COOL: A Cloud-Optimized Structure for MPI Collective Operations

Mohammed Alfatafta, Zuhair AlSader, Samer Al-Kiswany
Cheriton School of Computer Science
University of Waterloo, Canada
{maaalfat, zalsader, alkiswany}@uwaterloo.ca

***Abstract*— We present *COOL*, a simple and generic structure for MPI collective operations. *COOL* enables highly efficient designs for all collective operations in the cloud. We then present a system design based on *COOL* that implements frequently used collective operations. Our design efficiently uses the intra-rack network while minimizing cross-rack communication, thus improving the application performance and scalability. We use recent software-defined networking capabilities to build optimal network paths for I/O intensive collective operations. Our analytical evaluation shows that our design imposes the least possible network overhead across racks. Furthermore, when compared with OpenMPI and MPICH, our design reduces the number of steps to only three, decreases the number of exchanged messages by a factor of *N*, the total number of processes, and reduces the network load by up to an order of magnitude. These significant improvements come at the cost of a modest increase in the computation load on a few processes.**

Keywords – MPI, collective operations, communication patterns, cloud, software-defined networking

I. INTRODUCTION

Cloud infrastructure is increasingly being adopted as a platform for high-performance computing (HPC) science and engineering applications [1, 2, 3, 4, 5, 6], with major research organizations embracing the new platform [7, 8, 9] and cloud providers offering clusters targeting HPC applications [10, 11]. For HPC applications, the message passing interface (MPI) [12] and its classical implementations (e.g., OpenMPI [13] and MPICH [14]) remain a popular communication middleware. Among MPI operations, collective operations (e.g., broadcast, reduce, scatter, and gather) are the most I/O intensive and performance critical.

Although classical MPI implementations can be deployed at cloud data centers, they are inefficient [7]. Historically, MPI applications use large-scale supercomputer machines that are customized to support MPI collective operations (e.g., IBM BlueGene [15] and Cray XC40 [16]). Supercomputers use over-provisioned special network topologies (e.g., 3D torus [17], 5D torus [18], and Dragonfly [16]), and have 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 [17, 19]. 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 [20] 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 [20]. 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 [13, 14]. For instance, they propagate a broadcast message between processes following a tree pattern, or perform a reduction following a ring pattern. These patterns do not differentiate between local or cross-rack communications, and hence do not exploit the inherent locality between collocated processes on the same node or the same rack of nodes. Consequently, they generate high overhead on the network across racks.

In this paper, we present the *c*loud-*o*ptimized c*oll*ective structure (*COOL*) for collective operations. *COOL* is a simple yet generic structure, 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 optimal for it. This approach provides the MPI designer with explicit control over the communication performed within a node, within a rack, and across racks.

Unlike classical implementations, *COOL* is flexible. Communication patterns typically present a tradeoff between the number of steps needed, the number of messages sent, and the generated network and process loads. *COOL* allows exploring this tradeoff by combining more than one pattern and allows selecting 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 detail the design for 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 [21, 22] 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 networking (SDN) [23] 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 impose the least possible 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 generated load on the network and nodes. Our evaluation reveals that *COOL*-based design brings significant performance gains: it completes all operations in three steps, it reduces the number of messages across racks to the bare minimum, 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 a cost: a modest increase in the load of leader processes.

The rest of this paper is organized as follows. In Section II we present the classical design of the most frequently used MPI collective operations. We present the *COOL* structure and a system design that embodies it in Section III. We present the analytical evaluation in Section IV. We discuss related work in Section V, and conclude in Section VI.

II. BACKGROUND

In this section, we present frequently used collective operations, their communication patterns, and an overview of the modern data center architecture.

A. Most Frequently Used Collective Operations

Collective operations are the most network-intensive operations in MPI; they involve communicating with all the processes in a communication group, typically over many steps. The following list enumerates frequently used collective operations. Characterization studies of MPI applications [21, 22] indicate that the following operations consume more than 65% of the CPU time that all MPI collective operations use.

- MPI_Reduce: applies an aggregation operation (e.g., summation and multiplication) 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.
- MPI_Allgather: similar to MPI_Gather, but the final array is available in all processes in a group.
- MPI_Bcast: broadcasts a message from one process to all processes in a group.
- MPI_Scatter: chunks and distributes an array from one process to all processes in a group.

