Optimizing MPI Collective Operations for Cloud Deployments

by

Zuhair AlSader

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This work is based on a paper published in the 2018 IEEE 11th International Conference on Cloud Computing (IEEE CLOUD'18) co-authored with Mohammed Alfatafta under the supervision of Prof. Samer Al-Kiswany [4]. It was then significantly expanded to become this thesis under the supervision of Prof. Samer Al-Kiswany and Prof. Tim Brecht. In addition, this thesis provides a more thorough and detailed analytic evaluation of the collective operations we support in Chapter 4.

Abstract

Cloud infrastructures are increasingly being adopted as a platform for high performance computing (HPC) science and engineering applications. For HPC applications, the Message-Passing Interface (MPI) is widely-used. Among MPI operations, collective operations are the most I/O intensive and performance critical. However, classical MPI implementations are inefficient on cloud infrastructures because they are implemented at the application layer using network-oblivious communication patterns. These patterns do not differentiate between local or cross-rack communication and hence do not exploit the inherent locality between processes collocated on the same node or the same rack of nodes. Consequently, they can suffer from high network overheads when communicating across racks.

In this thesis, we present *COOL*, a simple and generic approach for Message-Passing Interface (MPI) collective operations. *COOL* enables highly efficient designs for collective operations in the cloud. We then present a system design based on *COOL* that describes how to implement frequently used collective operations. Our design efficiently uses the intra-rack network while significantly reducing cross-rack communication, thus improving application performance and scalability. We use software-defined networking capabilities to build more efficient network paths for I/O intensive collective operations. Our analytic evaluation shows that our design significantly reduces the network overhead across racks. Furthermore, when compared with OpenMPI and MPICH, our design reduces the latency of collective operations by a factor of $\log N$, where N is the total number of processes, decreases the number of exchanged messages by a factor of N and reduces the network load by up to an order of magnitude. These significant improvements come at the cost of a small increase in the computation load on a few processes.

Acknowledgements

First and foremost, I would like to express my sincere gratitude and appreciation for my advisors, Prof. Samer Al-Kiswany and Prof. Tim Brecht, for their academic, professional, mental, and personal guidance. Samer was involved in this project from the beginning and helped me get accustomed to the life in Canada and be prepared for the academic atmosphere. He was always there whenever needed support. Tim joined in after a year and he added so much input and insights to our research explorations and helped make this thesis get to its current shape. Samer and Tim, I am very thankful for you, this thesis would not be possible without your support.

I would like to thank my thesis readers, Prof. Ali Mashtizadeh and Prof. Omid Abari for their valuable feedback and comments.

I am thankful for my friend and collaborator, Mohammed Alfatafta for his contribution to this thesis and his thoughtful comments. I would like to extend my gratitude to the rest of my friends at Waterloo Advanced Systems Lab (WASL), Ahmed, Ashraf, Ibrahim, Harsha and Hatem for providing a supportive, thought-provoking, and joyful environment. I appreciate our interesting discussions and our helpful comments on each other's work.

I would also like to thank my counselors, Cheri and Shelley, and my psychiatric, Scott, for helping me cope with depression. Thank you Cheri for helping at the beginning and helping me realise when I needed help. Thank you Shelley for teaching me the basics of CBT, for talking me out of my negative thought spirals and for directing me to see Scott for treatment. Thank you all for helping me remember who I really was and for helping me give the right size to my problems. I could not have got here without your support.

No words could describe my profound gratitude to my parents, siblings and friends, for their unconditional love and support. Even when the distance between us grew big, you never stopped showering me with your love.

Dedication

To the broken wings that learned to fly again. To those who love to fly. To my parents, siblings, and friends. To those who helped me stay on my grind.

Table of Contents

List of Tables

List of Figures

Abbreviations

COOL *C*loud-*O*ptimized c*OL*lective approach

HPC high-performance computing

MPI Message-Passing Interface

SDN Software-Defined Networks

COOL-B-R *COOL*-Binomial-tree-Recursive-doubling

COOL-P-P *COOL*-Parallel-Parallel

Chapter 1

Introduction

Cloud infrastructures are increasingly being adopted as a platform for high-performance computing (HPC) science and engineering applications [40, 19, 37], with major research organizations embracing the new platform [44] and cloud providers offering clusters targeting HPC applications [5]. For HPC applications, the Message-Passing Interface (MPI) [30, 16] and its classical implementations (e.g., OpenMPI [15] and MPICH [17]) are popular communication middleware. Among MPI operations, collective operations (e.g., broadcast, reduce, scatter and gather) are the most I/O intensive and performance critical [36].

Although classical MPI implementations can be deployed in cloud data centers, they are inefficient [44]. Historically, MPI applications use large-scale supercomputer machines that are customized to support MPI collective operations (e.g., IBM BlueGene [21] and Cray XC40 [11]). Supercomputers use over-provisioned special network topologies (e.g., 3D torus [2], 5D torus [9] and Dragonfly [11]) and use interconnections optimized for I/O-intensive operations. For instance, IBM's BlueGene comes with a network dedicated to collective operations and another one optimized for fast barriers [2]. Classical MPI implementations are optimized to exploit these capabilities. On the other hand, data center networks are drastically different from supercomputers': they do not provide specialized support for collective operations, and they adopt a tree topology [6] that is well provisioned within racks, but is oversubscribed between racks. Oversubscription ratios, the ratio of the

bandwidth within a rack to the bandwidth across racks, of 4 times and up to 10 times are common [6]. Consequently, reducing the communication across racks is key to achieving higher performance and scalability.

The fundamental reason for the poor performance of classical implementations of MPI collective operations in the cloud is that they are implemented at the application layer using network-oblivious communication patterns [15]. For instance, they propagate a broadcast message between processes following a tree pattern, or perform a reduction by communicating using a ring pattern. These patterns do not differentiate between local or cross-rack communications and hence do not exploit the inherent locality between processes collocated on the same node or the same rack of nodes. Consequently, they can suffer from high network overheads when communicating across racks.

In this thesis, we present the *C*loud-*O*ptimized c*OL*lective approach (*COOL*) for collective operations. *COOL* is a simple and generic approach, as it can implement all collective operations. *COOL* divides the group of processes involved in a collective operation into a three-level hierarchy of subgroups: node level, rack level and data center level. All processes collocated on a node form a subgroup with one process being the subgroup leader (or node leader). All node leaders in a rack form a subgroup with one of them acting as a rack leader. Finally, all rack leaders are part of one data-center-wide subgroup. Collective operations are composed of three parts with each part running at one of the three levels. Each level can use the communication pattern that is best suited for that level. This approach provides the implementers of *COOL* with explicit control over the communication performed within a node, within a rack and across racks.

Unlike the designs of classical MPI implementations (e.g., MPICH and OpenMPI), *COOL* is flexible. Communication patterns typically present a trade-off between the number of steps needed, the number of messages sent, and the generated network and process loads. *COOL* allows its implementers to explore this trade-off by combining more than one pattern and allows them to select the best pattern for every level (i.e., node, rack and data center levels) of the data center infrastructure.

To demonstrate the feasibility of our approach, we present a system architecture that embodies *COOL* and provide a detailed design for the most frequently used collective operations. We focus on the following collective operations: MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Gather, MPI_Allgather and MPI_Scatter. We select these operations because they are the most complex, the most I/O intensive and the most frequently used. Characterization studies of MPI applications [36] indicate that these operations consume more than 65% of the total time of all collective operations. The proposed design discovers the network topology and uses this information to create a subgroup per rack and to select rack leaders. Furthermore, the design leverages the Software-Defined Networks (SDN) [33] capabilities of modern switches to build a hierarchy of multicast trees to support MPI_Bcast, MPI_Allreduce and MPI_Allgather.

Our analysis shows that *COOL*-based collective operations significantly reduce the network overhead across racks. Furthermore, we compare our design with OpenMPI and MPICH in terms of the number of steps that each operation takes, the number of exchanged messages and the load imposed on the network and processes. Our evaluation reveals that our *COOL*-based design provides significant performance gains: it completes all operations in three steps, it greatly reduces the number of messages across racks, it reduces the total number of messages by a factor of N in most cases, where N is the number of processes, and it reduces the generated network overhead by up to an order of magnitude. These improvements come at the cost of an increase in the load of leader processes.

The rest of this thesis is organized as follows. In Chapter 2 we present the classical designs of the most frequently used MPI collective operations. We present the *COOL* structure and a system design that embodies it in Chapter 3. We present the analytic evaluation in Chapter 4. We discuss related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Background

In this section, we introduce the Message-Passing Interface and MPI collective operations. Then, we present the frequently used collective operations, their communication patterns and an overview of a typical modern data center architecture.

2.1 Message-Passing Interface

MPI (Message-Passing Interface) is an application programmable interface (API) specification for message-passing, proposed as a standard by a broad community of parallel computing vendors, computer scientists and application developers. It primarily addresses the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through a cooperative mechanism (when two processes communicate, one sends and the other receives) [30, 16]. MPI is a specification, not an implementation or a language and it includes operations expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard. The main goals of establishing the MPI standard are portability and ease of use. For instance, MPI provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases for which they can provide hardware support [21, 11], thereby enhancing scalability. In addition, code written

for one MPI implementation can run on other MPI-compliant platforms with little or no modifications to the source code. Implementations of MPI are widely-available, including OpenMPI [15] and MPICH [17] which are both open source.

2.2 MPI Collective Operations

Collective operations are the most network-intensive operations in MPI; they involve communicating with all the processes in a communication group, typically requiring many steps. Collective communication is defined as communication that involves a group or groups of processes.

2.2.1 Most Frequently Used Collective Operations

The following list enumerates the most frequently used collective operations. Characterization studies of MPI applications [36, 38] indicate that the following operations consume more than 65% of the CPU time that all of MPI collective operations use.

- MPI_Reduce: applies an aggregation operation (e.g., summation, multiplication, maximum or a user-defined function) to data items distributed across a group and makes the result available in one process only.
- MPI_Allreduce: similar to MPI_Reduce, but the final result is available in all processes in a group.
- MPI_Gather: collects data items from all processes in a group and concatenates them into an array in one process (Figure 2.1).
- MPI_Allgather: similar to MPI_Gather, but the final array is available in all processes in a group (Figure 2.1).
- MPI_Bcast: broadcasts a message from one process to all processes in a group (Figure 2.1).

- MPI_Scatter: is the inverse operation of MPI_Gather. It divides an array in one process into chuncks and distributes them between all processes in a group (Figure 2.1).

2.3 Common Communication Patterns

Classical MPI implementations implement collective operations at the application layer using network-oblivious communication patterns. They typically have multiple designs for the same operation, with each using a different communication pattern[15] based on the data size and the number of processes. Nevertheless, all classical patterns are network oblivious; they assume that the communication cost between any two processes is equal, regardless of the network topology. This assumption leads to high inefficiency in the tree-based topologies of modern data centers. Communication patterns use logical addresses (a.k.a. ranks): consecutive integers that identify processes within a group. Typically, these ranks range from 0 to $N-1$, where N is the number of processes in the group.

Unfortunately, only a few of the communication patterns are documented in the literature. To understand the most common communication patterns and their implementation, we dissected the implementation of two popular MPI frameworks: OpenMPI [15] and MPICH [17]. For both of these frameworks, we studied the latest production version and extracted the communication patterns used to implement the collective operations from the source code and documentation. In addition to these patterns, we analyzed patterns proposed in the literature [41, 7]. The following subsections describe the common communication patterns we found.

2.3.1 Recursive Doubling

A *recursive doubling* pattern is used in MPI_Allreduce and MPI_Allgather. It takes $\log N$ steps and in every step i, every process exchanges values with the process 2^{i-1} ranks away (Figure 2.2a). N messages are sent per step. The recursive doubling pattern uses $N \log N$ messages to complete.

