate enough to be used for accurate selection of the best performing broadcast algorithm. Using non-linear regression does not improve the situation as the function $RTT(m)$ is typically near linear. Therefore, we propose to estimate the model parameters separately for each broadcast algorithm. More specifically, we propose to design a specific communication experiment for each broadcast algorithm, so that the algorithm itself would be involved in the execution of the experiment. Moreover, the execution time of this experiment must be dominated by the execution time of this broadcast algorithm. Then, we conduct a number of experiments on the target platform for a range of numbers of processors and message sizes. From those experiments, we can derive a sufficiently large number of equations with unknown model parameters, and then use an appropriate solver to find their values.

Our approach to this problem is the following. We consider $\gamma^c(P)$ and $Q(m)$ platform-specific but algorithm-independent parameters and design a separate communication experiment for their estimation. The values of $\gamma^c(P)$ and $Q(m)$ found from this experiment are then used as known constants in the algorithm-specific systems of equations for the τ-Lop model parameters. We present this approach in Sections 5.2.1 and 5.2.2. The motivation behind assuming $\gamma^c(P)$ and $Q(m)$ algorithm-independent is that otherwise they would appear in the derived equations as unknowns and make the the equations non-linear.

5.2.1 Estimation of $\gamma^c(P)$ And $Q(m)$

The model parameter $\gamma^c(P)$ represents the increase in the cost of the overlapping $P-1$ non-blocking transmissions through the network in the NBFT with respect to a single point-to-point message transmission. The NBFT is only used for broadcasting of a segment in the tree-based segmented broadcast algorithms. Thus, in the context of Open MPI, the NBFT will always broadcast a message of size m_s to a relatively small number of processes.

According to Formula (4.23),

$$\gamma^c(P) = \frac{T^c_{NBFT}(P, m_s)}{T^c_{p2p}(m_s)} = \frac{T^c_{NBFT}(P, m_s)}{T^c_{NBFT}(2, m_s)}. \tag{5.2}$$

where $P \in \{2, ..., P^{max(c)}_{NBFT}\}$. $P^{max(c)}_{NBFT}$ is the maximum number of the processes communicating through channel c in NBFTs on a given platform. Therefore, in order to estimate $\gamma^c(P)$, we need a method for estimation of $T^c_{NBFT}(P, m_s)$. We use the following method:

- For each $2 \leq P \leq P^{max(c)}_{NBFT}$, we measure on the root the execution time $\theta^c(P, N)$ of N successive calls of the NBFT separated by barriers. The routine broadcasts a message of size m_s.
- We estimate $T^c_{NBFT}(P, m_s)$ as

$$T^c_{NBFT}(P, m_s) = \frac{\theta^c(P, N)}{N}.$$

The experimentally obtained discrete function $\frac{T^c_{NBFT}(P, m_s)}{T^c_{NBFT}(2, m_s)}$ is used as a platform-specific but algorithm-independent estimation of $\gamma^c(P)$.

Parameter $Q(m)$ only appears in the analytical models of the Open MPI broadcast algorithms in the context of NBFTs broadcasting a segment of fixed size m_s (see Section 4.3.1). Therefore, we only need to estimate $Q(m_s)$. By definition $T^c_{NBFT}(2, m_s) = T^c_{p2p}(m_s)$, and using formula (4.18) we estimate $Q(m_s)$ as

$$Q(m_s) = \frac{T^1_{NBFT}(2, m_s)}{T^0_{NBFT}(2, m_s)} \tag{5.3}$$

5.2.2 Estimation of Algorithm Specific Model Parameters

To estimate the model parameters for a given broadcast algorithm, we design a communication experiment, which starts and finishes on the root (in order to accurately measure its execution time using the root clock), and involves the execution of the modelled broadcast algorithm so that the total time of the experiment would be dominated by the time of its execution. In this section, we present the estimation of τ-Lop model parameters.

For all broadcast algorithms, the communication experiment consists of a broadcast of a message of size m (where m is a multiple of segment size m_s), using the modelled broadcast algorithm, followed by the *flat-without-synchronisation* gather algorithm. This algorithm works by gathering messages of size m_s on the root. The execution time of this experiment on P nodes, $T_{comm\_experiment}(P, m)$, can be estimated as follows,

$$T_{comm\_experiment}(P, m) = \\ T_{bcast}(P, m) + T_{flat\_gather}(P, m_s). \tag{5.4}$$

The execution time of the flat-without-synchronisation gather algorithm, gathering a segment of size m_g on the root from $P-1$ processes, is estimated as follows [124],

$$T_{flat\_gather}(P, m_s) = \sum_{i=1}^{P-1} T^{c_i}_{p2p}(m_s) = \\ \sum_{i=1}^{P-1} \begin{cases} T^0_{p2p}(m_s), \text{ if } c_i = 0 \\ T^1_{p2p}(m_s), \text{ if } c_i = 1 \end{cases} = \\ \sum_{i=1}^{P-1} \begin{cases} o^0 + 2L^0, \text{ if } c_i = 0 \\ o^1 + 2L^0 + L^1, \text{ if } c_i = 1 \end{cases} \tag{5.5}$$

where o^0, o^1, L^0 and L^1 denote $o^0(m_s)$, $o^1(m_s)$, $L^0(m_s)$ and $L^1(m_s)$

respectively. Thus, as $\{c_i\}_{i=1}^{P-1}$ are all knowns for the experimental setup, $T_{flat\_gather}(P, m_s)$ is estimated as a linear function of unknown τ-Lop model parameters o^0, o^1, L^0 and L^1.

To explain in detail the contribution of the broadcast algorithm in the estimated time of the experiment, we assume the binomial tree broadcast algorithm. Therefore, according to formulas (4.18), (4.19) and (4.24), the execution time of the broadcast algorithm will be expressed as follows,

$$
\begin{aligned}
T_{bcast}(P,m) = T_{binomial}(P,m) = \\
\sum_{i=1}^{N} \max_j T_{NBFT}\left(P_{ij}, C_{ij}, m_s\right) = \\
\sum_{i=1}^{N} \max_j \begin{cases} \gamma^0(P_{ij}) \cdot T^0_{p2p}(m_s), \text{if } C_{ij} = 0 \\ \gamma^1(P^\circ_{ij}) \cdot T^1_{p2p}(m_s), \text{otherwise} \end{cases} = \\
\sum_{i=1}^{N} \max_j \begin{cases} \frac{\gamma^0(P_{ij})}{Q(m_s)} \cdot T^1_{p2p}(m_s), \text{if } C_{ij} = 0 \\ \gamma^1(P^\circ_{ij}) \cdot T^1_{p2p}(m_s), \text{otherwise} \end{cases} = \\
T^1_{p2p}(m_s) \cdot \sum_{i=1}^{N} \max_j \begin{cases} \frac{\gamma^0(P_{ij})}{Q(m_s)}, \text{if } C_{ij} = 0 \\ \gamma^1(P^\circ_{ij}), \text{otherwise} \end{cases} = \\
\left(o^1 + 2L^0 + L^1\right) \cdot \sum_{i=1}^{N} \max_j \begin{cases} \frac{\gamma^0(P_{ij})}{Q(m_s)}, \text{if } C_{ij} = 0 \\ \gamma^1(P^\circ_{ij}), \text{otherwise} \end{cases}
\end{aligned}
\tag{5.6}
$$

where N is the number of execution stages of the algorithm, $N = \lfloor log_2 P \rfloor + \frac{m}{m_s} - 1$; P_{ij} and C_{ij} are the number of nodes and the number of network point-to-point communications in the j-th NBFT at i-th stage respectively; $P^\circ_{ij} = C_{ij} + \lfloor \frac{P_{ij} - C_{ij} - 1}{Q(m_s)} \rfloor$.

In this experiment, $\{P_{ij}\}$ and $\{C_{ij}\}$ are all knowns. As presented in Section 5.2.1, $\gamma^c(P)$ and $Q(m_s)$ are algorithm-independent parameters, which are estimated separately, before the estimation of algorithm-specific $\tau-$Lop parameters. Therefore, $P^\circ_{ij} = C_{ij} + \lfloor \frac{P_{ij} - C_{ij} - 1}{Q(m_s)} \rfloor$, $\frac{\gamma^0(P_{ij})}{Q(m_s)}$ and $\gamma^1(P^\circ_{ij})$ are also all knowns for all i and j. Thus, like $T_{flat\_gather}(P, m)$, $T_{bcast}(P, m)$ is also estimated as a linear function of unknown τ-Lop model

$$\begin{cases} \lambda_{11} \cdot o^0 + \lambda_{12} \cdot o^1 + \lambda_{13} \cdot L^0 + \lambda_{14} \cdot L^1 = T_1 \\ \lambda_{21} \cdot o^0 + \lambda_{22} \cdot o^1 + \lambda_{23} \cdot L^0 + \lambda_{24} \cdot L^1 = T_2 \\ \dots \\ \lambda_{M1} \cdot o^0 + \lambda_{M2} \cdot o^1 + \lambda_{M3} \cdot L^0 + \lambda_{M4} \cdot L^1 = T_M \end{cases}$$

Figure 5.2: A system of M linear equations with o^0, o^1, L^0 and L^1 as unknowns, derived from M communication experiments, each consisting of the execution of the binomial tree broadcast algorithm, broadcasting a message of size m_i ($i = 1, ..., M$) from the root to the remaining $P - 1$ processes, followed by the flat-without-synchronisation gather algorithm, gathering messages of size m_s (segment size) on the root. The execution times, T_i, of these experiments are measured on the root.

parameters o^0, o^1, L^0 and L^1.

Therefore, for each pair of (P, m), formula (5.4) will yield one linear equation with unknown τ-Lop model parameters of the form

$$T_{comm\_experiment}(P, m) = \lambda_1 \cdot o^0 + \lambda_2 \cdot o^1 + \lambda_3 \cdot L^0 + \lambda_4 \cdot L^1 \qquad (5.7)$$

where λ_1, λ_2, λ_3 and λ_4 are constants. By repeating this experiment with different m, we obtain a system of linear equations for o^0, o^1, L^0 and L^1 (Figure 5.2). Each equation in this system is represented in the canonical form, $\lambda_{i1} \cdot o^0 + \lambda_{i2} \cdot o^1 + \lambda_{i3} \cdot L^0 + \lambda_{i4} \cdot L^1 = T_i, (i = 1, ..., M)$. Finally, we use the least-square regression to find unknown model parameters. Similarly, we build systems of linear equations for other broadcast algorithms implemented in Open MPI.

Chapter 6

Experimental Results

The goal of this thesis is selection of the optimal collective algorithms using analytical performance models. To achieve this, we derive new analytical performance models from implementing code in Chapter 4 and propose a new method to measure model parameters in Chapter 5. This chapter presents an experimental validation of the proposed approach to selection of the optimal collective algorithms using Open MPI broadcast and gather operations. The approach is validated for two different configurations: (1) MPI processes are mapped by CPU, *One-Process-Per-CPU*. (2) MPI processes are mapped by CPU-core, *One-Process-Per-Core*. In all experiments, we use the default Open MPI configuration (without any collective optimization tuning).

6.1 One-Process-Per-CPU

6.1.1 Experiment Setup

For experiments, we use Open MPI 3.1 running on a dedicated Grisou and Gros clusters of the Nancy site of the Grid‘5000 infrastructure [38]. The Grisou cluster consists of 51 nodes each with 2 Intel Xeon E5-2630 v3 CPUs (8 cores/CPU), 128GB RAM, 2x558GB HDD, interconnected via 10Gbps Ethernet. The Gros cluster consists of 124 nodes each with Intel Xeon Gold 5220 (18 cores/CPU), 96GB RAM, 894GB SSD, interconnected via 2 x 25Gb

Number of processes (p)	$\gamma(p)$	
	Grisou	Gros
3	1.114	1.084
4	1.219	1.17
5	1.283	1.254
6	1.451	1.339
7	1.540	1.424

Table 6.1: Estimated values of $\gamma(p)$ on Grisou and Gros clusters.

Ethernet. We do not utilize the InfiniBand interconnect on the Grisou cluster in our experiments.

To make sure that the experimental results are reliable, we follow a detailed methodology: 1) We make sure that the cluster is fully reserved and dedicated to our experiments. 2) For each data point in the execution time of collective algorithms, the sample mean is used, which is calculated by executing the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. We also check that the individual observations are independent and their population follows the normal distribution. For this purpose, MPIBlib [77] is used.