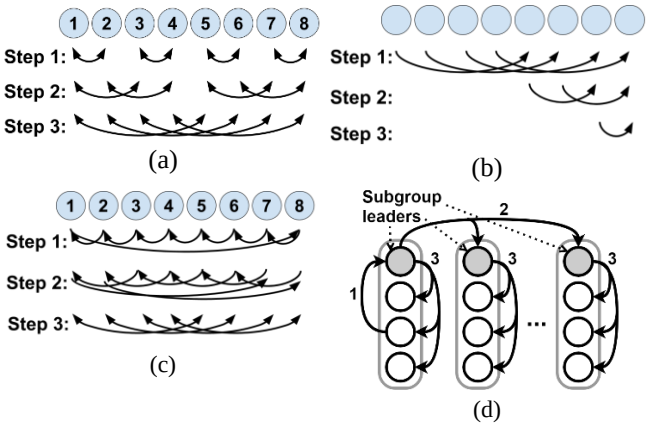

Figure 1. Collective communication patterns: (a) recursive doubling, (b) binomial tree, (c) Bruck's, and (d) *COOL* with the parallel pattern. Circles represent processes, numbers indicate the processes ranks, and arrows represent the direction of communication. Double arrows indicate that the two processes exchange messages. Numbers on arrows in (d) represent the three steps of a *COOL-based* MPI_Bcast.

B. Common Communication Patterns

Classical MPI implementations implement collective operations at the application layer using network-oblivious communication patterns. Typically, MPI implementations have multiple implementations for the same operation, with each using a different communication pattern [13, 14]. 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 tree-based topologies. Communication patterns use logical addresses (a.k.a. ranks): consecutive numbers that identify processes within a group. The following is a list of common communication patterns used in OpenMPI and MPICH:

- *Recursive doubling* 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 ranks away (Figure 1.a). N messages are sent per step. Recursive doubling uses $N \log N$ messages to complete.
- *Ring* 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. The ring pattern requires $N(N-1)$ messages to complete.
- *Binomial tree* 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 1.b), processes are divided into two halves; one half sends the data it 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. Binomial tree pattern takes $N-1$ messages to complete.
- *Rabenseifner's pattern* [24] is used in MPI_Reduce and MPI_Allreduce. Rabenseifner's pattern combines two

patterns: first, it uses the recursive halving pattern (similar to recursive doubling), and then, it uses the binomial tree pattern for MPI_Reduce, or the recursive doubling pattern for MPI_Allreduce. 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.

- *Bruck's pattern* [25] is used in MPI_Allgather. It takes $\log N$ steps, and in every step i, every process p sends its data and all of the data it has received so far to the process with rank $(p - 2^i)\ mod\ N$ (Figure 1.c). N messages are sent in every step. This pattern takes $N\log N$ messages to complete.

C. Target Deployment

Modern data centers adopt a tree-based topology [20, 26], in which nodes are organized in racks (e.g., each rack has 48 nodes) with each connected to the data center through a top of the 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 times to 10 times are common [20]. Consequently, increasing communication locality within racks is key to achieving higher performance and scalability.

Furthermore, modern data centers adopt SDN-capable switches. This new networking paradigm facilitates the external control of network operations. The OpenFlow standard API [23] provides per-packet control of network operations. Developers can use this capability to build network-optimal 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 are network oblivious, as they do not differentiate between communication within a rack or across racks, and they do not exploit the recent SDN capability to optimize the data paths for collective operations.

COOL enables collective operation designs that better fit the data center infrastructure than classical collective implementations. 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 the SDN capability to build efficient network paths for multicast-based data transfers within and across racks.

III. System Design

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

A. COOL

The *COOL* structure adopts a hierarchical approach to perform collective operations. *COOL* divides the communication group into a set of subgroups (Figure 1.d). Each subgroup has a leader process. All collocated processes 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. Subgroups are small, consisting of a few tens of processes. During the 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. 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 can choose different communication patterns within and across subgroups. This flexibility allows selecting the best pattern for each subgroup. For instance, an implementer 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 conscience pattern for the cross-rack level even if it slightly increases the number of steps.

As an example, consider the MPI_Bcast operation, and, for simplicity, assume 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 1.d). 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.

B. Parallel Pattern

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. For instance, in a *parallel* MPI_Reduce, all processes send their data to a single process that performs the reduction operation. Similarly, in a *parallel* MPI_Bcast, a single 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 deployments.

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-optimal multicast trees.