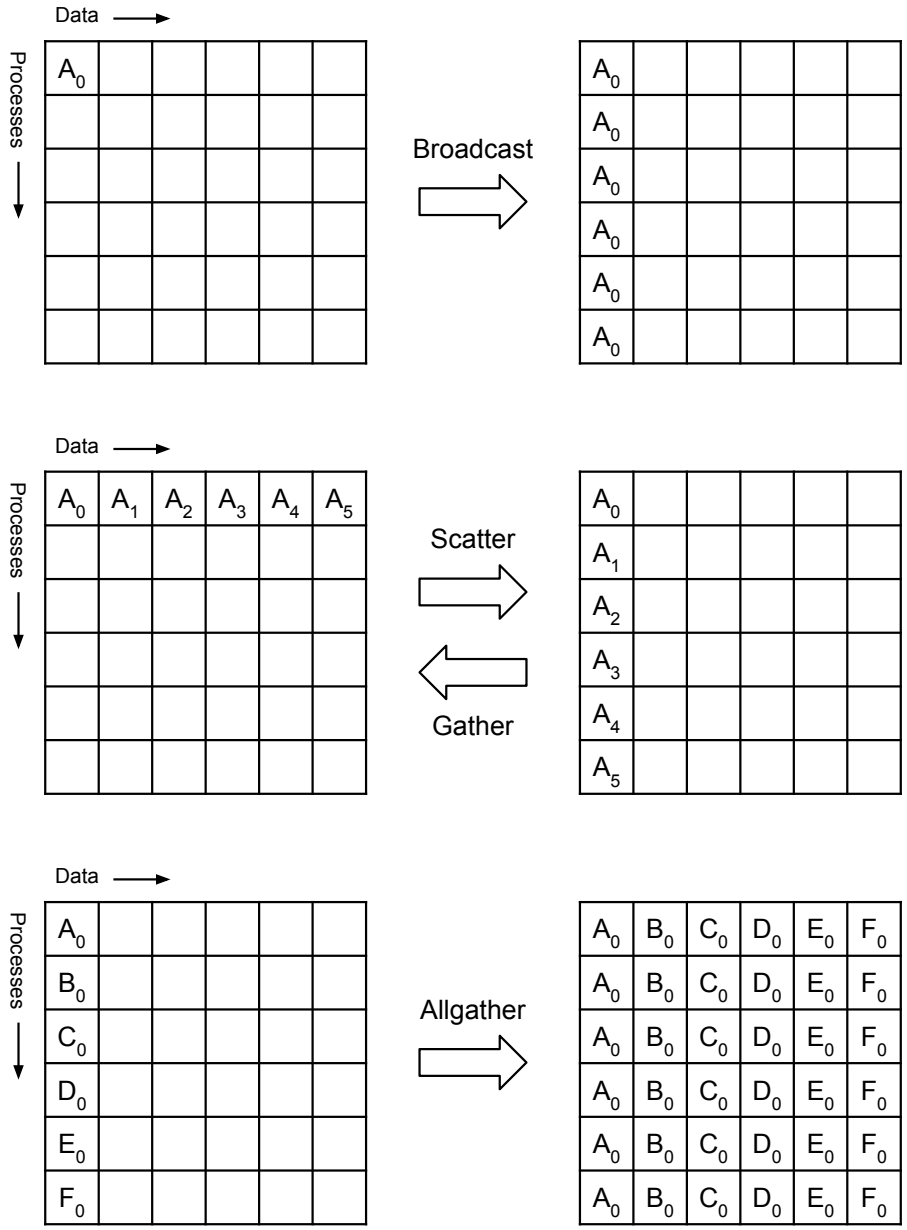
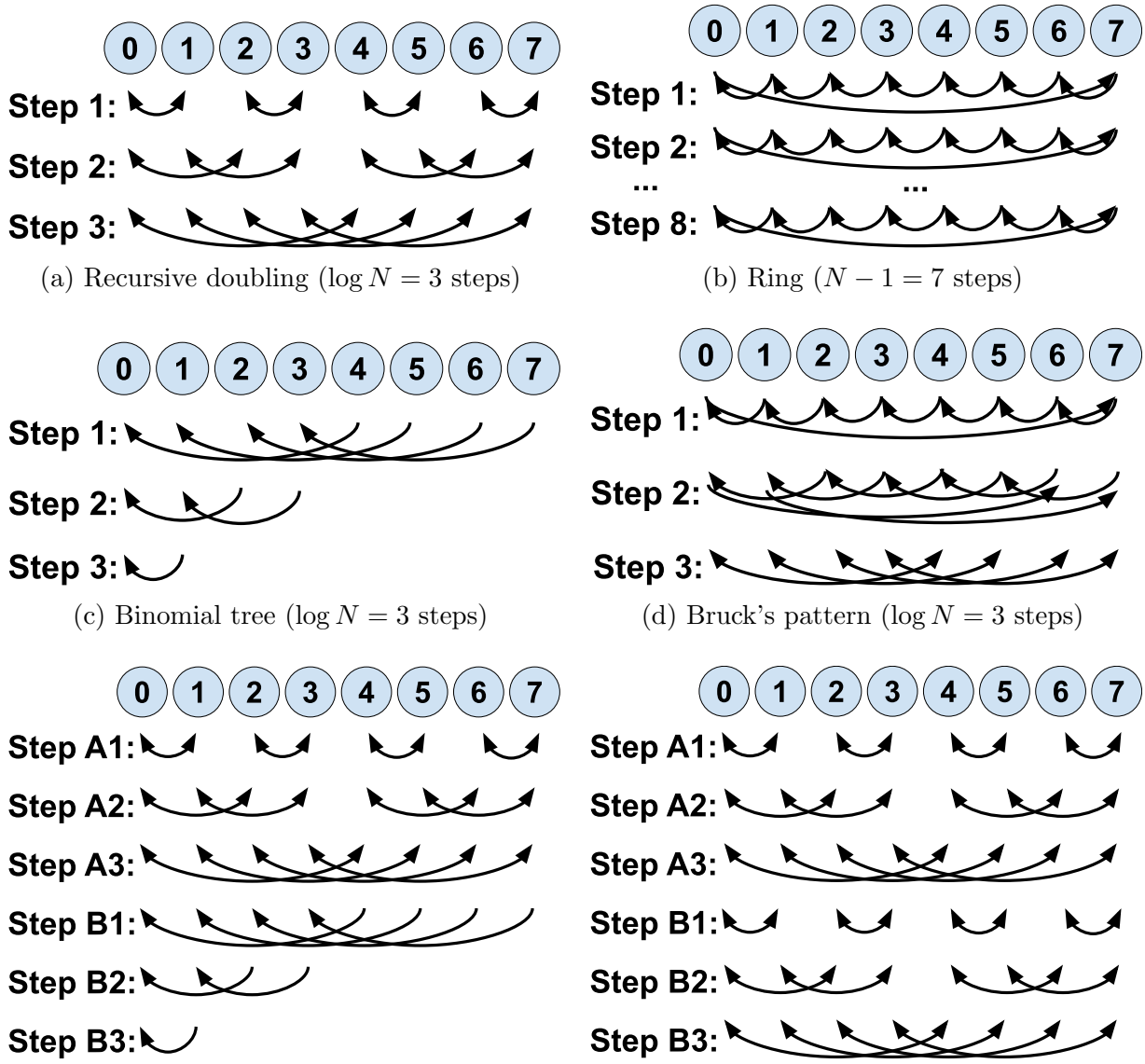


Figure 2.1: Collective operations for a group of six processes. In each case, each row represents data locations in one process. For example, in the broadcast, initially only the first process contains the data item A_0, but after the broadcast all processes contain it.



(a) Recursive doubling ($\log N = 3$ steps)

(b) Ring ($N - 1 = 7$ steps)

(c) Binomial tree ($\log N = 3$ steps)

(d) Bruck's pattern ($\log N = 3$ steps)

(e) Rabenseifner's pattern for MPI_Reduce ($2 \log N = 6$ steps)

(f) Rabenseifner's pattern for MPI_Allreduce ($2 \log N = 6$ steps)

Figure 2.2: Collective communication patterns for a group of $N = 8$ processes. Circles represent processes, numbers indicate the processes' ranks, and arrows represent the direction of message communication. Double arrows indicate that the two processes exchange messages in both directions.

2.3.2 Ring

A *ring* pattern is used in MPI_Allreduce and MPI_Allgather. The ring pattern organizes the processes in a ring. It takes $N-1$ steps and in each step, every process receives a message from its predecessor in the ring and sends a message to its successor (Figure 2.2b). The ring pattern requires $N\,(N-1)$ messages to complete.

2.3.3 Binomial Tree

A *binomial tree* pattern is used in MPI_Bcast, MPI_Gather, MPI_Scatter and MPI_Reduce. It takes $\log N$ steps to complete. For instance, in every step of MPI_Gather (Figure 2.2c), processes are divided into two halves; one half sends the data it has gathered so far to the other half. The receiving half repeats this procedure until a single process is left. That process will have all data items from all processes. Similarly, MPI_Scatter runs the same steps in reverse order: in every step i, each process that has data divides its data in half, keeps one half and sends the other half to a process 2^{i-1} ranks away until all processes have a chunk of data. The binomial tree pattern requires $N-1$ messages to complete.

2.3.4 Rabenseifner's Pattern

Rabenseifner's pattern [41] is used in MPI_Reduce and MPI_Allreduce. Rabenseifner's pattern uses two phases combining two patterns. In the first phase of both operations, it uses the recursive halving pattern (analogous to the recursive doubling pattern) to perform *reduce-scatter*, where the data array is divided to chunks and each chunk is reduced to a different node (Figures 2.2e and 2.2f, steps A1-A3). In the second phase, it uses the binomial tree pattern to gather the reduced chunks in one process for MPI_Reduce (Figure 2.2e, steps B1-B3), or the recursive doubling pattern to gather the reduced chunks in all processes for MPI_Allreduce (Figure 2.2f, steps B1-B3). Rabenseifner's pattern takes $2\log N$ steps to complete, and it exchanges $N + N\log N$ and $2N\log N$ messages for MPI_Reduce and MPI_Allreduce, respectively.

2.3.5 Bruck's Pattern

Bruck's pattern [7] is used in MPI_Allgather. It takes $\log N$ steps, and in every step i, each process p sends its data and all of the data it has received so far to the process with rank $(p - 2^{i-1}) \mod N$ (Figure 2.2d). Since N messages are sent in every step, this pattern sends $N \log N$ messages in total.

2.4 Software-Defined Networks

The Software-Defined Networks (SDN) architecture divides the network into two planes: data and control. The data plane is a traffic forwarding plane that uses the information in the switch forwarding tables to forward messages. The control plane is an external software-based logically-centralized component that controls one or more switches by altering the entries in each switch's forwarding tables. The communication API between the controller and the switches is based on the widely adopted OpenFlow standard [33]. The OpenFlow standard [29] facilitates external control of a single-switch forwarding table. It allows inserting or deleting forwarding rules. Each forwarding entry includes a matching rule and an action list. If a packet matches a rule, the actions in the actions list are performed in order on the packet. OpenFlow has a rich set of matching rules including wild cards for matching IP and MAC addresses, protocol or port numbers. The actions include packet forwarding to a specific switch port, dropping the packet, sending the packet to the controller, or modifying the packet. The possible modifications include changing the source/destination MAC/IP addresses. To avoid the need for switches to contact the controller on every packet, forwarding rules are stored on switches and have an expiry period that is set by the controller. Controllers can update, delete, or extend the validity of the existing rules at any time. These capabilities enable fine-grained control of network operations and facilitate application-optimized traffic engineering.

2.5 Target Deployment

Modern data centers adopt a tree-based topology[6][10], in which nodes are organized in racks (e.g., each rack has 48 nodes) and each rack is connected to the other racks in the data center through a Top-of-Rack (ToR) switch. ToRs are connected via two-tier switching fabric of aggregation and core switches. The inter-rack fabric is typically oversubscribed (i.e., the bandwidth across racks is a fraction of the bandwidth available within a rack). Oversubscription ratios of 4 to 10 times are common [6]. Consequently, increasing communication locality within racks is key to achieving higher performance and scalability.

Furthermore, many modern data centers adopt SDN-capable switches. This new networking paradigm facilitates the external control of network operations. The OpenFlow standard API [33] provides per-packet control of network operations. Developers can use this capability to build network-efficient multicasting trees for I/O intensive broadcast operations.

Large-scale science applications utilize hundreds to thousands of nodes spanning tens of racks. Unfortunately, classical collective implementations have not been designed with data center network communication in mind, as they do not differentiate between communication within a rack or across racks, and they do not exploit SDN capabilities or information about the network topology to optimize the data paths for collective operations.

COOL enables collective operation designs that better fit the data center infrastructure than classical collective operation designs. In particular, it enables the utilization of the access locality between processes collocated on a node, or on nodes on the same rack and enables the exploitation of SDN capabilities to build efficient network paths for multicast-based data transfers within and across racks.

Chapter 3

System Design

In this section, we first introduce the *COOL* approach, then we present a communication pattern that better fits *COOL*'s small subgroups, as well as a system architecture that embodies *COOL*. Finally, we discuss the design of the most frequently used collective operations.

3.1 COOL

The *COOL* approach is a hierarchical approach to perform collective operations. *COOL* divides the communication group into a set of *subgroups* (Figure 3.1). Each subgroup has a leader process. All processes collocated on a node form a subgroup with one process being the subgroup leader (or node leader). All node leaders in a rack form a subgroup with one of them acting as the subgroup leader (or rack leader). Finally, all rack leaders are part of one data-center-wide subgroup. Subgroups are small, consisting of a few tens of processes. During a collective operation, a process can exchange messages with only the processes in its subgroup. Only a subgroup leader can exchange messages with other subgroup leaders at its level.

Typically, a *COOL* collective operation proceeds in *phases*. Communication within a phase is constrained to be between processes in the same subgroup at the same level. A

phase may involve one or more steps depending on the communication pattern used in the phase. The order of these phases depends on the collective operation. First, part of the collective operation is performed in parallel in all node-level subgroups. Second, the node leaders perform part of the operation per rack. Third, rack leaders complete the operation across racks. Finally, the result is propagated down the hierarchy to a specific process or to all processes. *COOL* does not dictate which communication pattern should be used within a subgroup; an implementer of *COOL* can choose different communication patterns within and across subgroups. This flexibility allows for the selection of the best pattern for each subgroup. For instance, an implementer of *COOL* may choose, for the rack-level subgroup, a pattern that completes in a few steps but imposes a high network overhead and may choose a more network-conscious pattern for the cross-rack level even if it slightly increases the number of steps.

As an example, consider the MPI_Bcast operation. For simplicity, assume there is one process per node. In a *COOL* MPI_Bcast, the source process of the broadcast message sends the message to its rack leader (Phase 1 in Figure 3.1). Then, the leader multicasts the message to the other rack leaders (Phase 2). Finally, all leaders multicast the message to the processes in their racks (Phase 3). Different communication patterns can be employed to perform Phase 2 or 3.

The main advantage of *COOL* is that it provides explicit control of the communication within and across racks and enables optimizing the communication at every level of the operation. This approach facilitates tailoring communication patterns to minimize the communication between racks. Implementations of the *COOL* approach must be compliant with the MPI specification, so code written for any other MPI implementations can run on a *COOL* implementation with little or no modifications to the source code. Implementers of *COOL* are free to pre-define the patterns used within and across subgroups for each collective operation, select these patterns dynamically or give the user an extra API to configure which patterns are used.

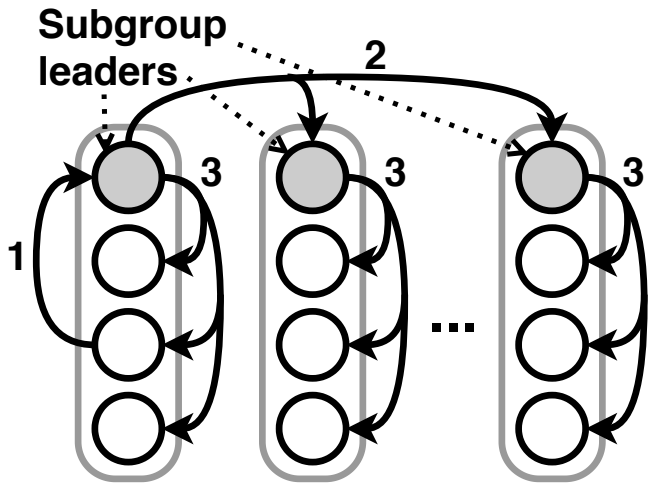


Figure 3.1: *COOL* with the parallel pattern. Circles represent processes, numbers indicate the processes' ranks and arrows represent the direction of communication. Numbers on arrows represent the three steps of a *COOL-based* MPI_Bcast.