In our communication experiments, MPI programs use the one-process-per-CPU configuration, and the maximal total number of processes is equal to 90 on Grisou and 124 on Gros clusters. The message segment size, m_s, for segmented broadcast algorithms is set to 8KB and is the same in all experiments. This segment size is commonly used for segmented broadcast algorithms in Open MPI. Selection of optimal segment size is out of the scope of this thesis.

6.1.2 Experimental Estimation of Model Parameters

First of all, we would like to stress again that we estimate model parameters for each cluster separately.

Estimation of parameter $\gamma(p)$ for our experimental platforms follows the method presented in Section 5.2.1. With the maximal number of processes

equal to 90 (Grisou) and 124 (Gros), the maximal number of children in the linear tree broadcast algorithm with non-blocking communication, used in the segmented Open MPI broadcast algorithms, will be equal to seven. Therefore, the number of processes in our communication experiments ranges from 2 to 7 for both clusters. By definition, $\gamma(2) = 1$. The estimated values of $\gamma(p)$ for p from 3 to 7 are given in Table 6.1.

After estimation of $\gamma(p)$, we conduct communication experiments to estimate algorithm-specific values of parameters α and β for six broadcast algorithms and three gather algorithms following the method described in Section 5.2.2. In these experiments we use 40 processes on Grisou and 124 on Gros. The message size, m, varies in the range from 8KB to 4MB in the broadcast experiments, and from 64KB to 1MB in the gather experiments. We use 10 different sizes for broadcast algorithms, $\{m_i\}_{i=1}^{10}$, and 5 different sizes for gather algorithms, $\{m_i\}_{i=1}^{5}$, separated by a constant step in the logarithmic scale, $\log m_{i-1} - \log m_i = const$. Thus, for each collective algorithm, we obtain a system of 10 linear equations with α and β as unknowns. We use the Huber regressor [125] to find their values from the system. It is less sensitive to outliers in data than the squared error loss. We visualized the lines built using the Ordinary Least Squares [126], Theil-Sen [127], Random Sample Consensus (RANSAC) [128] and Huber regressors [125], and the Huber regressor was the best fit (Figure 6.1). Besides, Table 6.2 shows that estimated negative values using OSL, Theil-Sen and RANSAC have no physical meaning.

	OSL	**Theil-Sen**	**RANSAC**
Linear tree	2.8×10^{-4}, 0.0	1.6×10^{-12}, 2.0×10^{-8}	3.5×10^{-5}, 0.0
K-Chain tree	-2.6×10^{-5}, 4.7×10^{-9}	-2.5×10^{-5}, 4.6×10^{-9}	-3.0×10^{-5}, 5.0×10^{-9}
Chain tree	6.3×10^{-4}, -2.7×10^{-8}	2.5×10^{-3}, -1.8×10^{-7}	7.7×10^{-3}, -6.2×10^{-7}
Split-binary tree	8.7×10^{-4}, -5.3×10^{-8}	1.0×10^{-3}, -6.4×10^{-8}	1.8×10^{-3},-1.1×10^{-7}
Binary tree	-2.2×10^{-5}, 3.9×10^{-9}	-1.8×10^{-5}, 3.7×10^{-9}	-2.3×10^{-5}, 4.0×10^{-9}
Binomial tree	-3.2×10^{-5}, 4.5×10^{-9}	-3.4×10^{-5},4.7×10^{-9}	-3.2×10^{-5}, 4.5×10^{-9}

Table 6.2: Estimated values of α and β using OSL, Theil-Sen and RANSAC algorithms for Open MPI broadcast algorithms.

The values of parameters α and β obtained this way can be found in Table 6.3 and Table 6.4. We can see that the values of α and β do vary depending

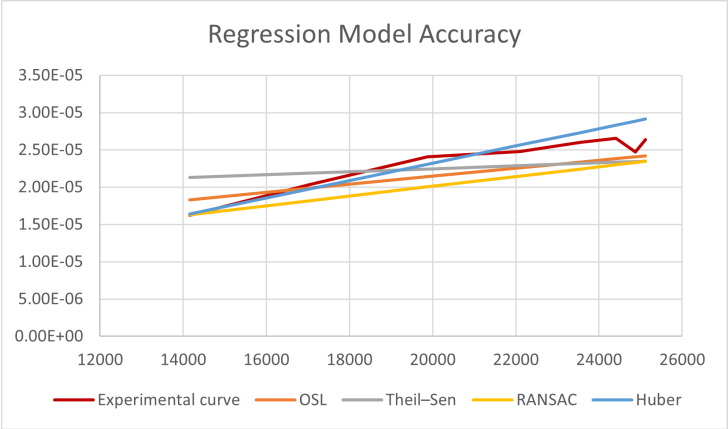


Figure 6.1: Comparison of the accuracy of the linear regression algorithms. Experimental curve plotted using the system of linear equations presented in Figure 5.1 where $P = 50$. X and Y axises present β coefficient and left side coefficient respectively in the second system of linear equations in Figure 5.1.

on the collective algorithm, and the difference is more significant between algorithms implementing different collective operations. The results support our original hypothesis that the average execution time of a point-to-point communication will very much depend on the context of the use of the point-to-point communications in the algorithm. Therefore, the estimated values of the α and β capture more than just sheer network characteristics. One interesting example is the Split-binary tree and Binary tree broadcast algorithms. They both use the same virtual topology, but the estimated time of a point-to-point communication, $\alpha + \beta \times m$, is smaller in the context of the Split-binary one. This can be explained by a higher level of parallelism of the Split-binary algorithm, where a significant part of point-to-point communications is performed in parallel by a large number of independent pairs of processes from the left and right subtrees.

Collective algorithm	$\alpha(sec)$	$\beta\ (\frac{sec}{byte})$
Broadcast		
Linear tree	2.2×10^{-12}	1.8×10^{-8}
K-Chain tree	5.7×10^{-13}	4.7×10^{-9}
Chain tree	6.1×10^{-13}	4.9×10^{-9}
Split-binary tree	3.7×10^{-13}	3.6×10^{-9}
Binary tree	5.8×10^{-13}	4.7×10^{-9}
Binomial tree	5.8×10^{-13}	4.8×10^{-9}
Gather		
Linear tree without synchronisation	1.5×10^{-5}	1.5×10^{-9}
Binomial tree	1.2×10^{-4}	8.6×10^{-10}
Linear tree with synchronisation	5.9×10^{-5}	9.4×10^{-10}

Table 6.3: Estimated values of α and β for the Grisou cluster and Open MPI broadcast and gather algorithms.

Collective algorithm	$\alpha(sec)$	$\beta\ (\frac{sec}{byte})$
Broadcast		
Linear tree	1.4×10^{-12}	1.1×10^{-8}
K-Chain tree	5.4×10^{-13}	4.5×10^{-9}
Chain tree	4.7×10^{-12}	3.8×10^{-8}
Split-binary tree	5.5×10^{-13}	4.5×10^{-9}
Binary tree	5.8×10^{-13}	4.7×10^{-9}
Binomial tree	1.2×10^{-13}	1.0×10^{-9}
Gather		
Linear tree without synchronisation	1.3×10^{-4}	3.4×10^{-10}
Binomial tree	1.0×10^{-4}	4.2×10^{-10}
Linear tree with synchronisation	9.5×10^{-5}	3.9×10^{-10}

Table 6.4: Estimated values of α and β for the Gros cluster and Open MPI broadcast and gather algorithms.

6.1.3 Accuracy of Selection of Optimal Collective Algorithms Using The Constructed Analytical Performance Models

The constructed analytical performance models of the Open MPI broadcast and gather collective algorithms are designed for the use in the MPI_Bcast and MPI_Gather routines for runtime selection of the optimal algorithm, depending on the number of processes and the message size. While the efficiency of the selection procedure is evident from the low complexity of the analytical formulas derived in Section 4.2, the experimental results on the accuracy are presented in this section.

Figure 6.2 and 6.3 show the results of our experiments for MPI_Bcast and MPI_Gather on Grisou and Gros clusters respectively. For both operations, we present results of experiments with 50, 80 and 90 processes on Grisou, and 80, 100 and 124 on Gros. The message size, m, varies in the range from 8KB to 4MB in the broadcast experiments, and from 64KB to 1MB in the gather experiments. We use 10 different sizes for broadcast algorithms, $\{m_i\}_{i=1}^{10}$, and 5 different sizes for gather algorithms, $\{m_i\}_{i=1}^{5}$, separated by a constant step in the logarithmic scale, $\log m_{i-1} - \log m_i = const$. The graphs show the execution time of the collective operation as a function of message size. Each data point on a blue line shows the performance of the algorithm selected by the Open MPI decision function for the given operation, number of processes and message size. Each point on a red line shows the performance of the algorithm selected by our decision function, which uses the constructed analytical models. Each point on a green line shows the performance of the best Open MPI algorithm for the given collective operation, number of processes and message size.

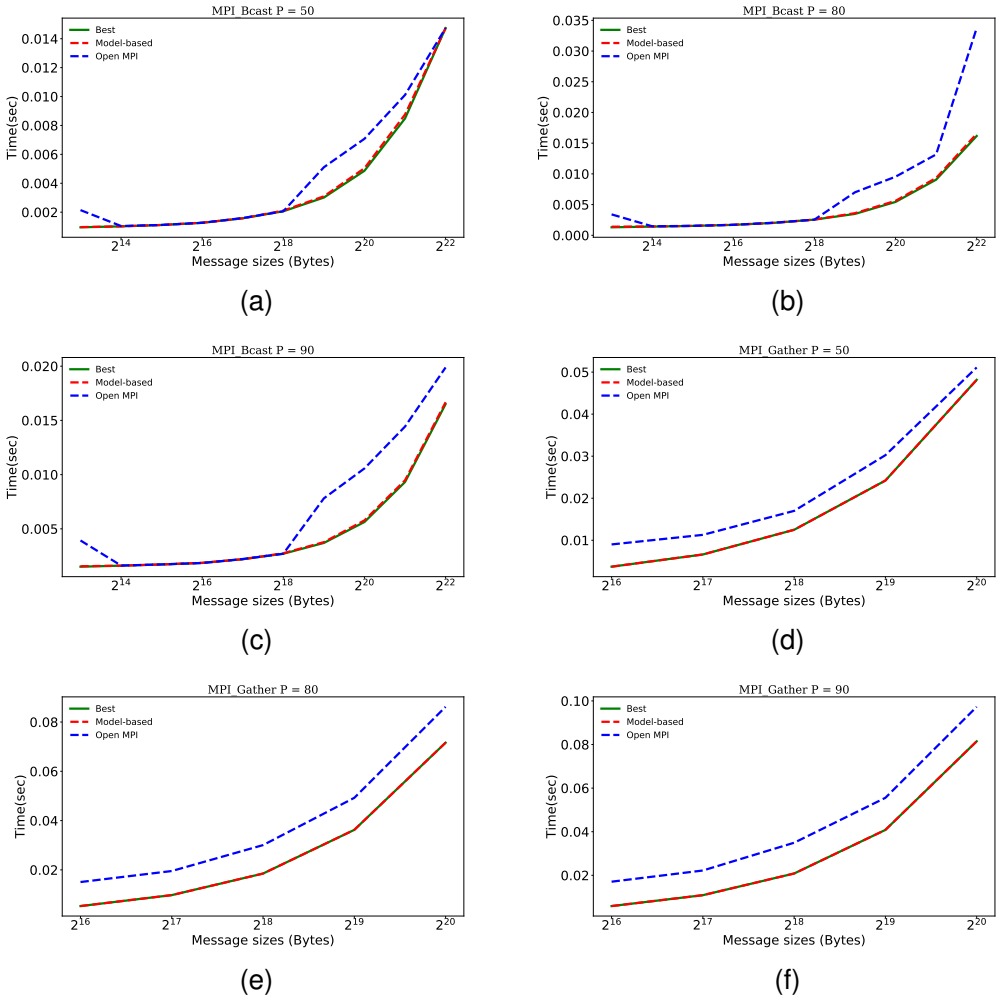


Figure 6.2: Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast and MPI_Gather on Grisou cluster.