3.2 Parallel Pattern

The *COOL* approach divides a large communication group into a set of smaller subgroups, allowing for different communication patterns in each of these small subgroups. Since subgroups in *COOL* are small, we present the *parallel* communication pattern, a simple pattern that efficiently implements all collective operations for small groups. In the *parallel* pattern, all processes exchange messages with a single process that does all of the necessary computation. We refer to this process as the *root* process. For instance, in a *parallel* MPI_Reduce, all processes send their data to the root process, which performs the reduction operation. Similarly, in a *parallel* MPI_Bcast the root process sends a message to all processes in its group. The *parallel* pattern completes any collective operation in one or two steps, but it does not scale to large communication groups.

The *parallel* pattern can be efficiently implemented in small groups of a few tens of processes, as is the case in *COOL* subgroups. For processes collocated on the same node, the *parallel* pattern can be efficiently implemented using shared memory. For operations that involve sending the same message to multiple recipients the *parallel* pattern can exploit

SDN capabilities to construct network-efficient multicast trees for each subgroup when the communication group is created, as explained in Section 2.5.

3.3 COOL Collective Operations

COOL is generic; it can be used to optimize all collective operations for cloud deployments. The following subsections present a *COOL* design for each of the frequently used collective operations discussed in Section 2.2.1. Although *COOL* can employ various communication patterns at different levels, for simplicity, we present a design using the *parallel* pattern at all levels. We also present a design using the binomial tree pattern within racks and the recursive doubling pattern across racks to demonstrate COOL's flexibility. We omit the discussion about the communication between processes collocated on the same node, as it can be efficiently implemented with the *parallel* pattern by using shared memory, and it has a relatively negligible impact on performance. Hence, we assume a single process per node.

Since we are using the parallel pattern or a single point to point message in each phase of the following designs, we can see that all operations complete in only three steps and use the network across racks only in step 2. The next chapter extends our analysis and compares our design with OpenMPI and MPICH implementations.

In the descriptions below, the *root process* refers to the source process sending a broadcast message in MPI_Bcast, or the destination process in MPI_Reduce and MPI_Gather. The *root leader* refers to the leader of the rack that contains the operation's root process. We also present the number of messages exchanged for each operation. The system has N nodes, with each running a single process. The nodes are organized in r racks, so r of them are rack leaders.

3.3.1 MPI_Reduce

(1) Each process sends its value to its rack leader (using $N - r$ messages). Each leader reduces the values of its subgroup to an intermediate value. (2) All leaders send their intermediate values to the root leader (using $r - 1$ messages). The root leader reduces all the values it receives to a single final value. Finally, (3) the root leader forwards the final value to the root process (in a single message). This approach requires a total of N messages.

3.3.2 MPI_Allreduce

(1) Each process sends its value to its rack leader ($N-r$ messages). Each leader reduces the values of its subgroup to an intermediate value. (2) All leaders multicast their intermediate values to all other leaders (r multicast messages). Every leader reduces all of the values it receives to a single final value. Finally, (3) every leader multicasts the final value to its rack subgroup (r multicast messages). This approach requires a total of $N - r$ messages and $2r$ multicast messages.

3.3.3 MPI_Gather

(1) Each process sends its value to its rack leader ($N - r$ messages). Each leader concatenates the values of its subgroup into a subarray. (2) All leaders send their subarrays to the root leader ($r - 1$ messages). The root leader concatenates all the subarrays it receives into a single final array. Finally, (3) the root leader forwards the final array to the root process (in a single message). This approach requires N messages in total.

3.3.4 MPI_Allgather

(1) Each process sends its value to its rack leader ($N - r$ messages). Each leader concatenates all the values of its subgroup into a subarray. (2) All leaders multicast their subarrays

to all other leaders (r multicast messages). Every leader concatenates all of the subarrays it receives into a single final array. Finally, (3) every leader multicasts the final array to its rack subgroup (r multicast messages). This approach requires the total of $N - r$ messages and $2r$ multicast messages.

3.3.5 MPI_Bcast

(1) The root process sends its value to the root leader (one message). (2) The root leader multicasts the value to all other rack leaders (one multicast message). Finally, (3) every rack leader multicasts the value to its subgroup (r multicast messages). This approach requires one message and $r + 1$ multicast messages.

3.3.6 MPI_Scatter

(1) The root process sends its array to the root leader (one message). (2) The root leader divides the array into r subarrays, keeps one subarray and send a subarray to every leader ($r - 1$ messages). Finally, (3) every leader, including the root leader, sends the individual data items from its subarray to every process in its subgroup ($N - r$ messages). This approach requires a total of N messages.

3.3.7 COOL-Binomial-Tree-Recursive-Doubling for MPI_Allreduce

COOL-Binomial-tree-Recursive-doubling (*COOL-B-R*) uses the binomial tree pattern within a rack and the recursive doubling pattern between racks. In MPI_Allreduce it works as follows. (1) Each process sends its value to its rack leader using the binomial tree communication pattern. In each step, processes in each rack are divided into two halves; one half applies the reduce operation to its local data and the data it has gathered so far and sends the result to the other half. The receiving half repeats this procedure until a single process (the rack leader) is left. Each rack leader will then have the reduced result from

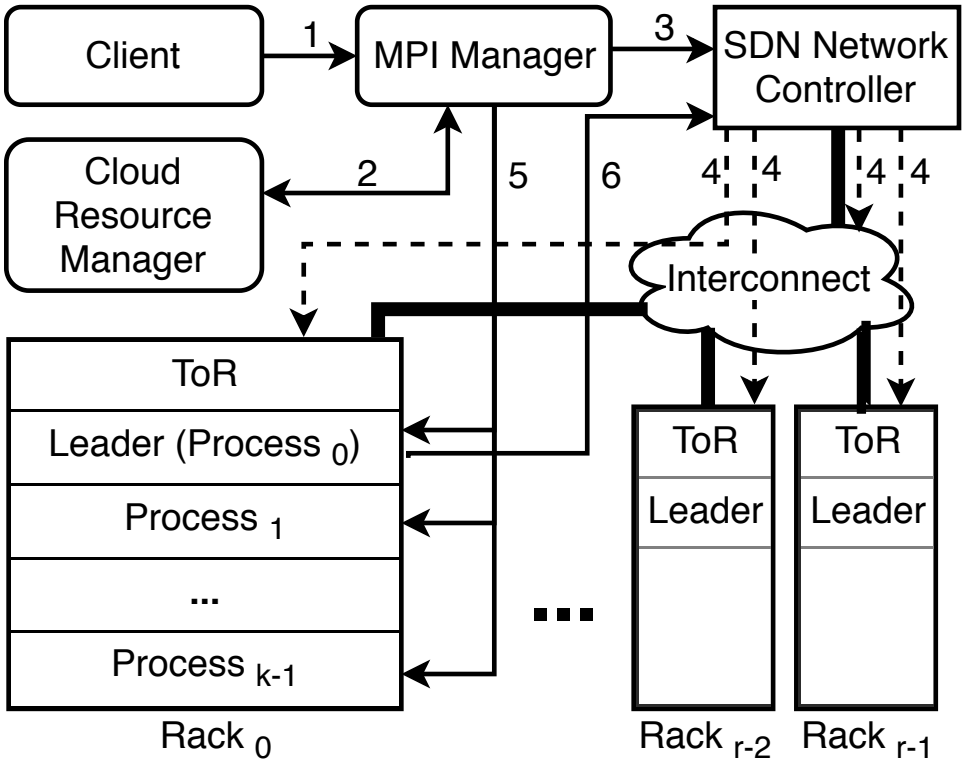


Figure 3.2: System Architecture. Bold lines represent network connections. Solid arrows represent communication messages. Dashed arrows represent OpenFlow control messages.

all processes in its rack. This phase requires $N - r$ messages. (2) Each rack leader sends its reduced result to all other rack leaders using the recursive doubling communication pattern. In every step i (starting from 1), each pair of rack leaders with a distance of 2^{i-1} between their rack ids combine their data and the data they have received so far and exchange that data with each other. They then apply the reduce operation to that data. After $\log N$ steps all rack leaders will have the reduced result from all the other processes. This phase requires $r \log r$ messages. Finally, (3) every rack leader multicasts the final reduced result to the other $k - 1$ processes in its rack (r multicast message each to $k - 1$ recipients). Therefore, the total number of messages exchanged in *COOL-B-R* is $N - r + r \log r$ messages and r multicast messages.

3.4 System Architecture

The goal of this section is to demonstrate the feasibility of building an MPI implementation that embodies *COOL*. Our system architecture has four components (Figure 3.2): an MPI manager, rack leaders, an SDN network controller and a *COOL* library. The MPI manager controls the MPI application lifecycle from allocating resources from the cloud resource manager, to bootstrapping the MPI processes and terminating the application. All processes are divided into subgroups at the node, rack and data center levels during the bootstrap process. The SDN network controller is an OpenFlow-based controller that manages all the switches in the deployment. The SDN network controller installs packet-forwarding rules to create a hierarchy of multicast trees for each communication group and its constituent subgroups. The *COOL* library implements all collective operations as discussed in the previous section.

3.4.1 Bootstrap Process

As depicted in Figure 3.2, when a client starts a new job, it sends the job parameters to the MPI manager (step 1 in Figure 3.2). The manager allocates a number of nodes for that client (2). Then, the SDN network controller configures the network for the MPI job (3). The SDN network controller first discovers the network topology connecting the allocated nodes using the Link Layer Discovery Protocol (LLDP) protocol[20]. The discovery step identifies the racks, assigns them serial identification (ID) numbers and discovers which nodes are in each rack (4). Then, the SDN network controller divides the group of MPI processes into per-node and per-rack subgroups. Additionally, for each subgroup, it selects as the leader the process with the smallest rank in the subgroup. Finally, (5) the manager runs the MPI processes on the allocated nodes. The manager informs every MPI process of its node and its rack leaders, as well as the rank and rack IDs of all the other processes and all of the other subgroup leaders.

3.4.2 Group Creation

When an MPI application creates a new communication group (step 6 in Figure 3.2) the leader of the first rack (i.e., rack 0) informs the SDN network controller of the new group. As in the bootstrap process, the SDN network controller divides the processes into subgroups, with each one of them containing all the processes collocated in the same rack and chooses a leader for each subgroup. The SDN network controller also creates, using OpenFlow, a multicast tree in every rack and creates a multicast tree for rack leaders across racks.

Chapter 4

Evaluation

In this chapter, we analytically compare collective operations in *COOL* with classical implementations (OpenMPI and MPICH). We focus our evaluation on four metrics: the number of steps an operation requires, the number of messages exchanged in total and across racks, the generated network load in total and across racks and the maximum generated load on processes. We define these metrics in Section 4.2 and use them to analyze each of the patterns used to classically implement MPI collective operations as well as *COOL*'s implementations (Section 4.3). Section 4.4 summarizes the analysis for each metric and compares classical implementations with *COOL*.

4.1 Assumptions

To simplify our analysis, we assume that all *COOL* subgroups are equal in size and that the total number of nodes and the number of nodes in every rack is a power of two. We assume that every node runs a single MPI process, as communication between collocated processes on the same node adds a relatively negligible overhead.

The performance of the classical implementations of collective operations is affected by how the processes are ranked, which affects the communication order. In our evaluation, we select the best ranking that minimizes the number of cross-rack messages for each

operation-pattern combination. We note that it is infeasible for classical MPI implementations to use these best rankings because a communication group typically uses multiple collective operations, each of which has a different best ranking, yet processes in a group have fixed ranks. Collective operations have different best rankings even if they use the same pattern. For instance, the best ranking for the binomial tree pattern in MPI_Bcast is different than the best ranking used for the binomial tree in MPI_Reduce.

For *COOL*, we analyze the *COOL*-Parallel-Parallel (*COOL-P-P*) pattern that uses the *parallel* pattern within and across racks. Section 4.5 discusses a *COOL*-based design that uses the binomial tree and the recursive doubling patterns, named *COOL*-Binomial-tree-Recursive-doubling (*COOL-B-R*).

Table 4.1: Analytical model parameters.

Symbol	**Description**
N	Total number of processes. $N = k \cdot r$
r	Number of racks
k	Number of nodes within each rack
xM_y	x multicast messages to y recipients.

4.2 Evaluation Metrics

In this section, we define the evaluation metrics we use in our analysis.

Operation latency. We analyze operation latency by examining the number of steps each pattern takes to complete. Each step includes a parallel communication phase, which consists of a concurrent exchange of data with one or more processes and a computation phase for data copying, segmentation or an aggregation operation (as in the case of MPI_Reduce and MPI_Allreduce).

Exchanged messages. We analyze the number of messages exchanged in total between all processes and the number of transmissions across racks.

Network load. We compare the total network load generated by the different communication patterns required for each operation. The total network load metric aggregates the load generated (i.e., number of messages) on every link in the topology. In our analysis, we assume a simple network topology with only one core switch connecting all racks (Figure 4.1). For instance, a message from one node to another node in the same rack uses two links: one link to the ToR switch and one from the ToR switch to the destination node (shown with dashed arrows in Figure 4.1). A message sent across racks traverses four links: one link to the source ToR switch, one to the core switch, one to the destination ToR switch, and one to the destination node (shown with solid arrows in Figure 4.1). Similarly, a multicast message within a rack ($1M_{k-1}$) uses k links: one to the ToR switch and $k-1$ to all other nodes in the rack. A multicast message across rack leaders ($1M_{r-1}$) traverses $2r$ links: one from the source leader to the ToR switch in its rack, one to the core switch, $r-1$ to all other racks' ToR switches and $r-1$ from the ToR switches to their respective rack leaders. Our analysis is conservative; a typical data center network is more complex. Hence, a single message will traverse more core links in typical data center networks than in our model, which amplifies the difference between *COOL* and the classical implementations.

Process load. The process load is the number of messages each process sends or receives during a single step. It is indicative of the amount of communication and computation required on each process per step.

4.3 Analysis

In this section, we analyze the costs of each collective operation using different communication patterns based on the evaluation metrics defined in Section 4.2. The results form this section are summarized in Section 4.4. Table 4.2 summarizes the number of steps each communication pattern takes for each collective operation, Table 4.3 contains the total number of exchanged messages and the number of messages exchanged between racks, Table 4.4 contains the total network load and the network load across racks and Table 4.5

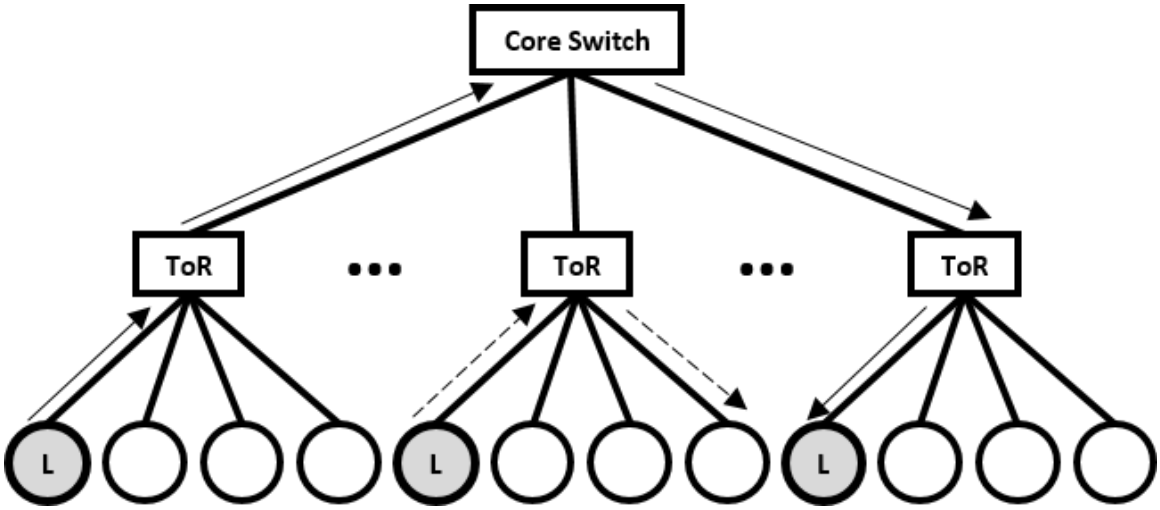


Figure 4.1: Network model. Tree topology with one core switch. Solid arrows illustrate the path for a message across racks, and dashed arrows illustrate the path for a message within a rack.

summarizes the maximum load on processes at any step in the operation. Section 4.4 also includes a discussion and a comparison between *COOL* and the other collective communication patterns.

4.3.1 Ring Pattern

The ring pattern organizes the processes in a ring. In each step of an operation (MPI_Allgather or MPI_Allreduce), every process sends a message to its successor on the ring and receives a message from its predecessor. For instance, to implement the MPI_Allgather operation using the ring pattern, each process receives a message from its predecessor, adds its local value to the message and sends everything to its successor. After $N-1$ steps, all processes will have all the values from all the other processes (See the example in Figure 2.2b). MPI_Allreduce uses the ring pattern in a similar fashion as MPI_Allgather. Once every process gets all the values, it can perform the reduce operation.

Number of steps. For each process to receive all messages from all other processes, the ring pattern takes $N-1$ steps.

Maximum process load. During every step, each process receives one message and sends one message. Therefore, the maximum process load is 2.

Total number of messages. During every step each of the N processes sends exactly one message. Since there are $N-1$ steps, the total number of messages sent is $N\,(N-1)$.

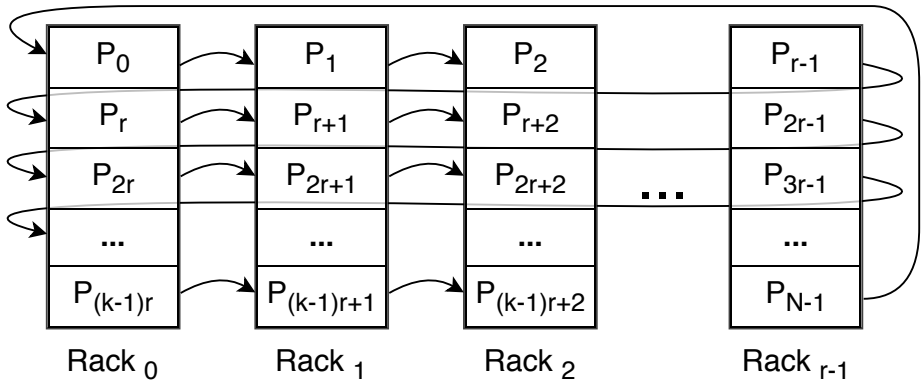
Messages sent across racks. Each MPI process has a rank. The processes form a ring following the ordering of the ranks, meaning that process p has process $p-1 \mod N$ as a predecessor and process $p+1 \mod N$ as a successor. Unfortunately, process ranks are traditionally assigned with no relation to the nodes' locations in the network. The assignment of ranks to nodes in the network has a significant impact on the number of messages sent between racks and consequently on the total network load. To illustrate the importance of this issue we present an example of the worst case assignment for the ring pattern (Figure 4.2a). In this case, every neighboring processes in the ring could reside on a different rack. Hence, all messages sent using the ring pattern will traverse the data center's core network.

In our analysis, we assign process ranks to MPI processes to minimize the communication between racks. Figure 4.2b shows such assignment. This assignment of process ranks maximizes the communication within a rack. Only one message leaves each rack in every step of the ring pattern. Consequently, while all of the $N-1$ steps in the ring pattern use the data center network, in each step only one message leaves each rack (i.e., r messages per step). In total, the ring pattern generates $r\,(N-1)$ messages across racks and the rest of the messages (i.e., $(N-r)\,(N-1)$) are sent/received within a rack.

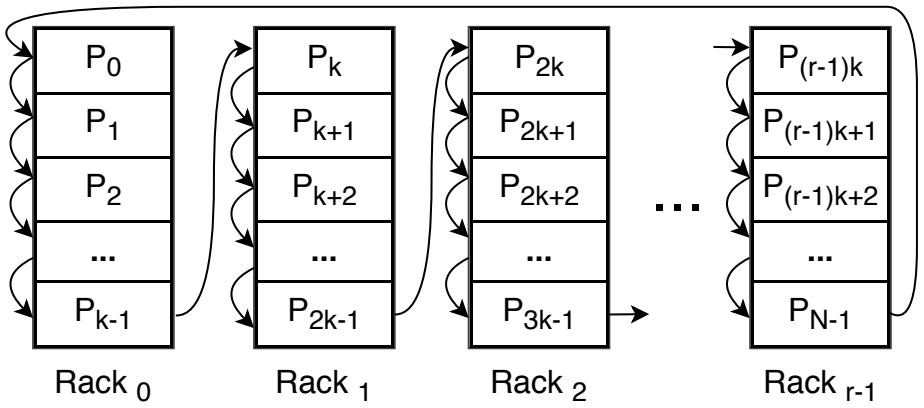
Network load. As explained in Section 4.2, each of the $(N-r)\,(N-1)$ messages sent within a rack traverses two links, so they contribute $2\,(N-r)\,(N-1)$ to the total network load. Additionally, each of the $r\,(N-1)$ messages sent across racks traverses four links, so they contribute $4r\,(N-1)$ to the total network load. The total generated network load is therefore

$$2\,(N-r)\,(N-1) + 4r\,(N-1) = 2\,(N+r)\,(N-1)$$

Each of the $r\,(N-1)$ messages sent between racks traverses 2 links, so the network load across racks is $2r\,(N-1)$.



(a) Worst case rank assignment.



(b) Optimal rank assignment.

Figure 4.2: Rank assignments for the ring pattern. The arrows represent the direction of message communication at any step.

4.3.2 Binomial Tree Pattern

A *binomial tree* pattern is used in MPI_Gather, MPI_Reduce, MPI_Scatter and MPI_Bcast. To implement the MPI_Gather operation using the binomial tree pattern, in each step, processes are divided into two halves; one half sends its local data with the data it has gathered so far to the other half. The receiving half repeats this procedure until a single process is left. That process, called the root process, will have all data items from all processes (See the example in Figure 2.2c). MPI_Reduce uses the binomial tree pattern in a similar fashion as MPI_Gather. Once the root process gets all the values, it can perform the reduce operation.

Similarly, to implement the MPI_Scatter operation, the same steps are run in reverse order: starting with one process (the root process), in every step i, each process that has data divides its data in half, keeps one half and sends the other half to a process 2^{i-1} ranks away until every process gets a chunk of data. MPI_Bcast uses the binomial tree pattern in a similar fashion as MPI_Scatter, except that the complete data buffer is copied in each step instead of being divided in half.

Number of steps. In MPI_Gather and MPI_Reduce the number of participating processes starts at N and is divided by two in each step until we reach one process. In MPI_Scatter and MPI_Bcast, the number of participating processes starts at one and is multiplied by two until we reach N. Thus, the total number of steps required using the binomial tree pattern is $\log N$.

Maximum process load. During every step each process either receives one message, sends one message, or does nothing. Therefore, the maximum process load is 1.

Total number of messages. In MPI_Gather and MPI_Reduce operations each process (except the root process) sends exactly one message. Similarly, in MPI_Scatter and MPI_Bcast each process (except the root process) receives a single message. Hence, the total number of messages in any operation that uses the binomial tree pattern is $N - 1$.

Messages sent across racks. Each MPI process has a rank. Processes send messages following the ordering of the ranks. For instance, in MPI_Gather and MPI_Reduce for

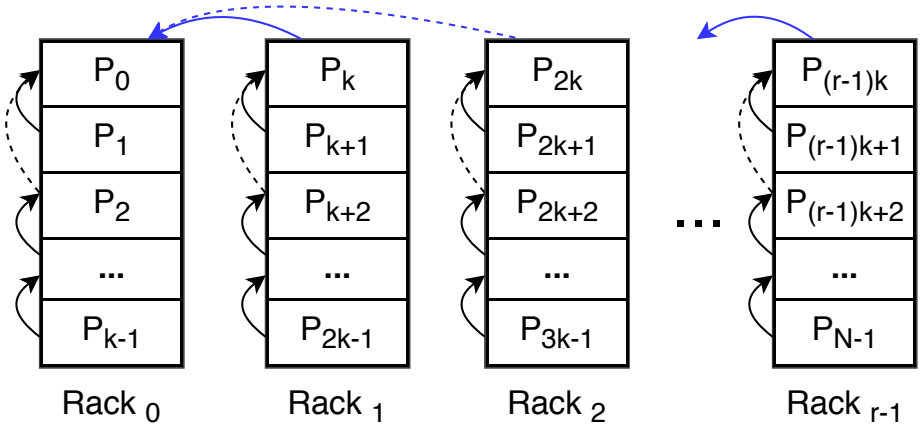
every step i starting from 1 to $\log N$, each process p where $N/2^i \leq p < N/2^{i-1}$ sends a message to process $p - N/2^i$. In the case of MPI_Scatter and MPI_Bcast, for every step i starting from 1 to $\log N$, each process p where $0 \leq p < 2^{i-1}$ sends a message to process $p + 2^{i-1}$. Unfortunately, process ranks are traditionally assigned with no relation to the nodes' locations in the network.

In our analysis, we assign process ranks to MPI processes to minimize the communication between racks. Figure 4.3a shows such an assignment for MPI_Gather and MPI_Reduce. This assignment of process ranks maximizes the communication within a rack by doing the steps that involve the most number of messages within racks and the rest across racks. Only in the steps i where $\log N - \log r < i \leq \log N$ do all exchanged messages leave the racks. This implies that only $\log r$ steps in the binomial tree pattern use the data center network. Since $2^{-i}N$ messages are sent in each step i where $\log N - \log r < i \leq \log N$, the binomial tree pattern generates $r - 1$ messages in total across racks. The rest of the messages (i.e., $N - r$) are sent/received within a rack.

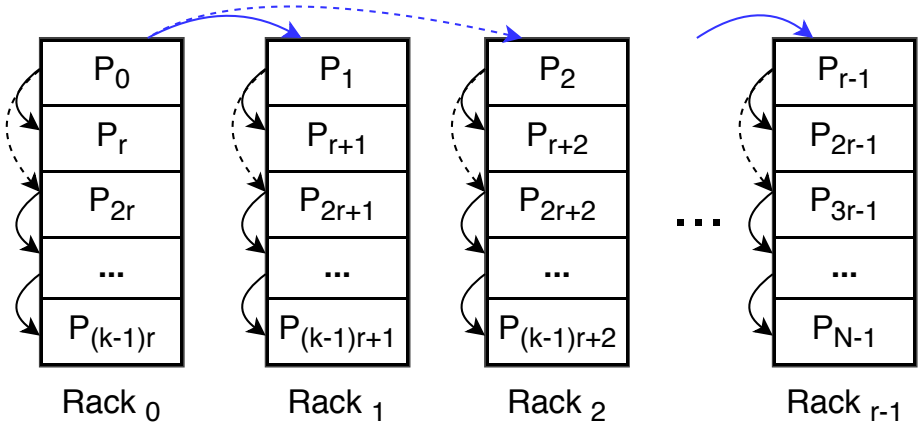
For MPI_Scatter and MPI_Bcast the order of the steps is reversed compared with MPI_Gather and MPI_Reduce. Therefore, the rank assignment using the binomial tree pattern (Figure 4.3b) is different than that of MPI_Gather and MPI_Reduce. This assignment of process ranks maximizes the communication within a rack by doing the steps that involve the most number of messages within racks and the rest across racks. Only in the steps i where $1 \leq i \leq \log r$ do all exchanged messages leave the racks. This implies that only $\log r$ steps in the binomial tree pattern use the data center network. Since 2^{i-1} messages are sent in each step i where $1 \leq i \leq \log r$, the binomial tree pattern generates $r-1$ messages in total across racks. The rest of the messages (i.e., $N-r$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $N - r$ messages sent within a rack traverses two links, so they contribute $2\,(N - r)$ to the total network load. Additionally, each of the $r - 1$ messages sent across racks traverses four links, so they contribute $4\,(r - 1)$ to the total network load. The total generated network load is therefore

$$2\,(N - r) + 4\,(r - 1) = 2N + 2r - 4$$



(a) Rank assignment for MPI_Gather and MPI_Reduce. Solid black lines represent step 1, dashed black lines represent step 2, solid blue lines represent step $\log N - \log r + 1$ and dashed blue lines represent step $\log N - \log r + 2$



(b) Rank assignment for MPI_Scatter and MPI_Bcast. Solid blue lines represent step 1, dashed blue lines represent step 2, solid black lines represent step $\log N - \log r + 1$ and dashed black lines represent step $\log N - \log r + 2$

Figure 4.3: Rank assignments for the binomial tree pattern.