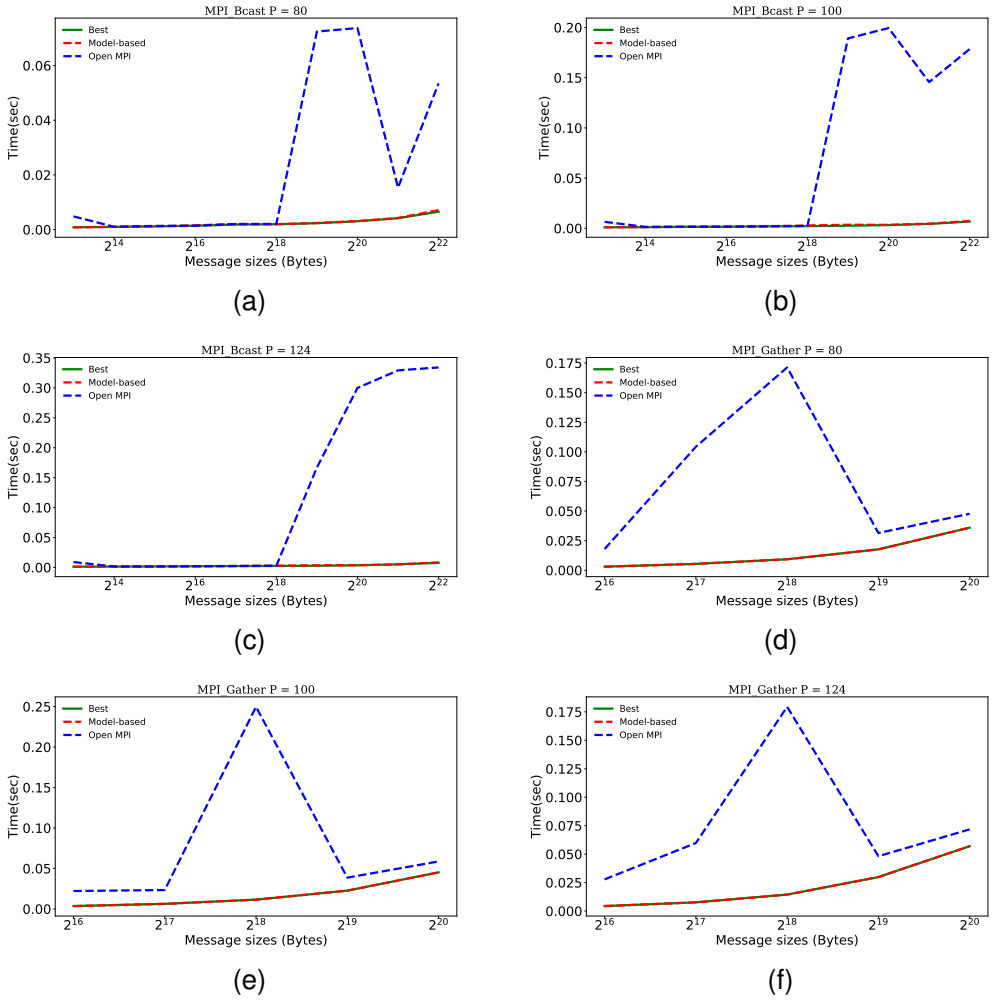


Figure 6.3: Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast and MPI_Gather on Gros cluster.

P=90, MPI_Bcast, Grisou

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
8	*binomial*	*binary* (3)	*split_binary* (160)
16	*binary*	*binary* (0)	*split_binary* (1)
32	*binary*	*binary* (0)	*split_binary* (0)
64	*split_binary*	*binary* (1)	*split_binary* (0)
128	*binary*	*binary* (0)	*split_binary* (1)
256	*split_binary*	*binary* (2)	*split_binary* (0)
512	*split_binary*	*binary* (2)	*chain* (111)
1024	*split_binary*	*binary* (3)	*chain* (88)
2048	*split_binary*	*binary* (2)	*chain* (55)
4096	*split_binary*	*binary* (1)	*chain* (20)

P=100, MPI_Bcast, Gros

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
8	*binary*	*binomial* (3)	*split_binary* (549)
16	*binomial*	*binomial* (0)	*split_binary* (32)
32	*binomial*	*binomial* (0)	*split_binary* (3)
64	*split_binary*	*binomial* (8)	*split_binary* (0)
128	*split_binary*	*binomial* (8)	*split_binary* (0)
256	*binary*	*binary* (0)	*split_binary* (6)
512	*binary*	*binary* (0)	*chain* (7297)
1024	*split_binary*	*binary* (7)	*chain* (6094)
2048	*split_binary*	*binary* (4)	*chain* (3227)
4096	*split_binary*	*binary* (9)	*chain* (2568)

Table 6.5: Comparison of the model-based and Open MPI selections with the best performing MPI_Bcast algorithm. For each selected algorithm, its performance degradation against the optimal one is given in braces.

P=90, MPI_Gather, Grisou

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
64	binomial	*binomial* (0)	linear_sync (190)
128	*binomial*	*binomial* (0)	*linear_sync* (104)
256	*binomial*	*binomial* (0)	*linear_sync* (68)
512	*binomial*	*binomial* (0)	*linear_sync* (36)
1024	*binomial*	*binomial* (0)	*linear_sync* (19)

P=100, MPI_Gather, Gros

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
64	*binomial*	*binomial* (0)	*linear_sync* (524)
128	*binomial*	*binomial* (0)	*linear_sync* (271)
256	*binomial*	*binomial* (0)	*linear_sync* (2073)
512	*binomial*	*binomial* (0)	*linear_sync* (71)
1024	*binomial*	*binomial* (0)	*linear_sync* (36)

Table 6.6: Comparison of the model-based and Open MPI selections with the best performing MPI_Gather algorithm. For each selected algorithm, its performance degradation against the optimal one is given in braces.

Table 6.5 presents selections made for MPI_Bcast using the proposed model-based runtime procedure and the Open MPI decision function. For each message size m, the best performing algorithm, the model-based selected algorithm, and the Open MPI selected algorithm are given. For the latter two, the performance degradation in percents in comparison with the best performing algorithm is also given. We can see that for the Grisou cluster, the model-based selection either picks the best performing algorithm, or the algorithm, the performance of which deviates from the best no more than 3%. Given the accuracy of measurements, this means that the model-based selection is practically always optimal as the performance of the selected algorithm is indistinguishable from the best performance. The Open MPI selection is near optimal in 50% cases and causes significant, up to 160%, degradation in the remaining cases. For the Gros cluster, the model-based selection picks either the best performing algorithm or the algorithm with near optimal performance, no worse than 10% in comparison with the best performing algorithm. At the same time, while near optimal in

40% cases, the algorithms selected by the Open MPI demonstrate catastrophic degradation (up to 7297%) in 50% cases.

Table 6.6 shows results for MPI_Gather. One can see that the Open MPI selection significantly degrades the performance of MPI_Gather for all message sizes and numbers of processes (up to 190% on Grisou, and up to 2073% on Gros), while our model-based selection always picks the best performing algorithm.

The Open MPI decision functions select the algorithm depending on the message size and the number of processes. For example, the Open MPI broadcast decision function, shown in Listing 1.1, selects the chain broadcast algorithm for large message sizes. However, from Table 6.5 it is evident that the chain broadcast algorithm is not the best performing algorithm for large message sizes on both clusters. From the same table, one can see that the model-based selection procedure accurately picks the best performing binomial tree broadcast algorithm for 16KB and 32KB message sizes on the Gros cluster, where Open MPI only selects the binomial tree algorithm for broadcasting messages smaller than 2KB.

In this section, we present experimental validation of the proposed approach to selection of optimal broadcast algorithms on multi-core clusters.

6.1.4 Shaheen II

In Chapter 1 we discuss the remarkable growth in power and scale of supercomputers in the last 10 years. While our modelling approach is able to select the optimal collective algorithms accurately, due to the small number of nodes in the Grid5000 platform we could not validate the scalability of the proposed approach (see Section 6.1.3). Therefore in this section, we validate our approach in the large scale Shaheen II supercomputer using broadcast algorithms and we demonstrate that our approach is scalable in large scale platform.

Shaheen II is a supercomputer [40] owned by King Abdullah University of Science and Technology, Saudi Arabia. It consists of 6174 nodes (197568 cores) each with 2 Intel Haswell CPUs (16 processors core per CPU,

2.3GHz), 128 GB of memory per node, Cray Aries interconnect with Dragonfly topology. Unfortunately, we had limited access (limited time and sources) to the Shaheen II supercomputer, and therefore, we could use only 512 number of nodes in our experiments.

In our experiments, we use 512 nodes where MPI programs use the one-process-per-CPU. Thus, we have a total of 1024 MPI processes in all experiments. The message segment size, m_s, for segmented broadcast algorithms is set to 8KB. The estimated values of parameters α and β can be found in Table 6.7.

Collective algorithm	$\alpha(sec)$	β ($\frac{sec}{byte}$)
Broadcast		
Linear tree	2.6×10^{-12}	2.1×10^{-8}
K-Chain tree	2.5×10^{-13}	2.1×10^{-9}
Chain tree	2.1×10^{-13}	1.8×10^{-9}
Split-binary tree	3.4×10^{-13}	2.8×10^{-9}
Binary tree	3.1×10^{-13}	2.5×10^{-9}
Binomial tree	1.6×10^{-13}	1.3×10^{-9}

Table 6.7: Estimated values of α and β for the Shaheen II cluster and Open MPI broadcast algorithms.

P=1024, MPI_Bcast, Shaheen II

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
8	$binomial$	$binomial$ (0)	$split\_binary$ (44)
40	$binary$	$binary$ (0)	$split\_binary$ (6)
152	$split\_binary$	$split\_binary$ (0)	$split\_binary$ (0)
232	$binary$	$binary$ (0)	$binary$ (0)
328	$split\_binary$	$binary$ (1)	$split\_binary$ (0)
472	$binary$	$binary$ (0)	$chain$ (101)
648	$binary$	$binary$ (0)	$chain$ (146)
1032	$split\_binary$	$binary$ (1)	$chain$ (25)
2088	$split\_binary$	$binary$ (1)	$chain$ (27)
4840	$binary$	$binary$ (0)	$chain$ (16)

Table 6.8: Comparison of the model-based and Open MPI selections with the best performing MPI_Bcast algorithm. For each selected algorithm, its performance degradation against the optimal one is given in braces.

Figure 6.4 shows the results of our experiments in Shaheen II for

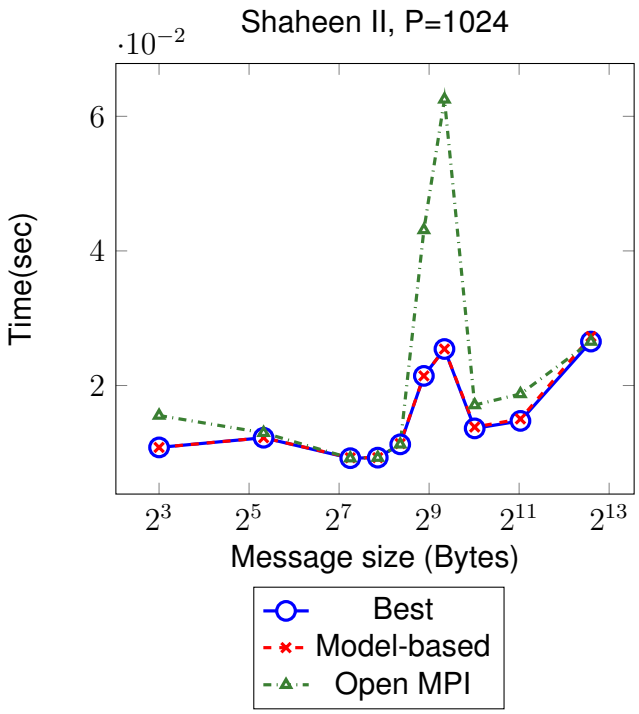


Figure 6.4: Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast on Shaheen II supercomputer.

MPI_Bcast. We present results of experiments with $P = 1024$ and the message size, m, varies in the range from 8KB to 4MB, and we use 10 different sizes, $m(KB) \in \{8, 40, 152, 232, 328, 472, 648, 1032, 2088, 4840\}$. The plot shows the execution time of the broadcast operation as a function of the message size. Each data point on a green line shows the performance of the algorithm selected by the Open MPI decision function for the given number of processes and message size. Each point on a red line shows the performance of the algorithm selected by our decision function, which uses the constructed analytical models. Each point on a blue line shows the performance of the best broadcast algorithm for MPI_Bcast.

Table 6.8 presents selections made for MPI_Bcast using the proposed model-based runtime procedure and the Open MPI decision function. For each message size m, the best performing algorithm, the model-based selected algorithm, and the Open MPI selected algorithm are given. For the

latter two, the performance degradation in percents in comparison with the best performing algorithm is also given. We can see that the model-based selection picks the best performing algorithm, or the algorithm, the performance of which deviates from the best no more than 1%. Given the accuracy of measurements, this means that the model-based selection is practically always optimal as the performance of the selected algorithm is indistinguishable from the best performance. The Open MPI selection is near optimal in 30% cases and causes significant, up to 146%, degradation in the remaining cases.