Each of the $r-1$ messages sent between racks traverses 2 links, so the network load across racks is $2r-2$.

4.3.3 Recursive Doubling Pattern

The recursive doubling pattern is used in MPI_Allreduce and MPI_Allgather. The pattern involves exchanging messages between pairs of processes that are initially a distance of 1 apart, then doubling that distance for each step until a distance of $N/2$ is reached (See the example in Figure 2.2a). For instance, to implement MPI_Allgather, in every step i (starting from 1), each pair of processes 2^{i-1} ranks apart exchange local data and data they received so far with each other. After $\log N$ steps, all processes will have all the values from all the other processes. MPI_Allreduce uses the recursive doubling pattern in a similar fashion as MPI_Allgather. Once every process gets all the values, it can perform the reduce operation.

Number of steps. The distance between each pair of processes that exchange messages is doubled in each step from 1 to $N/2$. Therefore, for each process to receive all messages from all other processes, the recursive doubling pattern requires $\log N$ steps.

Maximum process load. During every step each process receives one message and sends one message. Therefore, the maximum process load is 2.

Total number of messages. During every step each of the N processes sends exactly one message. Since there are $\log N$ steps, the total number of messages sent is $N \log N$.

Messages sent across racks. Each MPI process has a rank. The pairs of processes that exchange messages are chosen following the ordering of the ranks. This means that for each step i starting from 1 to $\log N$, process p exchanges messages with process $p \pm 2^{i-1}$ (The operation is $+$ or $-$ depending on whether $\lfloor p/2^{i-1} \rfloor$ is even or odd, respectively).

In our analysis, we assign process ranks to MPI processes to minimize the communication between racks. Figure 4.4 shows such an assignment. This assignment of process ranks maximizes the communication within a rack. Only in the steps where the distance between pairs of processes is bigger than the size of the rack ($2^{i-1} > k$, so $i > \log k + 1$) do

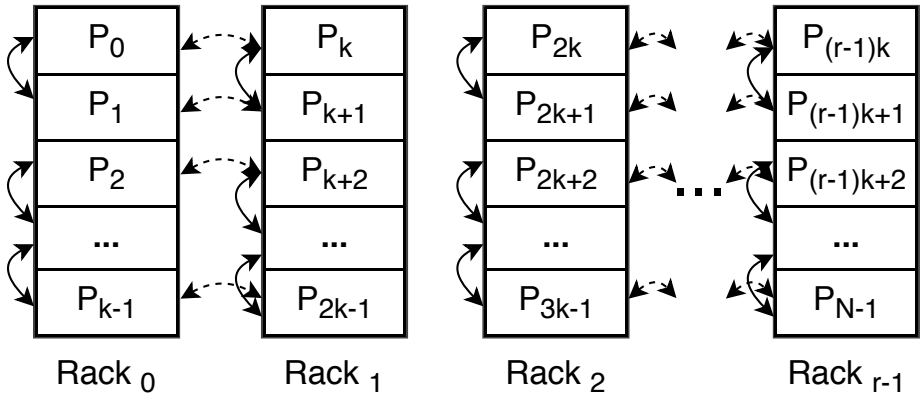


Figure 4.4: Rank assignments for the recursive doubling pattern. Solid lines represent step 1, while dashed lines represent step $\log k + 1$.

all exchanged messages leave the racks. This implies that only $\log r$ steps in the recursive doubling pattern use the data center network. Since N messages are sent in each step, the recursive doubling pattern generates $N \log r$ messages in total across racks. The rest of the messages (i.e., $N \log N - N \log r$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $N \log N - N \log r$ messages sent within a rack traverses two links, so they contribute $2\,(N \log N - N \log r)$ to the total network load. Additionally, each of the $N \log r$ messages sent across racks traverses four links, so they contribute $4\,(N \log r)$ to the total network load. The total generated network load is therefore

$$2\,(N \log N - N \log r) + 4N \log r = 2N \log N + 2N \log r$$

Each of the $N \log r$ messages sent between racks traverses 2 links, so the network load across racks is $2N \log r$.

4.3.4 Bruck's Pattern

The Bruck's pattern is used in MPI_Allgather. The pattern involves sending messages messages from each process to a process that is initially a distance of 1 away, then dou-

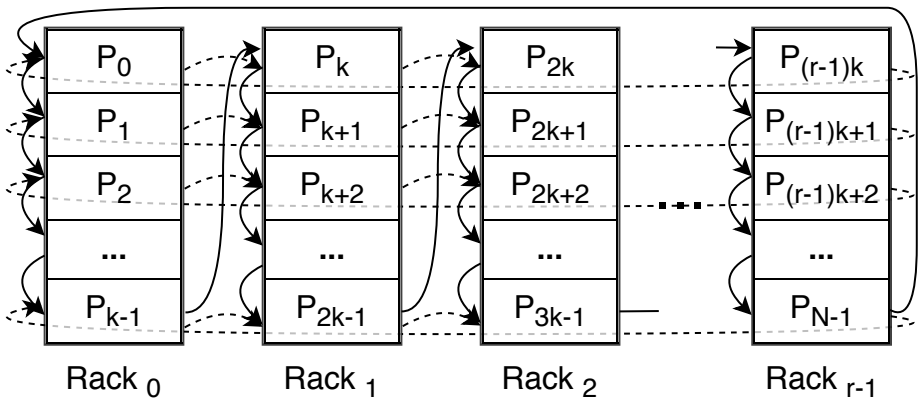


Figure 4.5: Rank assignments for the Bruck's pattern. Solid lines represent step 1, while dashed lines represent step $\log k + 1$

bling that distance for each step until a distance of $N/2$ is reached (See the example in Figure 2.2a). For instance, to implement MPI_Allgather, in every step i (starting from 1), each process sends local data and data it received so far to the processes 2^{i-1} ranks away. After $\log N$ steps, all processes will have all the values from all the other processes.

Number of steps. The distance between each pair of sending and receiving processes is doubled in each step from 1 to $N/2$. Therefore, for each process to receive all messages from all other processes, the Bruck's pattern takes $\log N$ steps.

Maximum process load. During every step each process receives one message and sends one message. Therefore, the maximum process load is 2.

Total number of messages. During every step each of the N processes sends exactly one message. Since there are $\log N$ steps, the total number of messages sent is $N \log N$.

Messages sent across racks. Each MPI process has a rank. The pairs of processes that exchange messages are chosen following the ordering of the ranks. This means that for each step i starting from 1 to $\log N$, process p exchanges messages with process $p + 2^{i-1}$ mod N.

In our analysis, we assign process ranks to MPI processes to minimize the communication between racks. Figure 4.5 shows such an assignment. This assignment of process

ranks maximizes the communication within a rack. In the steps where the distance between pairs of processes is bigger than the size of the rack ($2^{i-1} > k$, so $i > \log k + 1$), all exchanged messages leave the racks. Since N messages are sent in each of these steps, and there are $\log N - \log k = \log r$ of these steps, then $N \log r$ messages are generated in all steps $i > \log k + 1$. In the other steps, where the distance between pairs of processes is at most the size of the rack ($2^{i-1} \leq k$, so $i \leq \log k + 1$), only 2^{i-1} leave each rack. Since $2^{i-1}r$ messages are sent in each of these steps and there are $\log k$ of these steps, then $r\,(k-1) = N - r$ messages are generated in all steps $i \leq \log k + 1$. This implies that when using the Bruck's pattern all $\log N$ steps use the data center network and that the Bruck's pattern generates $N \log r + N - r$ messages in total across racks. The rest of the messages (i.e., $N \log N - N \log r - N + r$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $N \log N - N \log r - N + r$ messages sent within a rack traverses two links, so they contribute $2\,(N \log N - N \log r - N + r)$ to the total network load. Additionally, each of the $N \log r + N - r$ messages sent across racks traverses four links, so they contribute $4\,(N \log r + N - r)$ to the total network load. The total generated network load is therefore

$$2\,(N \log N - N \log r - N + r) + 4\,(N \log r + N - r) = 2N \log N + 2N \log r + 2N - 2r$$

Each of the $N \log r + N - r$ messages sent between racks traverses 2 links, so the network load across racks is $2N \log r + 2N - 2r$.

4.3.5 Rabenseifner's Pattern for MPI_Reduce

Rabenseifner's pattern [41] uses two phases combining two patterns to implement MPI_Reduce. In the first phase, it uses the recursive halving pattern (analogous to the recursive doubling pattern) to perform *reduce-scatter*, where the data array is divided to chunks and each chunk is reduced to a different node. This phase involves exchanging messages between pairs of processes that are initially a distance of 1 apart, then doubling that distance for each step until a distance of $N/2$ is reached (See the example in Figure 2.2e, steps A1-A3).

This means that in every step i (starting from 1), each pair of processes 2^{i-1} ranks apart combine the local data and data they received so far, divide the data in half, keep one half and exchange the other half with each other. They then apply the reduce operation to that data. After $\log N$ steps, each process will have a different chunk of data reduced from all the other processes.

In the second phase, Rabenseifner's Pattern for MPI_Reduce uses the binomial tree pattern to gather the reduced chunks from the previous phase at the root processes. In each step of this phase, processes are divided into two halves; one half sends its local data from the previous phase with the data it has gathered so far to the other half. The receiving half repeats this procedure until a single process is left. That process, called the root process, will have all reduced chunks from all processes (See the example in Figure 2.2e, steps B1-B3).

Number of steps. In the first phase, the distance between each pair of processes that exchange messages is doubled in each step from 1 to $N/2$. So for each process to receive all messages from all other processes, the recursive halving phase requires $\log N$ steps. In the second phase, the number of participating processes starts at N and is divided by two in each step until we reach one process. Thus, the total number of steps required in the binomial tree phase is $\log N$. Therefore, Rabenseifner's Pattern for MPI_Reduce requires $2 \log N$ steps in total.

Maximum process load. In the first phase each process receives one message and sends one message during every step, so the maximum process load for that phase is 2. During every step of the second phase each process either receives one message, sends one message, or does nothing, so the maximum process load for that phase is 1. Therefore the maximum process load is $\max(2, 1) = 2$.

Total number of messages. In the first phase, during every step each of the N processes sends exactly one message. Since there are $\log N$ steps in this phase, the number of messages sent in the recursive halving phase is $N \log N$. In the second phase, each process (except the root process) sends exactly one message, so the total number of messages is

$N-1$. Therefore, Rabenseifner's Pattern for MPI_Reduce requires $N \log N + N - 1$ messages in total.

Messages sent across racks. Each MPI process has a rank. The pairs of processes that exchange messages in either phase are chosen following the ordering of the ranks. This means that in the first phase for each step i starting from 1 to $\log N$, process p exchanges messages with process $p \pm 2^{i-1}$ (The operation is $+$ or $-$ depending on whether $\lfloor p/2^{i-1} \rfloor$ is even or odd, respectively). In the second phase, for every step i starting from 1 to $\log N$, each process p where $N/2^i \leq p < N/2^{i-1}$ sends a message to process $p - N/2^i$.

In our analysis, we assign process ranks to MPI processes to minimize the communication between racks. Figure 4.4 shows such an assignment for the first phase. This assignment of process ranks maximizes the communication within a rack. Only in the steps where the distance between pairs of processes is bigger than the size of the rack ($2^{i-1} > k$, so $i > \log k + 1$) do all exchanged messages leave the racks. This implies that only $\log r$ steps in the recursive halving phase use the data center network. Since N messages are sent in each step of that phase, the recursive halving phase generates $N \log r$ messages across racks. Figure 4.3a shows such an assignment for the second phase. This assignment of process ranks maximizes the communication within a rack by doing the steps that involve the most number of messages within racks and the rest across racks. Only in the steps i where $\log N - \log r < i \leq \log N$ do all exchanged messages leave the racks. This implies that only $\log r$ steps in the binomial tree phase use the data center network. Since $2^{-i}N$ messages are sent in each step i where $\log N - \log r < i \leq \log N$, the binomial tree phase generates $r - 1$ messages across racks. The rest of the messages (i.e., $N - r$) are sent/received within a rack.

Therefore, Rabenseifner's Pattern for MPI_Reduce generates $N \log r + N - r$ messages in total across racks. The rest of the messages (i.e., $N \log N - N \log r + r - 1$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $N \log N - N \log r + r - 1$ messages sent within a rack traverses two links, so they contribute $2\,(N \log N - N \log r + r - 1)$ to the total network load. Additionally, each of the $N \log r + N - r$ messages sent across

racks traverses four links, so they contribute $4\,(N \log r + N - r)$ to the total network load. The total generated network load is therefore

$$2\,(N \log N - N \log r + r - 1) + 4\,(N \log r + N - r) = 2N \log N + 2N \log r + 4N - 2r - 2$$

Each of the $N \log r + N - r$ messages sent between racks traverses 2 links, so the network load across racks is $2N \log r + 2N - 2r$.

4.3.6 Rabenseifner's Pattern for MPI_Allreduce

Rabenseifner's pattern [41] uses two phases combining two patterns to implement MPI_Allreduce. In the first phase, it uses the recursive halving pattern (analogous to the recursive doubling pattern) to perform *reduce-scatter*, where the data array is divided to chunks and each chunk is reduced to a different node. This phase involves exchanging messages between pairs of processes that are initially a distance of 1 apart, then doubling that distance for each step until a distance of $N/2$ is reached (See the example in Figure 2.2f, steps A1-A3). This means that in every step i (starting from 1), each pair of processes 2^{i-1} ranks apart combine the local data and data they received so far, divide the data in half, keep one half and exchange the other half with each other. They then apply the reduce operation to that data. After $\log N$ steps, each process will have a different chunk of data reduced from all the other processes.