6.2 One-Process-Per-Core

6.2.1 Experimental Setup And Methodology

We validate our approach on three large scale clusters. Two clusters, Grisou and Gros, are located in France and belong to the Grid5000 experimental infrastructure. The third cluster, MareNostrum4, is hosted by Barcelona Supercomputing Center.

Grid5000 is the large scale testbed with seven sites in Grenoble, Luxembourg, Lyon, Nancy, Nantes, Rennes and Sophia. We run our experiments on Grisou and Gros clusters of the Nancy site using Open MPI 3.1. Grisou consists of 51 nodes each with 2 Intel Xeon E5-2630 v3 CPUs (8 cores/CPU), 128GiB RAM, interconnected via 10Gbps Ethernet. Gros consists of 124 nodes each with Intel Xeon Gold 5220 CPU (18 cores/CPU), 96 GiB RAM, interconnected via 25 Gbps Ethernet.

MareNostrum4 is a cluster based on Intel Xeon Platinum processors from the Skylake generation in the Barcelona Supercomputing Center. It consists of 3456 nodes each with 2-socket Intel Xeon Platinum 8160 CPU with 24 cores per socket, 96 GiB of main memory 1.880 GB/core, interconnected via 10 Gbit Ethernet.

In our collective experiments, we use up to 38 nodes in Grisou, up to 56 nodes in Gros, and up to 10 nodes in MareNostrum4. MPI programs use the one-process-per-CPU-core configuration, and the maximal total number of

processes is 600 for Grisou, 1000 for Gros and 480 for MareNostrum4. They utilize all CPU-cores in the nodes used in experiments. The default serial mapping of MPI processes to cores is used in all programs. The message segment size, m_s, for segmented broadcast algorithms is set to 8KB and is the same in all experiments. This very segment size is commonly used for segmented broadcast algorithms in Open MPI. Selection of optimal segment size is out of the scope of this thesis.

We follow a detailed methodology to make sure that the experimental results are reliable: 1) We make sure that the cluster is fully reserved and dedicated to our experiments. 2) For each data point in the execution time of collective algorithms, the sample mean is used, which is calculated by executing the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. We also check that the individual observations are independent and their population follows the normal distribution. For this purpose, MPIBlib [77] is used.

6.2.2 Experimental Estimation of Model Parameters

Platform-specific but algorithm-independent model parameters $\gamma^c(P)$ and $Q(m_s)$ are estimated first (and separately for each experimental cluster) following the methodology described in Section 5.2.1.

The calculated values of $P_{NBFT}^{max(1)}$ and $P_{NBFT}^{max(0)}$ for our experimental setups are given in Table 6.9a. $\gamma^0(P_{NBFT}^{max(0)})$ is estimated 1 for all the clusters. $Q(m_s)$ is estimated 6 for Grisou, 12 for Gros and 9 for MareSotrum4. The estimated values of $\gamma^1(P)$ for P from 3 to 7 are given in Table 6.9b.

(a)

Cluster	$P_{NBFT}^{max(1)}$	$P_{NBFT}^{max(0)}$
Gros	7	3
Grisou	7	3
MareNostrum4	5	3

(b)

Number of processes (P)	$\gamma^1(P)$		
	Grisou	Gros	MareNostrum4
3	1.114	1.084	1.145
4	1.219	1.170	1.290
5	1.283	1.254	1.435
6	1.451	1.339	
7	1.540	1.424	

Table 6.9: Estimated values of $P_{NBFT}^{max(1)}$, $P_{NBFT}^{max(0)}$ and $\gamma^1(P)$ on Grisou, Gros and MareNostrum4 clusters.

Broadcast algorithm	$o^1, L^0, L^1(sec)$		
	Grisou	MareNostrum4	Gros
Linear tree	2.1×10^{-4}, 2.1×10^{-4}, 2.1×10^{-4}	6.8×10^{-4}, 6.7×10^{-4}, 6.8×10^{-4}	8.2×10^{-5}, 8.2×10^{-5}, 8.2×10^{-5}
K-Chain tree	2.2×10^{-4}, 4.1×10^{-5}, 2.2×10^{-4}	6.2×10^{-5}, 1.2×10^{-5}, 6.2×10^{-5}	8.0×10^{-6}, 1.1×10^{-5}, 8.0×10^{-6}
Chain tree	1.2×10^{-5}, 2.0×10^{-5}, 1.2×10^{-5}	5.4×10^{-5}, 5.8×10^{-6}, 5.4×10^{-5}	8.1×10^{-6}, 1.3×10^{-5}, 8.1×10^{-6}
Split-binary tree	7.4×10^{-5}, 3.5×10^{-4}, 7.4×10^{-5}	1.7×10^{-4}, 2.1×10^{-5}, 1.7×10^{-4}	3.2×10^{-5}, 9.3×10^{-6}, 3.2×10^{-5}
Binary tree	3.3×10^{-4}, 3.0×10^{-4}, 3.3×10^{-4}	3.0×10^{-4}, 7.5×10^{-5}, 3.0×10^{-4}	5.6×10^{-5}, 9.7×10^{-6}, 5.6×10^{-5}
Binomial tree	9.7×10^{-5}, 2.1×10^{-4}, 9.7×10^{-5}	7.9×10^{-5}, 7.9×10^{-6}, 7.9×10^{-5}	1.9×10^{-5}, 8.0×10^{-6}, 1.9×10^{-5}

Table 6.10: Estimated values of o^1, L^0 and L^1 on the Grisou, MareNostrum4 and Gros clusters for Open MPI broadcast algorithms.

Then, algorithm-specific τ-Lop model parameters are estimated for each platform and each algorithm, following the method described in Section 5.2.2. In the communication experiments, we use 600 processes on Grisou, 1000 processes on Gros, and 480 processes on MareNostrum4. The message size, m, varies in the range from 16KB to 4MB on all platforms. We use 9 different message sizes for Open MPI broadcast algorithms, $\{m_i\}_{i=1}^{9}$, separated by a constant step in the logarithmic scale, $\log_2 m_{i+1} - \log_2 m_i = 1$. Thus, for each broadcast algorithm, we obtain a system of 9 linear equations with τ-Lop model parameters as unknowns (See Figure 6.5). We use the Huber regressor [125] to find their values from the system. The values of the parameters for each platform can be found in Table 6.10. We can see that the values of model parameters do vary depending on the broadcast algorithm. The results support our original hypothesis that the average execution time of a point-to-point communication will very much depend on the context of the use of point-to-point communications in the algorithm. Therefore, the estimated values capture more than just sheer network characteristics. Despite the fact that the Split-binary tree and Binary tree broadcast algorithms use the same virtual topology, the estimated time

$$\begin{cases} 14.31 \cdot o^1 + 1012.31 \cdot L^0 + 14.31 \cdot L^1 = 0.0103169514 \\ 17.15 \cdot o^1 + 1015.15 \cdot L^0 + 17.15 \cdot L^1 = 0.010932707 \\ 22.85 \cdot o^1 + 1020.85 \cdot L^0 + 22.85 \cdot L^1 = 0.0121768391 \\ 34.24 \cdot o^1 + 1032.24 \cdot L^0 + 34.24 \cdot L^1 = 0.015099681 \\ 57.03 \cdot o^1 + 1055.03 \cdot L^0 + 57.03 \cdot L^1 = 0.0199503694 \\ 102.59 \cdot o^1 + 1100.59 \cdot L^0 + 102.59 \cdot L^1 = 0.0293413861 \\ 193.73 \cdot o^1 + 1191.73 \cdot L^0 + 193.73 \cdot L^1 = 0.0479402791 \\ 376.00 \cdot o^1 + 1374.00 \cdot L^0 + 376.00 \cdot L^1 = 0.0479402791 \\ 740.55 \cdot o^1 + 1738.55 \cdot L^0 + 740.55 \cdot L^1 = 0.1595142696 \end{cases}$$

Figure 6.5: A system of linear equations built in Gros cluster using binomial tree broadcast algorithm where $P = 1000$ and $m \in [16KB, 4MB]$.

of a point-to-point communication is smaller in the context of the Split-binary one. This can be explained by a higher level of parallelism of the Split-binary algorithm, where a significant part of point-to-point communications is performed in parallel by a large number of independent pairs of processes from the left and right subtrees. o^0 has been estimated 0 in all clusters because of the small size, 8K, of the message segment transmitted through a shared memory channel.

6.2.3 Accuracy of Selection of Optimal Collective Algorithms Using The Constructed Analytical Performance Models

The constructed analytical performance models of the Open MPI broadcast algorithms are designed for the use in the MPI_Bcast routines for efficient and accurate runtime selection of the optimal algorithm, depending on the number of processes and the message size. While the efficiency is evident from the low complexity of the analytical formulas derived in Section 4.3, the experimental results on the accuracy are presented in this section.

Figure 6.6, 6.7 and 6.8 show the results of our experiments in Grisou,

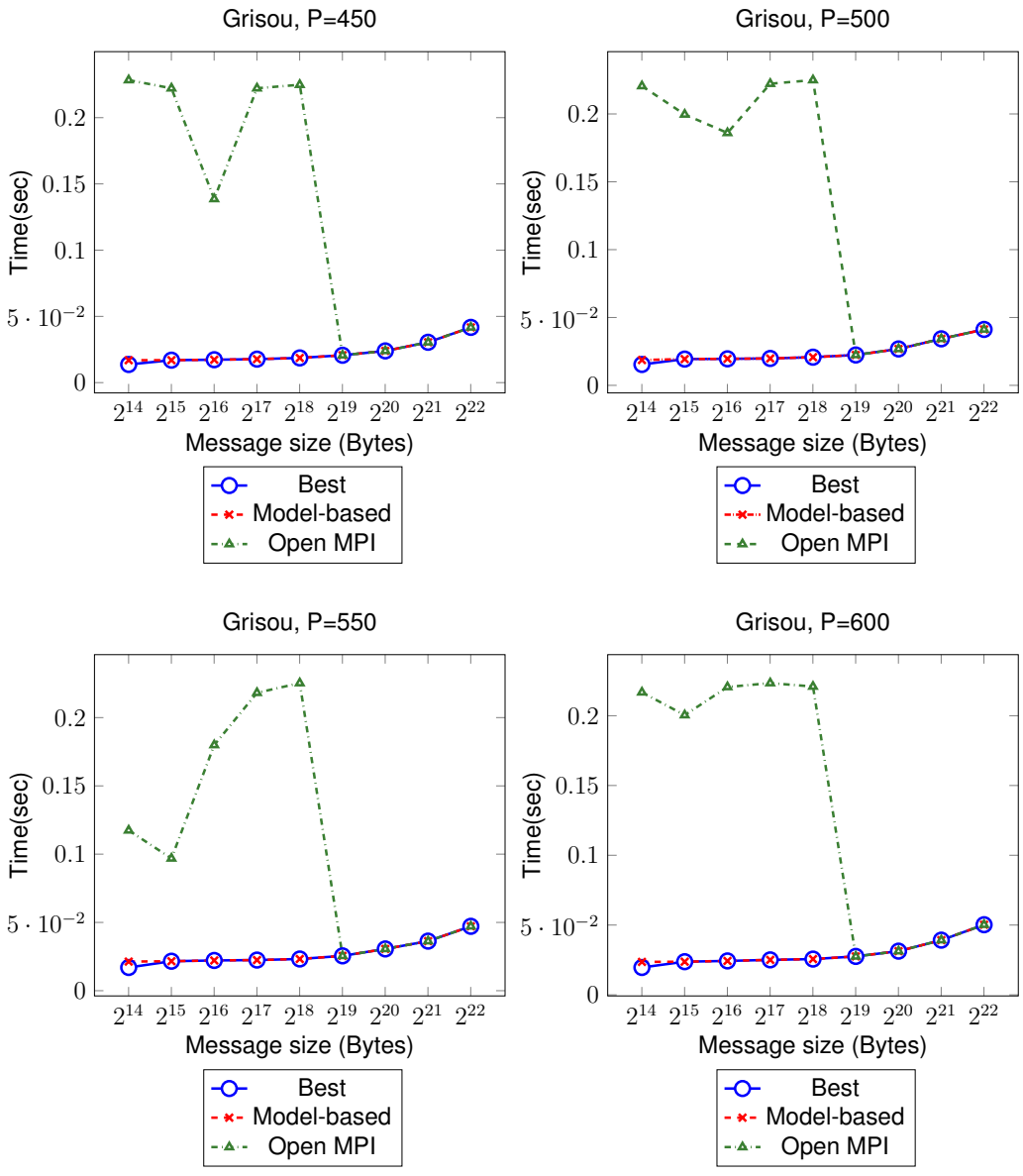


Figure 6.6: Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast on Grisou cluster.