In the second phase, Rabenseifner's Pattern for MPI_Allreduce uses the recursive doubling pattern to gather the reduced chunks from the previous phase in all processes. This phase involves exchanging messages between pairs of processes that are initially a distance of 1 apart, then doubling that distance for each step until a distance of $N/2$ is reached (See the example in Figure 2.2f, steps B1-B3). This means that in every step i (starting from 1), each pair of processes 2^{i-1} ranks apart exchange local data and data they received so far with each other. After $\log N$ steps, all processes will have all the reduced chunks from all the other processes.

Number of steps. In the first phase, the distance between each pair of processes that exchange messages is doubled in each step from 1 to $N/2$. So for each process to receive all messages from all other processes, the recursive halving phase requires $\log N$ steps. Similarly, in the second phase, the distance between each pair of processes that exchange messages is doubled in each step from 1 to $N/2$. So for each process to receive all messages from all other processes, the recursive doubling phase requires $\log N$ steps. Therefore, Rabenseifner's Pattern for MPI_Allreduce requires $2 \log N$ steps in total.

Maximum process load. In either phase each process receives one message and sends one message during every step. Therefore, the maximum process load is 2.

Total number of messages. In the first phase, during every step each of the N processes sends exactly one message. Since there are $\log N$ steps in this phase, the number of messages sent in the recursive halving phase is $N \log N$. In the second phase, during every step each of the N processes sends exactly one message. Since there are $\log N$ steps in this phase, the number of messages sent in the recursive doubling phase is $N \log N$. Therefore, Rabenseifner's Pattern for MPI_Allreduce requires $2N \log N$ messages in total.

Messages sent across racks. Each MPI process has a rank. The pairs of processes that exchange messages in either phase are chosen following the ordering of the ranks. This means that in either phase for each step i starting from 1 to $\log N$, process p exchanges messages with process $p \pm 2^{i-1}$ (The operation is $+$ or $-$ depending on whether $\lfloor p/2^{i-1} \rfloor$ is even or odd, respectively).

In our analysis, we assign process ranks to MPI processes to minimize the communication between racks. Figure 4.4 shows such an assignment for the first phase. This assignment of process ranks maximizes the communication within a rack. Only in the steps where the distance between pairs of processes is bigger than the size of the rack ($2^{i-1} > k$, so $i > \log k + 1$) do all exchanged messages leave the racks. This implies that only $\log r$ steps in the recursive halving phase use the data center network. Since N messages are sent in each step of that phase, the recursive halving phase generates $N \log r$ messages across racks. The same argument is used for the recursive doubling phase. There-

fore, Rabenseifner's Pattern for MPI_Allreduce generates $2N \log r$ messages in total across racks. The rest of the messages (i.e., $2N \log N - 2N \log r$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $2N \log N - 2N \log r$ messages sent within a rack traverses two links, so they contribute $2\,(2N \log N - 2N \log r)$ to the total network load. Additionally, each of the $2N \log r$ messages sent across racks traverses four links, so they contribute $4\,(2N \log r)$ to the total network load. The total generated network load is therefore

$$2\,(2N \log N - 2N \log r) + 4\,(2N \log r) = 4N \log N + 4N \log r$$

Each of the $2N \log r$ messages sent between racks traverses 2 links, so the network load across racks is $4N \log r$.

4.3.7 COOL-Parallel-Parallel for MPI_Gather, MPI_Reduce and MPI_Scatter

COOL-P-P uses the parallel communication pattern within a rack and between racks. For instance, in MPI_Gather it works as follows. (1) Each process sends its data to its rack leader. Each leader concatenates the data from its subgroup into a subarray. (2) All leaders send their subarrays to the root leader (the rack leader of the root process). The root leader concatenates all the subarrays it receives into a single final array. Finally, (3) the root leader forwards the final array to the root process. To implement the MPI_Reduce operation, *COOL-P-P* runs in a similar fashion, except that at the end of step (1), once each rack leader gets all the data from its rack, it performs the reduce operation. Additionally, at the end of step (2), once the root leader receives all the data from all rack leaders it performs the reduce operation.

Similarly, to implement the MPI_Scatter operation, the same steps are run in reverse order. (1) The root process sends its array to the root leader. (2) The root leader divides the array into r subarrays, keeps one subarray and sends a subarray to all other rack leaders.

Finally, (3) every rack leader, including the root leader, sends $1/k^{th}$ of the remaining data items from its subarray to every process in its rack.

Number of steps. Since there are three phases in *COOL*, and each phase uses the single-step parallel pattern, then there are three steps in total.

Maximum process load. In the MPI_Gather and MPI_Reduce operations during the first step, rack leaders receive $k-1$ messages. In the second step, the root leader receives $r-1$ messages, while other rack leaders send one message. In the third step, the root leader sends one message. On the other hand, in the MPI_Scatter operation, during the first step, the root leader receives one message. In the second step, the root leader sends $r-1$ messages, while other rack leaders receive one message. In the third step, rack leaders send $k-1$ messages. Therefore, in all the three operations the maximum process load *for each leader process* is $\max(r-1, k-1)$. During every step, each non-leader process sends one message, receives one message, or does nothing. Therefore, the maximum process load *for other processes* is 1.

Total number of messages. In the MPI_Gather and MPI_Reduce operations during the first step, each process sends one message to its rack leader ($r(k-1) = N-r$ messages). In the second step, each rack leader sends one message to the root leader ($r-1$ messages). Finally, the root leader sends one message to the root process (one message). In MPI_Scatter, during the first step, the root process sends one message to the root leader (one message). In the second step, the root leader sends one message to each rack leader ($r-1$ messages). Finally, each rack leader, including the root leader, sends one message to each process in its rack ($r(k-1) = N-r$ messages). This implies that *COOL-P-P* requires $N-r+r-1+1 = N$ messages in total.

Messages sent across racks. Messages are only sent between racks in the second step. Therefore, there are $r-1$ messages that use the core network. The rest of the messages ($N-r+1$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $N-r+1$ messages sent within a rack traverses two links, so they contribute $2(N-r+1)$ to the total network load. Additionally, each of the $r-1$ messages sent across racks traverses four links, so

they contribute $4\,(r-1)$ to the total network load. The total generated network load is therefore

$$2\,(N-r+1)+4\,(r-1)=2N+2r-2$$

Each of the $r-1$ messages sent between racks traverses 2 links, so the network load across racks is $2r-2$.

4.3.8 COOL-Parallel-Parallel for MPI_Bcast

COOL-P-P uses the parallel comunication pattern within a rack and between racks. The MPI_Bcast operation works as follows. (1) The root process sends its value to the root leader. (2) The root leader multicasts the value to all other rack leaders. Finally, (3) every leader multicasts the value to all processes its rack.

Number of steps. Since there are three phases in *COOL*, and each phase uses the single-step parallel pattern, then there are three steps in total.

Maximum process load. In MPI_Bcast, during the first step, the root process sends one message to the root leader. In the second step, the root leader sends one multicast message to the other $r-1$ rack leaders. In the third step, each rack leader, including the root leader, sends one multicast message to the other $k-1$ processes in its rack. Thus the maximum process load *for each leader process* is 1. During every step, each non-leader process sends one message, receives one message, or does nothing. Therefore, the maximum process load *for the other processes* is 1.

Total number of messages. During the first step of MPI_Bcast, the root process sends one message to the root leader (one message). In the second step, the root leader sends one message to the other $r-1$ rack leaders (1 multicast message to $r-1$ recipients, i.e., $1M_{r-1}$). Finally, each rack leader, including the root leader, sends one multicast message to all the processes in its rack (r multicast messages to $k-1$ recipients, i.e., rM_{k-1}). This implies that *COOL-P-P* requires $1+1M_{r-1}+rM_{k-1}$ messages in total.

Messages sent across racks. Messages are only sent between racks in the second step. Therefore, there is only one multicast message to $r-1$ recipients ($1M_{r-1}$) that uses the core network. The rest of the messages ($1 + rM_{k-1}$) are sent/received within a rack.

Network load. As explained in Section 4.2, the only message sent within a rack traverses two links, so it contributes $2\,(1)$ to the total network load. Additionally, each of the rM_{k-1} multicast messages to $k-1$ recipients within a rack uses k links, so they contribute $k\,(r)$ to the total network load. A multicast message across rack leaders ($1M_{r-1}$) traverses $2r$ links, so it contributes $2r\,(1)$ to the total network load. The total generated network load is therefore

$$2\,(1) + k\,(r) + 2r\,(1) = N + 2r + 2$$

Each multicast message sent between racks traverses r links in the core network, so the network load across racks is r.

4.3.9 COOL-Parallel-Parallel for MPI_Allgather and MPI_Allreduce

COOL-P-P uses the parallel communication pattern within a rack and between racks. For instance, the MPI_Allgather operation works as follows. (1) Each process sends its value to its rack leader. Each rack leader concatenates all the values of its subgroup into a subarray. (2) All rack leaders multicast their subarrays to all other rack leaders. Every leader concatenates all of the subarrays it receives into a single final array. Finally, (3) every leader multicasts the final array to the other $k-1$ processes in its rack. The MPI_Allreduce operation is done in a similar fashion, except that at the end of step (1), once each rack leader gets all the values from its rack, it performs the reduce operation on the data it has so far. Additionally, at the end of step (2), once each rack leader receives all values from all other rack leaders it performs the reduce operation. At the end of Step (3), all processes receive the final reduced data.

Number of steps. Since there are three phases in *COOL*, and each phase uses the single-step parallel pattern, then there are three steps in total.

Maximum process load. In MPI_Allgather and MPI_Allreduce, during the first step, rack leaders receive $k-1$ messages. In the second step, each rack leader sends one multicast message to all other $(r-1)$ rack leaders and receives $r-1$ messages. In the third step, each rack leader sends one multicast message to the other $k-1$ processes in its rack. Thus the maximum process load *for each leader process* is $\max(r, k-1)$. During every step, each non-leader process sends one message, receives one message, or does nothing. Therefore, the maximum process load *for other processes* is 1.

Total number of messages. In MPI_Allgather and MPI_Allreduce, during the first step, each process sends one message to its rack leader ($r(k-1) = N - r$ messages). In the second step, each rack leader sends one multicast message to all other $(r-1)$ rack leaders (rM_{r-1} messages). Finally, each rack leader sends one multicast message to the other $k-1$ processes in its rack (rM_{k-1} messages). This implies that *COOL-P-P* requires $N - r + rM_{r-1} + rM_{k-1}$ messages in total.

Messages sent across racks. Messages are only sent between racks in the second step. Therefore, there are r multicast message to $r-1$ recipients (rM_{r-1}) that use the core network. The rest of the messages ($N - r + rM_{k-1}$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $N-r$ messages sent within a rack traverses two links, so they contribute $2(N-r)$ to the total network load. Additionally, each of the rM_{k-1} multicast messages to $k-1$ recipients within a rack uses k links, so they contribute $k(r)$ to the total network load. Each of the rM_{r-1} multicast messages sent from one rack leader to the other $r-1$ rack leaders traverses $2r$ links, so they contribute $2r(r)$ to the total network load. The total generated network load is therefore

$$2(N-r) + k(r) + 2r(r) = 3N + 2r^2 - 2r$$

Each of the rM_{k-1} multicast messages sent between racks traverses r links in the core network, so the network load across racks is r^2.

4.3.10 COOL-Binomial-Tree-Recursive-Doubling for MPI_Allreduce

COOL-Binomial-tree-Recursive-doubling (*COOL-B-R*) uses the binomial tree pattern within a rack and the recursive doubling pattern between racks. In MPI_Allreduce it works as follows. In the first phase, each process sends its value to its rack leader using the binomial tree communication pattern. In each step, processes in each rack are divided into two halves; one half applies the reduce operation to its local data and the data it has gathered so far and sends the result to the other half. The receiving half repeats this procedure until a single process (the rack leader) is left. Each rack leader will then have the reduced result from all processes in its rack. In the second phase, each rack leader sends its reduced result to all other rack leaders using the recursive doubling communication pattern. In every step i (starting from 1), each pair of rack leaders with a distance of 2^{i-1} between their rack ids combine their data and the data they have received so far and exchange that data with each other. They then apply the reduce operation to that data. After $\log N$ steps all rack leaders will have the reduced result from all the other processes. Finally, in the third phase every rack leader multicasts the final reduced result to the other $k-1$ processes in its rack.

Number of steps. In the first phase of *COOL-B-R*, the number of processes that are involved in communication in each rack starts at k and is divided by two in each step until the rack leader is reached. Thus, the number of steps required in the first phase using the binomial tree pattern is $\log k$. In the second phase, the distance between the rack ids of each pair of rack leaders that exchange messages is increased exponentially from 1 to $r/2$. So for each rack leader to receive all messages from all other rack leaders, the second phase using the recursive doubling pattern requires $\log r$ steps. The third phase uses multicast messages to send data from rack leaders to all processes in their respective racks. Since these multicast messages happen simultaneously, that takes a single step. Therefore, the total number of steps is $\log k + \log r + 1 = \log kr + 1 = \log N + 1$.

Maximum process load. In the first phase, each process sends one message or receives one message in each step, so the maximum process load for each leader and non-leader process is 1. In the second phase, rack leaders send one message and receive one

message in each step, so the maximum process load for rack leaders is 2. The other processes do nothing in this phase, so the maximum process load for the other processes is 0. In the third phase, each rack leader sends one multicast message to the other $k-1$ processes in its rack, so the maximum process load for each leader and non-leader process is 1. Therefore, the maximum process load for *each rack leader* is $\max(1,2,1) = 2$, and the maximum process load for *the other processes* is $\max(1,0,1) = 1$.

Total number of messages. During the first phase, each process in each rack (except the rack leaders) sends exactly one message. Hence, the number of messages exchanged in the first phase is $N - r$. During every step of the second phase, each rack leader sends exactly one message. Since there are $\log r$ steps in the second phase, the number of messages exchanged in this phase is $r \log r$. Finally, in the third phase every rack leader sends one multicast message to the other $k-1$ processes in its rack, so there are rM_{k-1} messages in this phase. Therefore, the total number of messages exchanged in *COOL-B-R* is $N - r + r \log r + rM_{k-1}$.