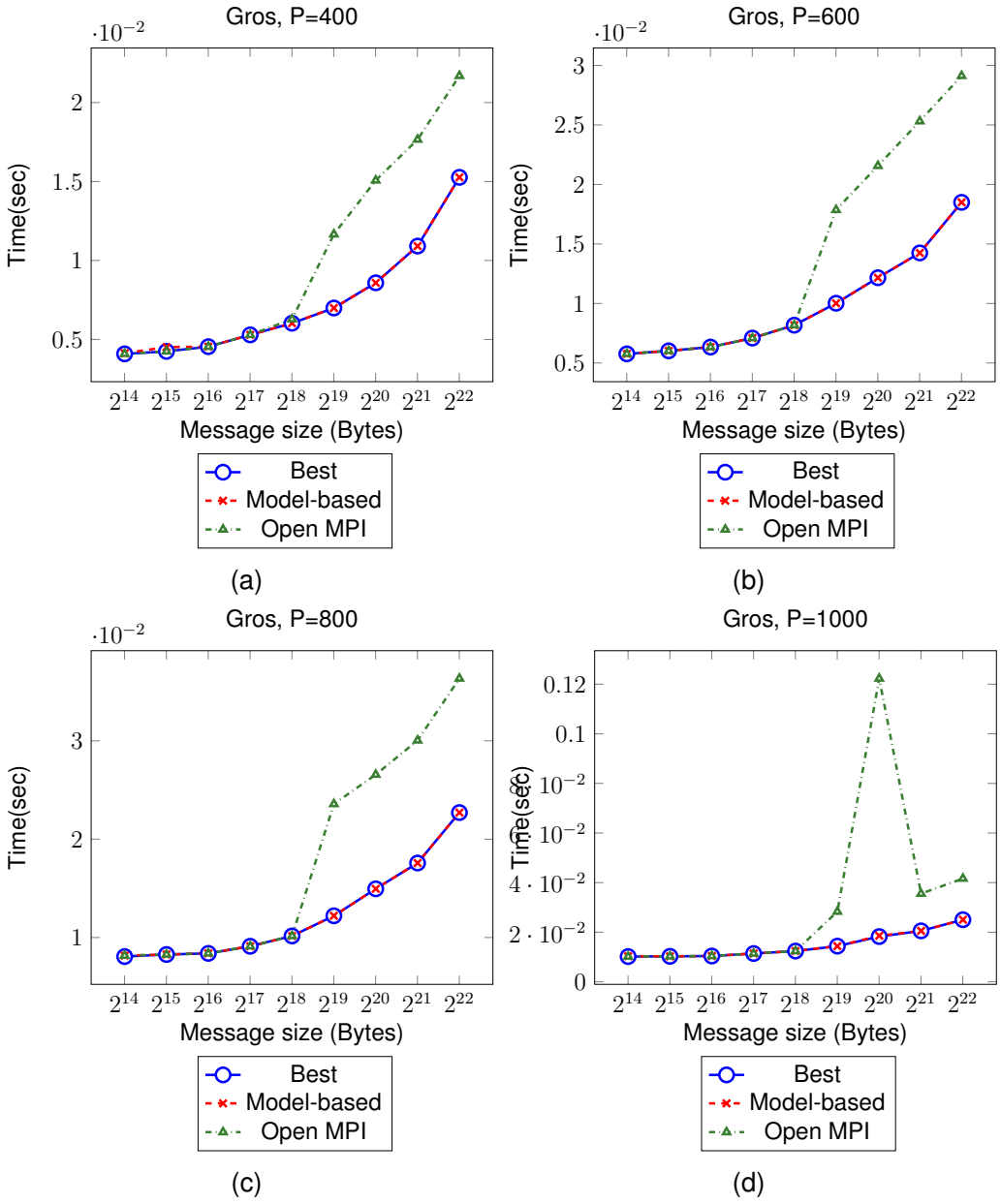


Figure 6.7: Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast on Gros cluster.

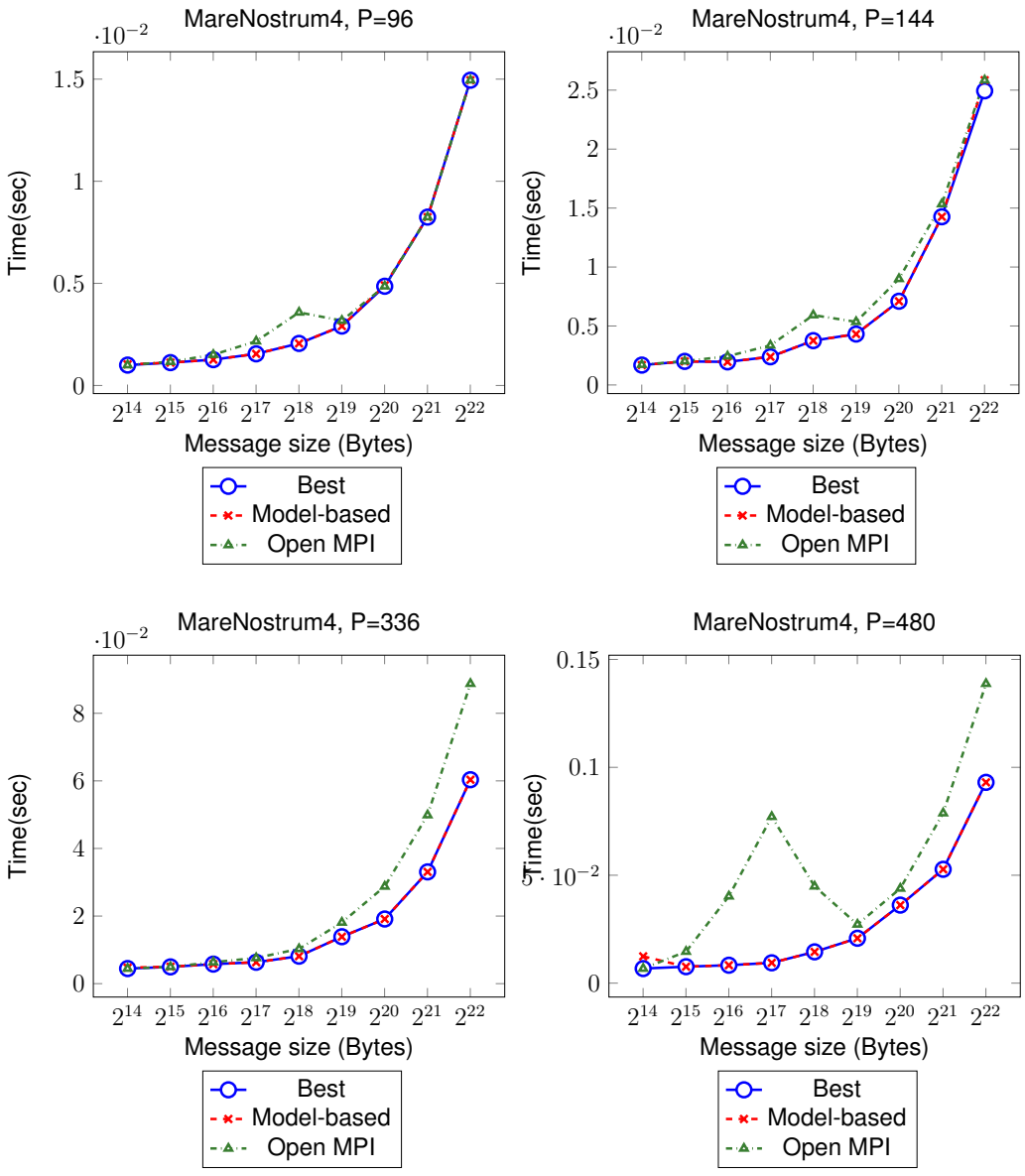


Figure 6.8: Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast on MareNostrum4 cluster.

P=450, Grisou

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
16	$flat$	$chain$ (24)	$split\_binary$ (1568)
32	$chain$	$chain$ (0)	$split\_binary$ (1211)
64	$chain$	$chain$ (0)	$split\_binary$ (702)
128	$chain$	$chain$ (0)	$split\_binary$ (1153)
256	$chain$	$chain$ (0)	$split\_binary$ (1106)
512	$chain$	$chain$ (0)	$chain$ (0)
1024	$chain$	$chain$ (0)	$chain$ (0)
2048	$chain$	$chain$ (0)	$chain$ (0)
4096	$chain$	$chain$ (0)	$chain$ (0)

P=600, Grisou

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
16	$flat$	$chain$ (20)	$split\_binary$ (1006)
32	$chain$	$chain$ (0)	$split\_binary$ (744)
64	$chain$	$chain$ (0)	$split\_binary$ (808)
128	$chain$	$chain$ (0)	$split\_binary$ (792)
256	$chain$	$chain$ (0)	$split\_binary$ (762)
512	$chain$	$chain$ (0)	$chain$ (0)
1024	$chain$	$chain$ (0)	$chain$ (0)
2048	$chain$	$chain$ (0)	$chain$ (0)
4096	$chain$	$chain$ (0)	$chain$ (0)

P=400, Gros

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
16	$split\_binary$	$binary$ (1)	$split\_binary$ (0)
32	$split\_binary$	$binary$ (6)	$split\_binary$ (0)
64	$split\_binary$	$split\_binary$ (0)	$split\_binary$ (0)
128	$split\_binary$	$split\_binary$ (0)	$split\_binary$ (0)
256	$k-chain$	$k-chain$ (0)	$split\_binary$ (4)
512	$k-chain$	$k-chain$ (0)	$chain$ (67)
1024	$k-chain$	$k-chain$ (0)	$chain$ (76)
2048	$k-chain$	$k-chain$ (0)	$chain$ (62)
4096	$k-chain$	$k-chain$ (0)	$chain$ (42)

P=1000, Gros

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
16	$split\_binary$	$binomial$ (1)	$split\_binary$ (0)
32	$split\_binary$	$split\_binary$ (0)	$split\_binary$ (0)
64	$split\_binary$	$split\_binary$ (0)	$split\_binary$ (0)
128	$split\_binary$	$split\_binary$ (0)	$split\_binary$ (0)
256	$split\_binary$	$split\_binary$ (0)	$split\_binary$ (0)
512	$split\_binary$	$split\_binary$ (0)	$chain$ (97)
1024	$split\_binary$	$k-chain$ (2)	$chain$ (570)
2048	$k-chain$	$k-chain$ (0)	$chain$ (73)
4096	$k-chain$	$k-chain$ (0)	$chain$ (66)

P=96, MareNostrum4

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
16	$split\_binary$	$k-chain$ (3)	$split\_binary$ (0)
32	$k-chain$	$k-chain$ (0)	$split\_binary$ (4)
64	$k-chain$	$k-chain$ (0)	$split\_binary$ (19)
128	$k-chain$	$k-chain$ (0)	$split\_binary$ (40)
256	$k-chain$	$k-chain$ (0)	$split\_binary$ (74)
512	$k-chain$	$k-chain$ (0)	$chain$ (9)
1024	$chain$	$k-chain$ (1)	$chain$ (0)
2048	$chain$	$chain$ (0)	$chain$ (0)
4096	$chain$	$chain$ (0)	$chain$ (0)

P=480, MareNostrum4

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
16	$split\_binary$	$k-chain$ (84)	$split\_binary$ (0)
32	$k-chain$	$k-chain$ (0)	$split\_binary$ (92)
64	$k-chain$	$k-chain$ (0)	$split\_binary$ (385)
128	$k-chain$	$k-chain$ (0)	$split\_binary$ (719)
256	$k-chain$	$k-chain$ (0)	$split\_binary$ (209)
512	$k-chain$	$k-chain$ (0)	$chain$ (31)
1024	$k-chain$	$k-chain$ (2)	$chain$ (21)
2048	$k-chain$	$k-chain$ (0)	$chain$ (49)
4096	$k-chain$	$k-chain$ (0)	$chain$ (49)

Table 6.11: Comparison of the model-based and Open MPI selections with the best performing MPI_Bcast algorithm. For each selected algorithm, its performance degradation against the optimal one is given in braces.