Messages sent across racks. Messages are only sent between racks in the second step. Therefore, there are $r \log r$ messages that use the core network. The rest of the messages ($N - r + rM_{k-1}$) are sent/received within a rack.

Network load. As explained in Section 4.2, each of the $N - r$ messages sent within a rack traverses two links, so they contribute $2(N-r)$ to the total network load. Additionally, each of the rM_{k-1} multicast messages to $k-1$ recipients within a rack uses k links, so they contribute $k(r)$ to the total network load. Each of the $r \log r$ messages sent across racks traverses four links, so they contribute $4(r \log r)$ to the total network load. The total generated network load is therefore

$$2(N-r) + k(r) + 4(r \log r) = 3N + 4r \log r - 2r$$

Each of the $r \log r$ messages sent between racks traverses 2 links, so the network load across racks is $2r \log r$.

4.4 Summary of Analysis and Discussion

This section summarizes the analysis presented in Section 4.3, discusses the results and compares between *COOL* and the other collective communication patterns. The analysis assumes the best ranking for classical implementations which achieves the least possible cross-rack communication. In this section, *COOL-P-P* denotes a *COOL* pattern that uses the parallel pattern for communication within and across racks and *COOL-B-R* denotes a *COOL* pattern that uses the binomial tree pattern within a rack and recursive doubling pattern across racks.

4.4.1 Operation Latency

We analyze operation latency by examining the number of steps each pattern takes to complete. Table 4.2 summarizes our results showing the total number of steps that various implementations of the collective operations require and the number of steps that use the inter-rack links (Table 4.1 describes the parameters used in Table 4.2). Classical implementations take from $\log N$ to N steps to complete with $\log r$ to N of the steps using the network between racks, while all *COOL-P-P* operations complete in three steps, with only one step using the network across racks. This is a byproduct of using the *parallel* pattern (as it can complete its part in a single step) and the hierarchical design (which confines the cross-rack communication to a single step).

Obviously, a step in the *parallel* pattern has a higher computation and communication overhead than a step in the other patterns. A process in the classical patterns exchanges messages with one or two processes in every step, whereas the leader in the *parallel* pattern may concurrently communicate with up to k processes in a rack, or r leaders across racks. We study this factor in Section 4.4.4. Nevertheless, if the amount of communication and computation that leaders perform becomes a concern, implementers of *COOL* can reduce this overhead by choosing a different pattern (e.g., binomial tree) for communication within or across racks (Section 4.5 presents one example).

Table 4.2: Number of steps. This table presents the number of steps that a pattern takes to finish and the number of steps requiring communication between racks.

Operation	**Pattern**	**# of steps**		**Section**
		Total	**Across racks**	
Reduce	*COOL-P-P*	3	1	4.3.7
	Binomial tree	$\log N$	$\log r$	4.3.2
	Rabenseifner	$2 \log N$	$2 \log r$	4.3.5
Allreduce	*COOL-P-P*	3	1	4.3.9
	COOL-B-R	$\log N + 1$	$\log r$	4.3.10
	Ring	$N - 1$	$N - 1$	4.3.1
	Recursive doubling	$\log N$	$\log r$	4.3.3
	Rabenseifner	$2 \log N$	$2 \log r$	4.3.6
Gather	*COOL-P-P*	3	1	4.3.7
	Binomial tree	$\log N$	$\log r$	4.3.2
Allgather	*COOL-P-P*	3	1	4.3.9
	Ring	$N - 1$	$N - 1$	4.3.1
	Recursive doubling	$\log N$	$\log r$	4.3.3
	Bruck	$\log N$	$\log N$	4.3.4
Bcast	*COOL-P-P*	3	1	4.3.8
	Binomial tree	$\log N$	$\log r$	4.3.2
Scatter	*COOL-P-P*	3	1	4.3.7
	Binomial tree	$\log N$	$\log r$	4.3.2

4.4.2 Exchanged Messages

Table 4.3 shows the number of messages exchanged in total and across racks. Our results show that *COOL-P-P* achieves the minimal number of messages of N for MPI_Reduce, MPI_Gather, MPI_Bcast and MPI_Scatter. Furthermore, *COOL-P-P* achieves the optimal number of messages across racks for all operations, as it generates no more than $r - 1$ messages, which is the absolute minimum required for exchanging data from one rack to the remaining $r - 1$ racks. This result is due to two design decisions. First, the cross-rack communication is only done between r rack leaders. Second, IP-level multicasting is used for all multicast phases in MPI_Bcast, MPI_Allreduce and MPI_Allgather (which

classical implementations do not support), leading to a significant reduction in the number of messages sent. Here, we count a multicast message as a single sent message, although its network load is higher than that of a single message. Section 4.4.3 addresses this issue.

Compared with other patterns, *COOL-P-P* reduces the number of exchanged messages by a factor of $\log N$ to N for all patterns except for the binomial tree pattern. If process ranks are ordered in a rack-aware fashion (which, as noted in Section 4.3, is something classical MPI implementations cannot do) the binomial tree pattern can minimize cross-rack communication. On the other hand, the binomial tree pattern significantly increases the number of steps to $\log N$ steps.

Table 4.3: Number of messages. This table presents the total number of exchanged messages and the number of messages requiring communication across racks.

Operation	**Pattern**	**Number of messages**		**Section**
		Total	**Across racks**	
Reduce	*COOL-P-P*	N	$r-1$	4.3.7
	Binomial tree	$N-1$	$r-1$	4.3.2
	Rabenseifner	$N\log N+N-1$	$N\log r+r-1$	4.3.5
Allreduce	*COOL-P-P*	$N-r+rM_{r-1}+rM_{k-1}$	rM_{r-1}	4.3.9
	COOL-B-R	$N-r+r\log r+rM_{k-1}$	$r\log r$	4.3.10
	Ring	$N\,(N-1)$	$r\,(N-1)$	4.3.1
	Recursive doubling	$N\log N$	$N\log r$	4.3.3
	Rabenseifner	$2N\log N$	$2N\log r$	4.3.6
Gather	*COOL-P-P*	N	$r-1$	4.3.7
	Binomial tree	$N-1$	$r-1$	4.3.2
Allgather	*COOL-P-P*	$N-r+rM_{r-1}+rM_{k-1}$	rM_{r-1}	4.3.9
	Ring	$N\,(N-1)$	$r\,(N-1)$	4.3.1
	Recursive doubling	$N\log N$	$N\log r$	4.3.3
	Bruck	$N\log N$	$N\log r+N-r$	4.3.4
Bcast	*COOL-P-P*	$1+1M_{r-1}+rM_{k-1}$	$1M_{r-1}$	4.3.8
	Binomial tree	$N-1$	$r-1$	4.3.2
Scatter	*COOL-P-P*	N	$r-1$	4.3.7
	Binomial tree	$N-1$	$r-1$	4.3.2

4.4.3 Network Load

We compare the total network load generated by the different operation-pattern combinations. Table 4.4 presents the total load generated and the load on the network across racks. It suffices to compare the various entries in Table 4.4 asymptotically. The results indicate that *COOL-P-P* reduces the network load by a factor of 2 to k in most cases for both the total network load and the load across racks. This significant reduction in the network load is a result of three factors: the explicit control of communication across racks, the use of the *parallel* pattern (which is highly efficient for small groups) and the use of network-optimal paths for multicast messages. Multicasting is used in three operations: MPI_Bcast, MPI_Allreduce and MPI_Allgather.

The binomial tree pattern performance is comparable to *COOL-P-P* performance for the MPI_Reduce, MPI_Scatter and MPI_Gather operations, whereas it doubles the network load in MPI_Bcast. Nevertheless, the binomial tree pattern takes $O(\log N)$ more steps to complete. Because the binomial tree pattern has different best rankings for different operations, achieving the best binomial tree performance for all operations is infeasible in classical implementations.

4.4.4 Process Load

The process load is the number of messages each process sends or receives during a single step. It is indicative of the amount of communication and computation required on each process per step. The *parallel* pattern (Section 3.2) completes a collective operation in a single step at the cost of an increased load on the leader processes. Hence, the parallel pattern does not scale well for large groups because of the high process load on the root process. Table 4.5 shows the maximum load generated on a process at any given step of the operation. Since not all processes have the same role in *COOL*, rack leaders see a higher load than other processes. Table 4.5 indicates that the maximum load on a leader does not exceed r (the number of racks) or k (the number of nodes in a rack) messages in any step. We argue that this amount of concurrent communication and computation

Table 4.4: Network load analysis. This table presents the total network load and the network load across racks.

Operation	Pattern	Network Load		Section
		Total	Across racks	
Reduce	*COOL-P-P*	$2N + 2r - 2$	$2r - 1$	4.3.7
	Binomial tree	$2N + 2r - 4$	$2r - 2$	4.3.2
	Rabenseifner	$2N \log N + 2N \log r + 2N + 2r - 4$	$2N \log r + 2r - 2$	4.3.5
Allreduce	*COOL-P-P*	$3N + 2r^2 - 2r$	r^2	4.3.9
	COOL-B-R	$3N + 4r \log r - 2r$	$2r \log r$	4.3.10
	Ring	$2(N + r)(N - 1)$	$2r(N - 1)$	4.3.1
	Recursive doubling	$2N \log N + 2N \log r$	$2N \log r$	4.3.3
	Rabenseifner	$4N \log N + 4N \log r$	$4N \log r$	4.3.6
Gather	*COOL-P-P*	$2N + 2r - 2$	$2r - 1$	4.3.7
	Binomial tree	$2N + 2r - 4$	$2r - 2$	4.3.2
Allgather	*COOL-P-P*	$3N + 2r^2 - 2r$	r^2	4.3.9
	Ring	$2(N + r)(N - 1)$	$2r(N - 1)$	4.3.1
	Recursive doubling	$2N \log N + 2N \log r$	$2N \log r$	4.3.3
	Bruck	$2N \log N + 2N \log r + 2N - 2r$	$2N \log r + 2N - 2r$	4.3.4
Bcast	*COOL-P-P*	$N + 2r + 2$	r	4.3.8
	Binomial tree	$2N + 2r - 4$	$2r - 2$	4.3.2
Scatter	*COOL-P-P*	$2N + 2r - 2$	$2r - 2$	4.3.7
	Binomial tree	$2N + 2r - 4$	$2r - 2$	4.3.2

can still be effectively performed in modern cloud environments since k and r are typically small (a few tens). For instance, a configuration of $k = 32$, $r = 32$ and 32 processes per node can support an MPI application with $32K$ processes, yet any collective operation will finish in three steps with a maximum load on rack leaders not exceeding 32 messages. Nevertheless, if the amount of communication and computation that leaders perform in the *parallel* pattern becomes a concern, implementers of *COOL* can choose a different pattern (e.g., binomial tree) for communication within a rack or across rack leaders (discussed in Section 4.5).

Table 4.5: Process load analysis. This table presents the maximum load on processes at any step in the operation.

Operation	Pattern	Max. Proc. Load		Section
		Leader	Other	
Reduce	*COOL-P-P*	$\max(r-1, k-1)$	1	4.3.7
	Binomial tree	1		4.3.2
	Rabenseifner	2		4.3.5
Allreduce	*COOL-P-P*	$\max(r, k-1)$	1	4.3.9
	COOL-B-R	2	1	4.3.10
	Ring	2		4.3.1
	Recursive doubling	2		4.3.3
	Rabenseifner	2		4.3.6
Gather	*COOL-P-P*	$\max(r-1, k-1)$	1	4.3.7
	Binomial tree	1		4.3.2
Allgather	*COOL-P-P*	$\max(r, k-1)$	1	4.3.9
	Ring	2		4.3.1
	Recursive doubling	2		4.3.3
	Bruck	2		4.3.4
Bcast	*COOL-P-P*	1	1	4.3.8
	Binomial tree	1		4.3.2
Scatter	*COOL-P-P*	$\max(r-1, k-1)$	1	4.3.7
	Binomial tree	1		4.3.2

4.5 COOL's Flexibility

To demonstrate *COOL*'s flexibility, we explore a design for the MPI_Allreduce operation that has lower network and process load than *COOL-P-P*. If we exclude the *parallel* pattern, because it increases the load on leader processes, Table 4.3 shows that the binomial tree pattern is the best pattern for the MPI_Reduce operation and recursive doubling is the best pattern for MPI_Allreduce. We combine these two patterns to create the *COOL*-Binomial-tree-Recursive-doubling (*COOL-B-R*) pattern. *COOL-B-R* uses the binomial tree pattern to reduce the values in a rack and recursive doubling to reduce the values

across rack leaders. Finally, the rack leaders multicast the final result to all processes in their rack.

Our analysis of *COOL-B-R* (Section 4.3.10) indicates that this composition is a middle ground between the classical recursive doubling and *COOL-P-P* pattern. Compared with *COOL-P-P*, *COOL-B-R* increases the number of steps to $\log N + 1$ total steps with $\log r$ of them using the core network (Table 4.2) and increases the number of messages exchanged across racks by a factor of $\log r$ (Table 4.3). On the other hand, *COOL-B-R* reduces the network load across racks by a factor of $r/2 \log r$ (Table 4.4) and the load on the leader processes to only two messages (Table 4.5).

This example demonstrates *COOL*'s flexibility. This flexibility facilitates the selection of various communication patterns that better fit each step of the collective operation. For instance, *COOL-B-R* trades more steps for lighter network and process load.

4.6 Summary

COOL's small subgroups (of processes within a node, nodes within a rack and rack leaders in the data center) allow the usage of the *parallel* pattern. Our evaluation shows that the *parallel* pattern can provide significant benefits: *COOL-Parallel-Parallel* composition completes any collective operation in three steps, reduces the cross-rack communication and also reduces the network load. These improvements come at the cost of an increase in load on leader processes.