Gros and MareNostrum4 clusters for MPI_Bcast respectively. We present results of experiments with $P \in \{450, 500, 550, 600\}$ in Grisou, with $P \in \{400, 600, 800, 1000\}$ in Gros and with $P \in \{96, 144, 336, 480\}$ in MareNostrum4. Again, the message size, m, varies in the range from 16KB to 4MB on all platforms, and we use 9 different sizes, $\{m_i\}_{i=1}^{9}$, separated by a constant step in the logarithmic scale, $\log_2 m_{i+1} - \log_2 m_i = 1$. The plots show the execution time of the broadcast operation as a function of the message size. Each data point on a green line shows the performance of the algorithm selected by the Open MPI decision function for the given number of processes and message size. Each point on a red line shows the performance of the algorithm selected by our decision function, which uses the constructed analytical models. Each point on a blue line shows the performance of the best broadcast algorithm for MPI_Bcast. Figure 6.9 demonstrates the accuracy of the model-based selection compare to the best performance and Open MPI decision function for all message sizes and the number of processes on three clusters.

Table 6.11 presents selections made for MPI_Bcast using the proposed model-based runtime procedure and the Open MPI decision function. For each message size m, the best performing algorithm, the model-based selected algorithm, and the Open MPI selected algorithm are given. For the latter two, the performance degradation in percents in comparison with the best performing algorithm is also given. As we presented in Section 5.2.2, in order to have accurately estimated algorithmic specific model parameters, the execution time of the broadcast algorithm should be dominant in the communication experiments. While the buffered mode is used for small message sizes ($m < E$), the execution time of the flat tree broadcast algorithm with blocking communication cannot be dominant in the experiments. Therefore, the model-based selection degrades the best performance of MPI_Bcast more than 10% just only for the $16KB$ message size on Grisou cluster. The Open MPI selection is near optimal in 40-50% cases and causes significant, up to 1106%, 570% and 719% degradation on Grisou, Gros and MareNostrum4 clusters respectively.

Open MPI decision function uses only three broadcast algorithms (*chain*

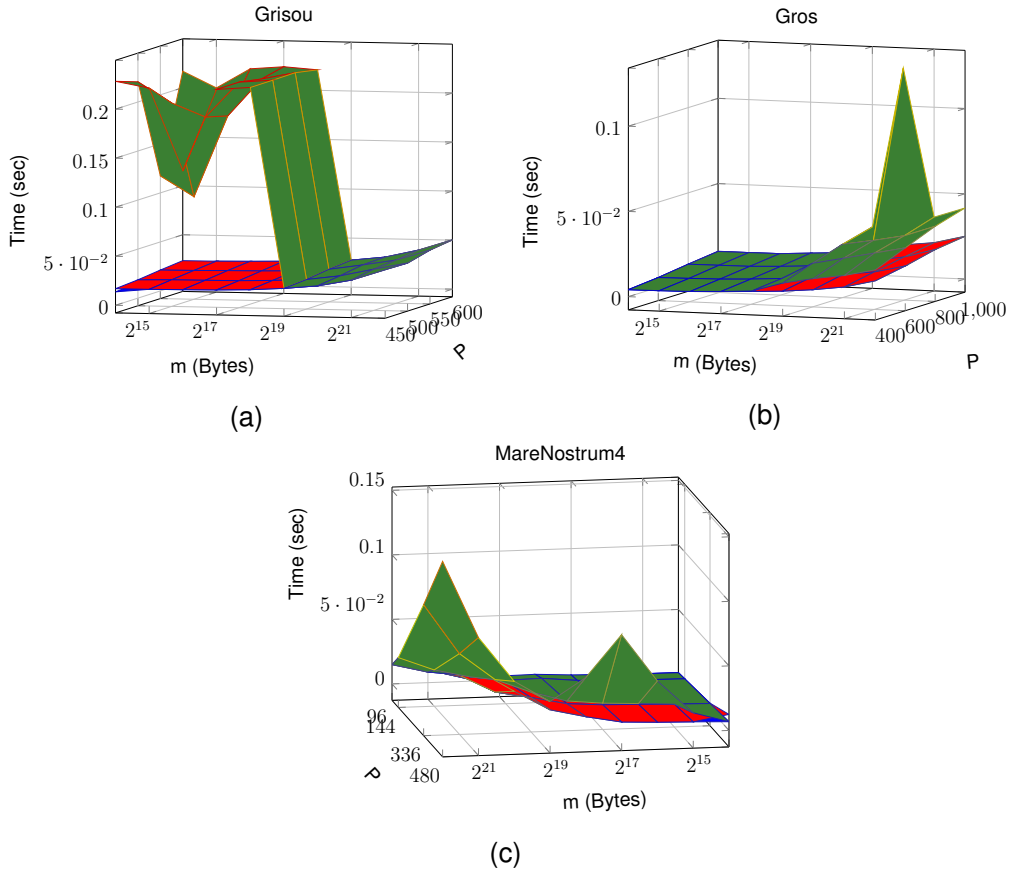


Figure 6.9: (6.9a), (6.9b) and (6.9c) present performance of MPI_Bcast on Grisou, Gros and MareNostrum4 clusters respectively. Blue, red and green surfaces present the best performance of MPI_Bcast, model-based estimation and Open MPI decision function respectively.

tree, split-binary tree and binomial tree) from six implemented broadcast algorithms (see Listing 1.1). For example, K-Chain tree broadcast algorithm, which is never used by Open MPI decision function, outperforms all the algorithms on Gros for $m \in$ {2MB, 4MB}, and on MareNostrum4 for all message sizes except 16KB. In contrast to Open MPI decision function, the model-based selection picks K-Chain broadcast algorithm on Gros and MareNostrum4 platforms with the minimum penalty compare to Open MPI decision function.

Chapter 7

Summary and Conclusion

In this thesis, we proposed a novel model-based approach to automatic selection of optimal algorithms for MPI collective operations, which proved to be both efficient and accurate. The novelty of the approach is two-fold. First, we proposed to derive analytical models of collective algorithms from the code of their implementation rather than from high-level mathematical definitions. Second, we proposed to estimate model parameters separately for each algorithm, using a communication experiment, where the execution of the algorithm itself dominates the execution time of the experiment.

We also developed this approach into a detailed method and applied it to Open MPI 3.1 and its MPI_Bcast and MPI_Gather operations. We developed different analytical models for the same collective algorithm depending on the configuration of the application. While we drove analytical models of collective algorithms from the code of their implementation, we took into account the topology of the communication channels on multi-core clusters as well. We experimentally validated this method on four different clusters and demonstrated its accuracy and efficiency. These results suggest that the proposed approach, based on analytical performance modelling of collective algorithms, can be successful in the solution of the problem of accurate and efficient runtime selection of optimal algorithms for MPI collective operations.

One limitation of the work presented in the thesis is the assumption that the values of model parameters do not depend on the number of processes

but only on the algorithm. When the maximum number of processes is relatively small, this simplification does not affect the predictive accuracy of the constructed models. This has been demonstrated in this thesis. However, for larger platforms, this assumption may negatively affect the accuracy of the models. A natural solution to this problem would be to break the full range of number of processes into a small number of segments and find the values of the model parameters separately for each segment. Unfortunately, our access to a sufficiently large platform was way too limited to conduct a non-speculative research in this direction. We hope to find collaborators with access to such platforms and continue working in this direction in the future.

MPI offers many collective routines each of which has its own semantics. In this thesis, we used only MPI_Bcast (*one-to-all*) and MPI_Gather (*all-to-one*) algorithms to validate our approach. The accurate and efficient selection of the optimal collective algorithms encourages us to apply our model-based approach to the rest of the MPI collectives in order to optimise them. For example, the reduce algorithms (MPI_Reduce and MPI_Allreduce) are the most used collective algorithms in scientific applications. Especially nowadays, we observe growth in the number of MPI based distributed AI frameworks. The most used collective routine in AI frameworks to combine gradient values is MPI_Allreduce. Considering the achieved optimization in MPI_Bcast and MPI_Gather using our approach (Section 6), the approach can be applied to reduce training time in MPI based distributed AI frameworks.

Bibliography

[1] "TOP500 LIST - NOVEMBER 2020." https://www.top500.org/lists/top500/list/2020/11/, 2021. Online; accessed 2 March 2021.

[2] "SUPERCOMPUTER FUGAKU." https://www.top500.org/system/179807/, 2021. Online; accessed 2 March 2021.

[3] "TOP500 LIST - NOVEMBER 2010." https://www.top500.org/lists/top500/list/2010/11/, 2021. Online; accessed 2 March 2021.

[4] "TIANHE-1 - NUDT TH-1 CLUSTER." https://top500.org/system/176546/, 2021. Online; accessed 2 March 2021.

[5] "MPICH - A Portable Implementation of MPI." http://www.mpich.org/, 2021. Online; accessed 2 March 2021.

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (D. Kranzlmüller, P. Kacsuk, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 97–104, Springer Berlin Heidelberg, 2004.

[7] "Open MPI - A High Performance Message Passing Library." https://www.open-mpi.org/, 2021. Online; accessed 2 March 2021.

[8] "A Message-Passing Interface Standard." https://www.mpi-forum.org/, 2021. Online; accessed 2 March 2021.

[9] R. Rabenseifner, “Automatic profiling of MPI applications with hardware performance counters,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 1697 of *Lecture Notes in Computer Science*, pp. 35–42, Springer Berlin Heidelberg, 1999.

[10] J. Worringen, “Pipelining and overlapping for MPI collective operations,” in *28th Annual IEEE International Conference on Local Computer Networks, 2003. LCN ’03. Proceedings.*, pp. 548–557, 2003.

[11] R. Rabenseifner and J. L. Träff, “More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (D. Kranzlmüller, P. Kacsuk, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 36–46, Springer Berlin Heidelberg, 2004.

[12] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby, “Efficient algorithms for all-to-all communications in multiport message-passing systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[13] M. Bernaschi, G. Iannello, and M. Lauria, “Efficient implementation of reduce-scatter in MPI,” *Journal of Systems Architecture*, vol. 49, no. 3, pp. 89 – 108, 2003. Parallel, Distributed and Network-based Processing - selected papers from the 10th Euromicro Workshop.

[14] A. Moody, D. H. Ahn, and B. R. de Supinski, “Exascale Algorithms for Generalized MPI_Comm_split,” in *Recent Advances in the Message Passing Interface* (Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 9–18, Springer Berlin Heidelberg, 2011.

[15] P. Sack and W. Gropp, “A Scalable MPI_Comm_split Algorithm for Exascale Computing,” in *Recent Advances in the Message Passing Interface* (R. Keller, E. Gabriel, M. Resch, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 1–10, Springer Berlin Heidelberg, 2010.

[16] J. L. Traff, "Hierarchical gather/scatter algorithms with graceful degradation," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 80–, 2004.

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

[18] T. Kielmann, H. E. Bal, and S. Gorlatch, "Bandwidth-efficient collective communication for clustered wide area systems," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pp. 492–499, 2000.

[19] "MPICH - Version 3.4.1." `http://www.mpich.org/static/downloads/3.4.1/mpich-3.4.1.tar.gz`, 2021. Online; accessed 2 March 2021.

[20] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra, "Kernel assisted collective intra-node MPI communication among multi-core and many-core CPUs," in *2011 International Conference on Parallel Processing*, pp. 532–541, 2011.

[21] B. Goglin and S. Moreaud, "KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, 2013.

[22] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis," in *2009 International Conference on Parallel Processing*, pp. 462–469, 2009.

[23] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over Infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[24] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges, "Infiniband scalability in Open MPI," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pp. 10 pp.–, 2006.

[25] M. J. Koop, T. Jones, and D. K. Panda, "MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, 2008.

[26] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. Dongarra, and E. Jeannot, "Flexible collective communication tuning architecture applied to Open MPI," in *Euro PVM/MPI*, 2006.

[27] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[28] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[29] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel Computing*, vol. 20, no. 3, pp. 389 – 398, 1994.

[30] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *SIGPLAN Not.*, vol. 28, p. 1–12, July 1993.

[31] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation," tech. rep., USA, 1995.

[32] T. Kielmann, H. E. Bal, and K. Verstoep, "Fast Measurement of LogP Parameters for Message Passing Platforms," in *Parallel and Distributed*

Processing (J. Rolim, ed.), (Berlin, Heidelberg), pp. 1176–1183, Springer Berlin Heidelberg, 2000.

[33] A. Priyam, G. Abhijeeta, A. Rathee, and S. Srivastava, "Comparative analysis of decision tree classification algorithms," *International Journal of current engineering and technology*, vol. 3, no. 2, pp. 334–337, 2013.

[34] V. Vapnik, *The nature of statistical learning theory*. Springer science & business media, 2013.

[35] C. M. Bishop, *Neural Networks for Pattern Recognition*. USA: Oxford University Press, Inc., 1995.