Furthermore, our evaluation demonstrates that *COOL* is flexible. This flexibility allows implementers of *COOL* the ability to explore the performance/overhead trade-offs present in different communication patterns. For instance, the *COOL* implementation for MPI_Allreduce with the *parallel* pattern within and across racks (*COOL-P-P*) reduces the number of steps and the number of messages but increases the load on leader processes. Alternatively, combining binomial tree and recursive doubling patterns (*COOL-B-R*) reduces the load on the leader processes and the network load but increases the number of steps and the number of messages. Unlike *COOL*, classical communication patterns do

not provide the opportunity to explore these trade-offs or select the best pattern for every level of the data center infrastructure.

Chapter 5

Related Work

5.1 Optimizing MPI

Many previous efforts have focused on optimizing MPI collective operations. MPICH [17], Nemesis [8] and hierarchical collectives [45] optimize collective operations between processes collocated on the same node using node-local cache and shared memory. *COOL* uses similar optimizations to optimize communication between processes on the same node. Additionally, Thakur et al. [41] present Rabenseifner's pattern which is one of the early examples that combined multiple collective communication patterns to optimize MPI_Reduce and MPI_Allreduce for large data sizes. We discuss this in more detail in Section 2.3.4. *COOL* aims to reduce cross-rack communication overhead by combining multiple collective communication patterns between and within racks. Furthermore, Mirsadeghi et al. [1] propose a heuristic to dynamically reorder process ranks to match the operation's communication patterns with the supercomputer's network architecture. Our evaluation indicates that this approach is insufficient, by showing that *COOL* brings significant performance gains compared to the classical communication patterns even if they use the best ranking.

The closest efforts to our work [3] explores a hierarchical design of MPI_Reduce and MPI_Allreduce, where processes are divided into equal-sized subgroups based on their process ranks. Then the reduce operation is performed in each of these subgroups, then

across subgroups. While this previous work offers an improvement in performance over the ring pattern, it has a performance comparable with the binomial tree pattern and Rabenseifner's pattern and they only cover a small subset of collective operations. In addition, their method for constructing subgroups does not take into account the data center's network architecture since process ranks are assigned to processes randomly [30], which results in a large amount of cross-rack traffic. Another close effort is the recent preliminary exploration proposing the use of SDN capabilities to optimize specific collective operations on fat-tree interconnects [39]. In that work, for each call of the MPI_Bcast operation, the SDN controller builds a shortest-path delivery tree from the root process to all receiving processes then sets up the required flow entries on each SDN switch. In contrast with *COOL*, this effort focuses on optimizing a single collective operation (MPI_Bcast) without considering a general approach for MPI collective operations in the cloud. In addition, their approach requires setting up delivery trees for each call of the MPI_Bcast operation, which is an unnecessary cost in comparison to *COOL*'s approach of setting up multicast delivery trees for each subgroup (within and across racks) that can be reused for further invocations of the MPI_Bcast operation or other collective operations.

In summary, most previous efforts are not optimized for data center network architectures or just focus on optimizing a small subset of the collective operations (e.g., MPI_Bcast, MPI_Reduce or MPI_Allreduce) without considering a generic approach for MPI collectives in the cloud, which results in a high cross-rack traffic. *COOL* on the other hand is a generic approach for data centers' network architectures that fits a larger set of collective operations.

5.2 Rack Awareness

Several studies explore building rack-aware systems. For example, location-awareness have been suggested in the context of distributed file systems [14], data processing engines [34], big data applications [24] and message oriented middleware [42]. For instance, HDFS [14], a distributed file system, uses information about network topology to place block replicas

on different racks. This provides data availability in the event of a network switch failure or partition within the cluster. Additionally, read and write requests are directed to replicas on the same or nearest rack in order to improve the network traffic while reading/writing an HDFS file.

Moreover, Purlieus [34], a MapReduce resource allocation system, allows MapReduce virtual machines to access input data and intermediate data from local or close-by physical machines by carefully placing nodes close to the input data and intermediate nodes. This allows jobs to run faster and reduces network overhead. Purlieus discovers the network topology by analyzing network flows. Similarly, Tashi [24] is a cluster management system designed particularly for enabling cloud computing applications to operate on repositories of big data. It allows for the placement of virtual machines closer to the data they use, improving performance.

In Kafka [42], an example of Message Oriented Middleware (MOM), replicas of the same stream can be configured to be replicated to brokers on different racks in order to tolerate failures of entire racks and to minimize data loss. Another advantage of rack awareness in Kafka is that it ensures balanced throughput across racks, since the algorithm used to assign replicas to brokers ensures that the number of leaders per broker will be constant, regardless of how brokers are distributed across racks.

In summary, several applications use rack-awareness for fault tolerance and to improve system performance by balancing load between racks and minimizing inter-rack network traffic, or by placing nodes closer to the data they use. *COOL* also aims to minimize inter-rack network traffic, but it achieves that by grouping processes based on their placement within nodes, within racks and across racks, then choosing adequate communication patterns at each level. To the best of our knowledge, this is the first effort to consider a rack-aware MPI design.

5.3 Network-accelerated Systems

Recent projects have used SDN capabilities to provide load balancing [12, 18, 43], seamless virtual machine migration [28] and access control [32] and to improve system security, virtualization and network efficiency [25]. NetCache [23] implements a caching service in a single switch. The controller tracks the most popular objects and updates the cached objects. SwitchKV [27] uses SDN's capabilities to route client requests to the caching node serving the key. A central controller fills the forwarding rules to invalidate routes for objects that are being modified and installs routes for newly cached objects.

A number of recent efforts use SDN's capabilities to optimize data replication protocols. Speculative Paxos [35] builds a mostly ordered multicast primitive and uses it to optimize the multi-Paxos consensus protocol. Network-ordered Paxos (NOPaxos) [26] utilizes modern network capabilities to order multicast messages and add a unique sequence number to every client request. NOPaxos uses these sequence numbers to serialize operations and to detect packet loss. NetPaxos [13] and NetChain [22] implement replication protocols on a group of switches. These protocols are suitable for systems that store only a few megabytes of data (e.g., 8 MB in the NetChain prototype).

COOL enables collective operation designs that better fit the data center infrastructure than classical collective operation designs. In particular, it enables the exploitation of SDN capabilities to explore the network and to build efficient network paths for multicast-based data transfers within and across racks in a similar manner to how the aforementioned efforts construct efficient multicast network paths.

Chapter 6

Conclusions and Future Work

Many science and engineering applications have turned to cloud infrastructures as a platform to perform high-performance computing (HPC) [40, 19, 37]. For HPC applications MPI is the de-facto standard, and MPI collective operations are the most I/O intensive and performance critical among MPI operations [36]. Classical MPI implementations, however, are inefficient on cloud infrastructures because they are implemented at the application layer using network-oblivious communication patterns [15] which do not differentiate between local or cross-rack communication. Hence, they do not exploit the inherent locality between processes collocated on the same node or the same rack of nodes. Consequently, they can suffer from high network overheads when communicating across racks.

In this thesis, we present *COOL*, a simple and generic approach for MPI collective operations in the cloud. *COOL* exploits the inherent locality between collocated MPI processes on the same node or in the same rack and reduces cross-rack communication when compared to existing widely-used systems like OpenMPI and MPICH. To demonstrate the feasibility of our approach, we present a system design that embodies *COOL* and describes how to implement frequently used collective operations. When compared with classical implementations (OpenMPI and MPICH), our design reduces cross-rack communication by a factor of from $\log N$ up to N (the total number of processes) for most operations and it reduces the total network load by a factor of from two up to k (the number of processes

in a rack) in most cases. The cost for this performance improvement is a small increase in the communication load on leader processes. Furthermore, our evaluation demonstrates that *COOL* is flexible. This flexibility allows implementers of *COOL* the ability to explore the performance/overhead trade-offs present in different communication patterns.

6.1 Future Work

Extending MPICH. We hope to extend MPICH to provide an implementation for collective operations using *COOL*. This would include extending MPICH with a network controller, extending the Hydra [31] management components and adding new *COOL*-based implementations to the MPI collective library.

Remaining collective operations. We would like to expand our work to include the remaining collective operations, including collective operations that exchange different sizes of data for each process and non-commutative MPI_Reduce and MPI_Allreduce operations. These operations offer new challenges that we hope to address in future work.

Exploring possible communication pattern combinations. This thesis presented a design for *COOL-B-R* which explored a trade-off between reducing the process load on leader processes and increasing the total network load. We hope to explore more of the possible collective communication pattern combinations for each phase of *COOL* and the trade-offs they incur in each collective operation.

Dynamic algorithms for choosing communication pattern combinations. We would like to explore algorithms that dynamically choose the best communication pattern combinations for each phase of *COOL* by utilizing some heuristics. For example, a different combination could be used depending on the exchanged data size, the number of processes, the number of racks, the distribution of processes across racks, the differences in latency and throughput between inter-rack and intra-rack networks and user-defined constraints or preferences.

6.2 Concluding Remarks

We believe that cloud infrastructures are attractive for HPC applications, which usually use MPI as a standard communication interface. Existing MPI implementations are not optimized for cloud data centers because they are implemented at the application layer using network-oblivious communication patterns. We believe that designing MPI implementations to take into account the network infrastructure is important, so in this thesis we have presented the *COOL* approach for MPI collective operations. Our analysis shows that *COOL* offers significant improvements in performance compared to classical MPI implementations. We expect that implementations of *COOL* will significantly benefit the execution of MPI applications in these environments.

References

[1] Mirsadeghi H. S. Afsahi A. Topology-aware rank reordering for MPI collectives. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1759–1768, 5 2016.

[2] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2.3):265–276, March 2005.

[3] Hasanov Khalid Lastovetsky Alexey. Hierarchical redesign of classic MPI reduction algorithms. *The Journal of Supercomputing*, 73:713–725, February 2017.

[4] Z. AlSader, M. Alfatafta, and S. Al-Kiswany. COOL: A cloud-optimized structure for MPI collective operations. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 746–753, July 2018.

[5] Amazon. High performance computing. https://aws.amazon.com/hpc/. Accessed: 2019-12-01.

[6] L. A. Barroso, J. Clidaras, and U. Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2013.

[7] 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*, 8(11):1143–1156, Nov 1997.

[8] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, volume 1, pages 10 pp.–530, May 2006.

[9] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, Nov 2011.

[10] Cisco Systems, Inc. Data center: Load balancing data center services SRND. https://learningnetwork.cisco.com/docs/DOC-3438, March 2004. Accessed: 2019-12-01.

[11] Cray. Cray XC40^{TM} series specifications. https://www.cray.com/sites/default/files/resources/cray_xc40_specifications.pdf. Accessed: 2018-01-28.

[12] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. Strata: High-performance scalable storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 17–31, Santa Clara, CA, 2014. USENIX.

[13] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[14] Borthakur Dhruba. HDFS architecture guide. *Hadoop Apache Project*, 53, 2008.

[15] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S.

Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[16] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, pages 128–135, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[17] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22(6):789–828, September 1996.

[18] Nikhil H, Mario Flajslik, Srini Seetharaman, Ramesh Johari, Nick Mckeown, Deutsche Telekom R, and D Lab Usa. Aster*x: Load-balancing as a network primitive, 2010.

[19] Qiming He, Shujia Zhou, Ben Kobler, Dan Duffy, and Tom McGlynn. Case study for running HPC applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, page 395–401, New York, NY, USA, 2010. Association for Computing Machinery.

[20] IEEE. IEEE standard for local and metropolitan area networks - station and media access control connectivity discovery. *IEEE Std 802.1AB-2016 (Revision of IEEE Std 802.1AB-2009)*, pages 1–146, March 2016.

[21] Pamela Lembke James Milano. *IBM system Blue Gene solution: Blue Gene/Q hardware overview and installation planning*. IBM Redbooks, 2013.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In

15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 35–49, Renton, WA, April 2018. USENIX Association.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 121–136, New York, NY, USA, 2017. ACM.

[24] Michael A. Kozuch, Michael P. Ryan, Richard Gass, Steven W. Schlosser, David O'Hallaron, James Cipar, Elie Krevat, Julio López, Michael Stroucken, and Gregory R. Ganger. Tashi: Location-aware cluster management. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 43–48, New York, NY, USA, 2009. ACM.

[25] A. Lara, A. Kolasani, and B. Ramamurthy. Network innovation using OpenFlow: A survey. *IEEE Communications Surveys Tutorials*, 16(1):493–512, First 2014.

[26] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, Savannah, GA, November 2016. USENIX Association.

[27] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, March 2016. USENIX Association.

[28] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. XvMotion: Unified virtual machine migration over long distance. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 97–108, Philadelphia, PA, June 2014. USENIX Association.

[29] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innova-

tion in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[30] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 2015. Version 3.1.

[31] MPICH. Hydra process management framework. https://wiki.mpich.org/mpich/index.php/Hydra_Process_Management_Framework, June 2014. Accessed: 2019-12-01.

[32] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic access control for enterprise networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 11–18, New York, NY, USA, 2009. ACM.

[33] The Open Networking Foundation. *Open networking foundation: OpenFlow switch specification*, 6 2012. Version 1.3.0.

[34] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware resource allocation for MapReduce in a cloud. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2011.

[35] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[36] Rolf Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 35–42, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[37] J. J. Rehr, F. D. Vila, J. P. Gardner, L. Svec, and M. Prange. Scientific computing in the cloud. *Computing in Science and Engg.*, 12(3):34–43, May 2010.

[38] Rabenseifner Rolf. Automatic MPI counter profiling. In *42nd CUG Conference*, 2000.

[39] Morimoto H. Dashdavaa K. Takahashi K. Kido Y. Date S. Shimojo S. Design and implementation of SDN-enhanced MPI broadcast targeting a fat-tree interconnect. In *2017 International Conference on High Performance Computing Simulation (HPCS)*, pages 252–258, 7 2017.

[40] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu. Understanding the performance and potential of cloud computing for scientific applications. *IEEE Transactions on Cloud Computing*, 5(2):358–371, April 2017.

[41] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[42] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with Apache Kafka. *Proc. VLDB Endow.*, 8(12):1654–1655, August 2015.

[43] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[44] Katherine Yelick, Susan Coghlan, and Richard Shane Canon Brent Draney. The Magellan report on cloud computing for science. *US Department of Energy, Washington DC, USA, Tech. Rep*, 2011.

[45] Hao Zhu, David Goodell, William Gropp, and Rajeev Thakur. Hierarchical collectives in MPICH2. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 325–326, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.