[36] J. Pješivac-Grbović, G. E. Fagg, T. Angskun, G. Bosilca, and J. J. Dongarra, "MPI collective algorithm selection and quadtree encoding," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 40–48, Springer Berlin Heidelberg, 2006.

[37] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 259–269, 2020.

[38] "Grid5000 testbed." `http://www.grid5000.fr`, 2021. Online; accessed 2 March 2021.

[39] "MareNostrum4 supercomputer." `https://www.bsc.es/marenostrum/marenostrum`, 2021. Online; accessed 2 March 2021.

[40] "Shaheen II supercomputer." `https://www.hpc.kaust.edu.sa/content/shaheen-ii`, 2021. Online; accessed 2 March 2021.

[41] "MPI-2: Extensions to the Message-Passing Interface." `https://www.mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf`, 2021. Online; accessed 2 March 2021.

[42] "MPI: A Message-Passing Interface Standard, Version 3.0." `https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2021. Online; accessed 2 March 2021.

[43] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An Evaluation of User-Level Failure Mitigation Support in MPI," in *Recent Advances in the Message Passing Interface* (J. L. Träff, S. Benkner, and J. J. Dongarra, eds.), (Berlin, Heidelberg), pp. 193–203, Springer Berlin Heidelberg, 2012.

[44] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski, "Architecture of LA-MPI, a network-fault-tolerant MPI," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 15–, 2004.

[45] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing," *The International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.

[46] H. Klimach, S. Roller, and C.-D. Munz, "Heterogeneous Parallel Aero-Acoustics Using PACX-MPI", booktitle="High Performance Computing on Vector Systems 2008", year="2009," (Berlin, Heidelberg), pp. 215–222, Springer Berlin Heidelberg.

[47] M. Chaarawi, E. Gabriel, R. Keller, R. L. Graham, G. Bosilca, and J. J. Dongarra, "Ompio: A modular software architecture for mpi i/o," in *Recent Advances in the Message Passing Interface* (Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 81–89, Springer Berlin Heidelberg, 2011.

[48] "Open MPI - Version 3.1." `https://www.open-mpi.org/software/ompi/v3.1/`, 2021. Online; accessed 2 March 2021.

[49] "SUMMIT - IBM POWER SYSTEM AC922." `https://www.top500.org/system/179397/`, 2021. Online; accessed 2 March 2021.

[50] "SIERRA - IBM POWER SYSTEM AC922." https://www.top500.org/system/179398/, 2021. Online; accessed 2 March 2021.

[51] "IBM® Spectrum MPI." https://www.ibm.com/products/spectrum-mpi, 2021. Online; accessed 2 March 2021.

[52] "Intel® MPI Library." https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html, 2021. Online; accessed 2 March 2021.

[53] "HP(Cray) MPI." https://buy.hpe.com/us/en/software/high-performance-computing-ai-software/hpe-message-passing-interface-mpi/p/1010144155, 2021. Online; accessed 2 March 2021.

[54] "Microsoft MPI." https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi, 2021. Online; accessed 2 March 2021.

[55] Numrich, Robert W. and Reid, John, "Co-Array Fortran for Parallel Programming," *SIGPLAN Fortran Forum*, vol. 17, p. 1–31, Aug. 1998.

[56] "International Fortran standards committee." https://wg5-fortran.org/, 2020. Online; accessed 11 December 2020.

[57] Numrich, Robert W. and Reid, John, "Co-Arrays in the next Fortran Standard," *SIGPLAN Fortran Forum*, vol. 24, p. 4–17, Aug. 2005.

[58] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: distributed shared memory programming*, vol. 40. John Wiley & Sons, 2005.

[59] E. Georganas, J. Gonzalez-Dominguez, E. Solomonik, Y. Zheng, J. Tourino, and K. Yelick, "Communication avoiding and overlapping for numerical linear algebra," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[60] R. Preissl, J. Shalf, N. Wichmann, S. Ethier, B. Long, and A. Koniges, "Multithreaded global address space communication techniques for Gyrokinetic fusion applications on ultra-scale platforms," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2011.

[61] P. Husbands and K. Yelick, "Multi-threading and one-sided communication in parallel LU factorization," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–10, 2007.

[62] Charles, Philippe and Grothoff, Christian and Saraswat, Vijay and Donawa, Christopher and Kielstra, Allan and Ebcioglu, Kemal and von Praun, Christoph and Sarkar, Vivek, "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), p. 519–538, Association for Computing Machinery, 2005.

[63] M. Dayarathna, C. Houngkaew, and T. Suzumura, "Introducing ScaleGraph: An X10 Library for Billion Scale Graph Analytics," in *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[64] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[65] P. H. Hargrove and D. Bonachea, "Gasnet-ex performance improvements due to specialization for the cray aries network," in *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pp. 23–33, 2018.

[66] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994.

[67] A. L. Lastovetsky, “mpc: a multi-paradigm programming language for massively parallel computers,” *ACM SIGPLAN Notices*, vol. 31, no. 2, pp. 13–20, 1996.

[68] D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis, “A programming environment for heterogenous distributed memory machines,” in *Proceedings Sixth Heterogeneous Computing Workshop (HCW’97)*, pp. 32–45, IEEE, 1997.

[69] A. Lastovetsky, “Adaptive parallel computing on heterogeneous networks with mpc,” *Parallel computing*, vol. 28, no. 10, pp. 1369–1407, 2002.

[70] A. L. Lastovetsky, *Parallel computing on heterogeneous networks*, vol. 24. John Wiley & Sons, 2003.

[71] A. Lastovetsky and R. Reddy, “HeteroMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers,” *Journal of Parallel and Distributed Computing*, vol. 66, pp. 197–220, 2006.

[72] L. V. Kale and S. Krishnan, “CHARM++: A Portable Concurrent Object Oriented System Based on C++,” in *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA ’93, (New York, NY, USA), p. 91–108, Association for Computing Machinery, 1993.

[73] J. C. Phillips, D. J. Hardy, J. D. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. Hénin, W. Jiang, *et al.*, “Scalable molecular dynamics on CPU and GPU architectures with NAMD,” *The Journal of chemical physics*, vol. 153, no. 4, p. 044130, 2020.

[74] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn, “Massively parallel cosmological simulations with ChaNGa,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12, 2008.

[75] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn, "On optimizing collective communication," in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*, pp. 145–155, 2004.

[76] W. Gropp, L. N. Olson, and P. Samfass, "Modeling MPI communication performance on smp nodes: Is it time to retire the ping pong test," in *Proceedings of the 23rd European MPI Users' Group Meeting*, pp. 41–50, 2016.

[77] A. Lastovetsky, V. Rychkov, and M. O'Flynn, "MPIBlib: Benchmarking MPI Communications for Parallel Computing on Homogeneous and Heterogeneous Clusters," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (A. Lastovetsky, T. Kechadi, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 227–238, Springer Berlin Heidelberg, 2008.

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

[79] F. Ino, N. Fujimoto, and K. Hagihara, "LogGPS: A Parallel Computational Model for Synchronization Analysis," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, (New York, NY, USA), p. 133–142, Association for Computing Machinery, 2001.

[80] W. Chen, J. Zhai, J. Zhang, and W. Zheng, "LogGPO: An accurate communication model for performance prediction of MPI programs," *Science in China Series F: Information Sciences*, vol. 52, no. 10, pp. 1785–1791, 2009.

[81] C. A. Moritz and M. I. Frank, "LoGPG: Modeling network contention in message-passing programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 4, pp. 404–415, 2001.

[82] C. A. Moritz and M. I. Frank, "LoGPC: Modeling Network Contention in Message-Passing Programs," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, p. 254–263, June 1998.

[83] L. Yuan, Y. Zhang, Y. Tang, L. Rao, and X. Sun, "LogGPH: A Parallel Computational Model with Hierarchical Communication Awareness," in *2010 13th IEEE International Conference on Computational Science and Engineering*, pp. 268–274, 2010.

[84] M. I. Frank, A. Agarwal, and M. K. Vernon, "LoPC: Modeling Contention in Parallel Algorithms," in *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '97, (New York, NY, USA), p. 276–287, Association for Computing Machinery, 1997.

[85] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, "LogfP - a Model for Small Messages in InfiniBand," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, (USA), p. 319, IEEE Computer Society, 2006.

[86] K. W. Cameron and R. Ge, "Predicting and evaluating distributed communication performance," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, (USA), p. 43, IEEE Computer Society, 2004.

[87] K. W. Cameron, R. Ge, and X. Sun, "$\log_n p$ and $\log_3 p$: Accurate analytical models of point-to-point communication in distributed systems," *IEEE Transactions on Computers*, vol. 56, no. 3, pp. 314–327, 2007.

[88] J.-A. Rico-Gallego and J.-C. Díaz-Martín, "τ-Lop: Modeling performance of shared memory MPI," *Parallel Computing*, vol. 46, pp. 14 – 31, 2015.

[89] J.-A. Rico-Gallego, J.-C. Díaz-Martín, and A. L. Lastovetsky, "Extending τ-lop to model concurrent MPI communications in multicore clusters," *Future Generation Computer Systems*, vol. 61, pp. 66 – 82, 2016.

[90] Jing Chen, Linbo Zhang, Yunquan Zhang, and Wei Yuan, "Performance evaluation of allgather algorithms on terascale linux cluster with fast ethernet," in *Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)*, pp. 6 pp.–442, 2005.

[91] J. Pjesivac-Grbovic, *Towards automatic and adaptive optimizations of MPI collective operations*. PhD thesis, University of Tennessee - Knoxville, 12 2007.

[92] P. Patarasuk, A. Faraj, and Xin Yuan, "Pipelined broadcast on ethernet switched clusters," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pp. 10 pp.–, 2006.

[93] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI–the Complete Reference: The MPI core*, vol. 1. MIT press, 1998.

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

[95] P. Sanders and J. L. Träff, "Parallel prefix (scan) algorithms for MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 49–57, Springer Berlin Heidelberg, 2006.

[96] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience: Research articles," *Concurr. Comput.: Pract. Exper.*, vol. 19, p. 1749–1783, Sept. 2007.

[97] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. Hofman, "Network performance-aware collective communication for clustered wide-area systems," *Parallel Computing*, vol. 27, no. 11, pp. 1431–1456, 2001. Clusters and computational grids for scientific computing.

[98] L. A. Barchet-Estefanel and G. Mounié, "Fast Tuning of Intra-cluster Collective Communications," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (D. Kranzlmüller, P. Kacsuk, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 28–35, Springer Berlin Heidelberg, 2004.

[99] K. Hasanov, J.-N. Quintin, and A. Lastovetsky, "Topology-oblivious optimization of MPI broadcast algorithms on extreme-scale platforms," *Simulation Modelling Practice and Theory*, vol. 58, pp. 30 – 39, 2015. Special Issue on TECHNIQUES AND APPLICATIONS FOR SUSTAINABLE ULTRASCALE COMPUTING SYSTEMS.

[100] K. Hasanov and A. Lastovetsky, "Hierarchical Redesign of Classic MPI Reduction Algorithms," *J. Supercomput.*, vol. 73, p. 713–725, Feb. 2017.

[101] K. Hasanov, *Hierarchical approach to optimization of MPI collective communication algorithms*. PhD thesis, PhD. thesis, University College Dublin, 2015.

[102] L. A. Barchet-Estefanel and G. Mounié, "Fast Tuning of Intra-cluster Collective Communications," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (D. Kranzlmüller, P. Kacsuk, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 28–35, Springer Berlin Heidelberg, 2004.

[103] L. A. Barchet-Steffenel and G. Mounié, "Total Exchange Performance Modelling Under Network Contention," in *Parallel Processing and Applied Mathematics* (R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski, eds.), (Berlin, Heidelberg), pp. 100–107, Springer Berlin Heidelberg, 2006.

[104] R. A. Van de Geijn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," tech. rep., USA, 1995.

[105] J.-N. Quintin, K. Hasanov, and A. Lastovetsky, "Hierarchical parallel matrix multiplication on large-scale distributed memory platforms,"

in *2013 42nd International Conference on Parallel Processing*, pp. 754–762, IEEE, 2013.

[106] K. Hasanov, J.-N. Quintin, and A. Lastovetsky, "Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms," *The Journal of Supercomputing*, vol. 71, no. 11, pp. 3991–4014, 2015.

[107] A. Lastovetsky, I. . Mkwawa, and M. O'Flynn, "An accurate communication model of a heterogeneous cluster based on a switch-enabled Ethernet network," in *12th International Conference on Parallel and Distributed Systems - (ICPADS'06)*, vol. 2, pp. 6 pp.–, 2006.

[108] A. Lastovetsky and M. O'Flynn, "A performance model of many-to-one collective communications for parallel computing," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, IEEE, 2007.

[109] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa, "LogP performance assessment of fast network interfaces," *IEEE Micro*, vol. 16, no. 1, pp. 35–43, 1996.

[110] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pp. 256–266, 1992.

[111] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGP in theory and practice – An in-depth analysis of modern interconnection networks and benchmarking methods for collective operations," *Simulation Modelling Practice and Theory*, vol. 17, no. 9, pp. 1511 – 1521, 2009. Advances in System Performance Modelling, Analysis and Enhancement.

[112] A. Lastovetsky and V. Rychkov, "Building the communication performance model of heterogeneous clusters based on a switched network," in *2007 IEEE International Conference on Cluster Computing*, pp. 568–575, 2007.

[113] A. Lastovetsky and V. Rychkov, "Accurate and efficient estimation of parameters of heterogeneous communication performance models," *The International Journal of High Performance Computing Applications*, vol. 23, no. 2, pp. 123–139, 2009.

[114] A. Lastovetsky, V. Rychkov, and M. O'Flynn, "Accurate heterogeneous communication models and a software tool for their efficient estimation," *The International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 34–48, 2010.

[115] T. G. Dietterich and E. B. Kong, "Machine learning bias, statistical bias, and statistical variance of decision tree algorithms," tech. rep., Technical report, Department of Computer Science, Oregon State University, 1995.

[116] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "Decision Trees and MPI Collective Algorithm Selection Problem," in *Euro-Par 2007 Parallel Processing* (A.-M. Kermarrec, L. Bougé, and T. Priol, eds.), (Berlin, Heidelberg), pp. 107–117, Springer Berlin Heidelberg, 2007.

[117] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[118] Tin Kam Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, pp. 278–282 vol.1, 1995.

[119] D. A. Freedman, *Statistical models: theory and practice*. cambridge university press, 2009.

[120] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), p. 785–794, Association for Computing Machinery, 2016.

[121] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[122] R. Higdon, *Generalized Additive Models*, pp. 814–815. New York, NY: Springer New York, 2013.

[123] U. Wickramasinghe and A. Lumsdaine, "A survey of methods for collective communication optimization and tuning," *arXiv preprint arXiv:1611.06334*, 2016.

[124] E. Nuriyev and A. Lastovetsky, "Accurate runtime selection of optimal MPI collective algorithms using analytical performance modelling," *arXiv preprint arXiv:2004.11062*, 2020.

[125] P. J. Huber, "Robust estimation of a location parameter," in *Breakthroughs in statistics*, pp. 492–518, Springer, 1992.

[126] G. D. Hutcheson, "Ordinary least-squares regression," *L. Moutinho and GD Hutcheson, The SAGE dictionary of quantitative management research*, pp. 224–228, 2011.

[127] M. G. Akritas, S. A. Murphy, and M. P. Lavalley, "The theil-sen estimator with doubly censored data and applications to astronomy," *Journal of the American Statistical Association*, vol. 90, no. 429, pp. 170–177, 1995.

[128] P. J. Huber, *Robust statistics*, vol. 523. John Wiley & Sons, 2004.

[129] "A tool to select the optimal collective algorithms.." `https://csgitlab.ucd.ie/emin.nuri/mpicollmodelling`, 2021. Online; accessed 2 March 2021.

[130] J. Calcote, *Autotools*. No Starch Press, 2019.

[131] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[132] "OMPI INFO TOOL." `https://www.open-mpi.org/doc/v3.0/man1/ompi_info.1.php`, 2021. Online; accessed 2 March 2021.

[133] R. Garreta and G. Moncecchi, *Learning scikit-learn: Machine learning in Python*. Packt Publishing Ltd, 2013.

[134] "GNU Scientific Library." `https://www.gnu.org/software/gsl/doc/html/`, 2021. Online; accessed 2 March 2021.

Appendix A

A tool to select the optimal collective algorithms.

The selection of the optimal collective algorithms using an analytical performance modelling approach is the main goal of the thesis. We build new analytical formulas to estimate the execution time of the algorithms accurately in Section 4.3 and 5.1. In Chapter 6 we demonstrate experimental validation of the proposed approach. In this chapter, we present the tool [129] implemented for the selection of the optimal collective algorithms at runtime using analytical performance models. Figure A.1 demonstrates the logical view of software architecture.

The tool is designed as a composition of two modules: (1) MPI C code calls the collective algorithms implemented in Open MPI using MPI_T interface and measures the execution time. (2) Python code receives the estimated execution time series, extracts algorithmic specific model parameters using Huber regression and selects the optimal collective algorithm using built models (see Figure A.2).

The first module is designed as a GNU Autotools [130] project, implemented in the C programming language [131]. The estimation of the execution time of the collective algorithms is implemented based on the approach introduced in Chapter 5. As presented in Chapter 1 MPI_T interface allows to control and manage internal variables of MPI

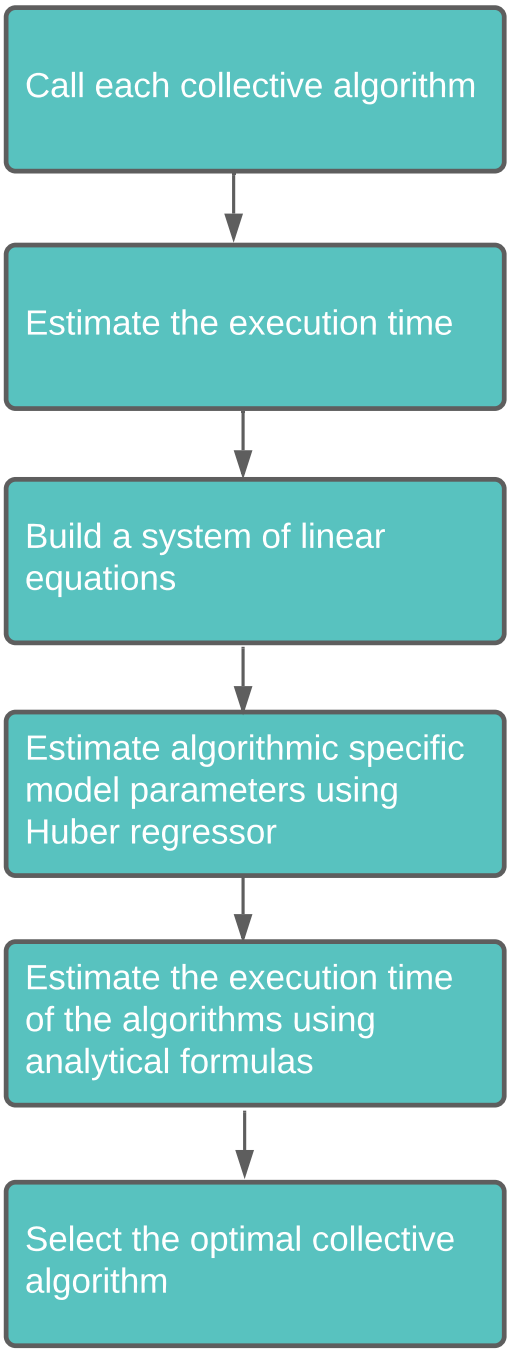


Figure A.1: Logical view of the software architecture.

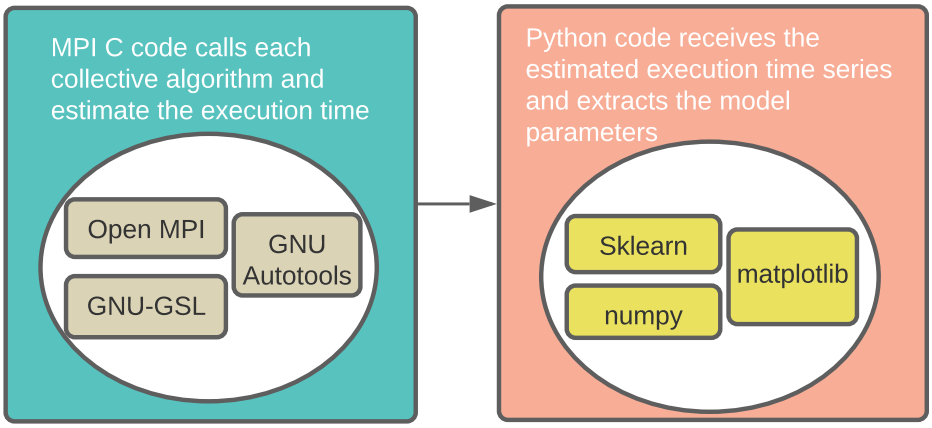


Figure A.2: The architecture of the tool. The arrow shows that the output of the first module is the input of the second module.

implementations. Like other MPI libraries, Open MPI provides a bunch of internal variables to control the performance of collectives. All internal variables used in Open MPI can be extracted using *ompi_info* tool [132]. The internal variables used in this tool are listed below.

- *coll_tuned_bcast_algorithm* - It allows to select the broadcast algorithm from the available list of implemented broadcast algorithms.
- *coll_tuned_bcast_algorithm_segmentsize* - It allows to set new segment size for segmented broadcast algorithms.
- *coll_tuned_gather_algorithm* - It allows to select the gather algorithm from the available list of implemented gather algorithms.

Listing A.1 shows the function controls Open MPI internal variables using MPI_T APIs. A separate set of initialization and finalization routines required for MPI_T interface. Line 2 and 21 initializes and finalizes MPI_T interface respectively. The function alters the value of *variable_name* passed as an argument. For example, in order to set a new broadcast algorithm for MPI_Bcast, we pass *coll_tuned_bcast_algorithm* internal variable as an

argument. Unsuccessful calls to the MPI_T interface are not fatal. Therefore, MPI_T APIs do not impact the execution of subsequent MPI routines.

```
int set_mca_variable(const char *variable_name, int value)
{
   int cidx, nvals, err, provided;
   int val;
   MPI_T_cvar_handle chandle;

   err = MPI_T_init_thread(MPI_THREAD_SINGLE, &provided);
   if (err != MPI_SUCCESS)
      return err;

   err = MPI_T_cvar_get_index(variable_name, &cidx);
   if (err != MPI_SUCCESS)
      return err;

   MPI_T_cvar_handle_alloc(cidx, NULL, &chandle, &nvals);
   err = MPI_T_cvar_write(chandle, &value);
   if (err != MPI_SUCCESS)
      return err;

   MPI_T_cvar_handle_free(&chandle);
   MPI_T_finalize();
   return EXIT_SUCCESS;
}
```

Listing A.1: Altering MCA variables using MPI_T interface.

The second module is designed using Python language and ecosystem of libraries (see Figure A.2). This module receives the estimated performance data of the collective algorithms from the first module. Then, received data is analysed and built the system of linear equations. While Python ecosystem offers many different liner regression implementations, we use Huber loss function [125, 133] for building robust regression.

```
MPI_Barrier(MPI_COMM_WORLD);
while (!stop && reps < precision.max_reps)
{
   MPI_Barrier(MPI_COMM_WORLD);
   if (rank == root)
      time = MPI_Wtime();
   MPI_Bcast(bcast_message, msg_size, MPI_CHAR, root, dump_comm);
   MPI_Barrier(MPI_COMM_WORLD);
   if (rank == root)
   {
      for (j = 1; j < comm_size; j++)
         MPI_Recv(&received_message[j], SEGSIZE, MPI_CHAR, j,
               tag, dump_comm, MPI_STATUSES_IGNORE);
      T[reps] = MPI_Wtime() - time;
      totaltime += T[reps];
   }
   else
      MPI_Send(&to_root, SEGSIZE, MPI_CHAR, root, tag, dump_comm);
   reps++;
   if (reps >= precision.min_reps)
   {
      if (rank == root)
      {
         ci = time_ci(precision.cl, reps, T);
         stop = ci * reps / totaltime < precision.eps;
      }
      MPI_Bcast(&stop, 1, MPI_INT, root, dump_comm);
   }
}
```

Listing A.2: Collective experiments for the broadcast algorithms.

Listing A.2 shows the code estimating the execution time of the experiment designed for the broadcast collective algorithms. The code is implemented using the approach proposed in Chapter 5. As presented in the code we

synchronize the processes by the double MPI_Barrier. The communication experiments are performed using max_reps number of repetitions which is controlled by the user. min_reps defines the lower bound for repetitions the experiments should be performed. We measure the execution time of the experiment at the root where it is started. The GNU Scientific Library [134] is used for statistical analysis. The maximum error in our experiments is equal to $eps = 0.025$. The execution time is estimated within a confidence interval $(1 - eps)$.