C. COOL Collective Operations

COOL is generic; it can be used to optimize all collective operations for cloud deployments. The following list presents a *COOL* design for the frequently used collective operations discussed in section II.A. Although *COOL* can employ various communication patterns at different levels, for simplicity, we present a design using the *parallel* pattern at all levels. 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, relatively, a negligible impact on performance. Hence, we assume a single process per node.

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 root process. We also present the number of messages exchanged in every operation. The system has *N* nodes, with each running a single process. The nodes are organized in *r* racks, with *r* rack leaders.

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

We note that all operations complete in only three steps, and use the network across racks only in step 2. The next section extends our analysis and compares our design with OpenMPI and MPICH implementations.

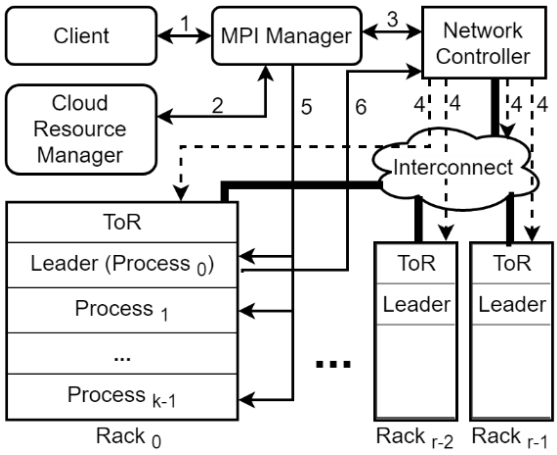

Figure 2. System Architecture. Bold lines represent network connections. Solid arrows represent communication messages. Dashed arrows represent OpenFlow control messages.

D. 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 2): an MPI manager, rack leaders, a network controller, and a *COOL* library. The MPI manager controls the MPI application lifecycle from allocating resources, to bootstrapping the MPI processes, to terminating the application. All processes are grouped into subgroups at the node, rack, and data center levels. The network controller is an OpenFlow-based controller that manages all the switches in the deployment. The network controller installs packet-forwarding rules to create a hierarchy of multicast trees for each communication group. The *COOL* library implements all collective operations as discussed in the previous subsection.

Bootstrap process. As depicted in Figure 2, when a client starts a new job, it sends the job parameters to the MPI

Table 1. An analytical comparison of *COOL*-based collective operations against classical implementations. *COOL-P-P* uses the parallel pattern for communication within and across racks. *COOL-B-R* uses the binomial tree pattern within a rack and recursive doubling pattern across racks. The analysis assumes the best ranking for classical implementations which achieves the least possible cross-rack communication.

Op.	Pattern	# of steps		Number of messages	
		Total	Across racks	Total	Across racks
Reduce	*COOL-P-P*	3	1	N	$r-1$
	Binomial tree	$\log N$	$\log r$	$N-1$	$r-1$
	Rabenseifner	$2\log N$	$2\log r$	$N\log N+N-1$	$N\log r+r-1$
Allreduce	*COOL-P-P*	3	1	$N-r+r_r+r_k$	r_r
	COOL-B-R	$\log N+1$	$\log r$	$N-r+r\log r+r_k$	$r\log r$
	Ring	$N-1$	$N-1$	$N(N-1)$	$N\,r$
	Recursive doubling	$\log N$	$\log r$	$N\log N$	$N\log r$
	Rabenseifner	$2\log N$	$2\log r$	$2N\log N$	$2N\log r$
Gather	*COOL-P-P*	3	1	N	$r-1$
	Binomial tree	$\log N$	$\log r$	$N-1$	$r-1$
Allgather	*COOL-P-P*	3	1	$N-r+r_r+r_k$	r_r
	Ring	$N-1$	$N-1$	$N(N-1)$	$N\,r$
	Recursive doubling	$\log N$	$\log r$	$N\log N$	$N\log r$
	Bruck	$\log N$	$\log N$	$N\log N$	$N\log r+N-r$
Bcast	*COOL-P-P*	3	1	r_k+1_r+1	1_r
	Binomial tree	$\log N$	$\log r$	$N-1$	$r-1$
Scatter	*COOL-P-P*	3	1	N	$r-1$
	Binomial tree	$\log N$	$\log r$	$N-1$	$r-1$

Table 2. Analytical model parameters.

Symbol	Description
N	Total number of processes. $N=k.r$
r	Number of racks
k	Number of nodes within each rack
x_y	x messages are multicasted to y recipients.

manager (step 1 in Figure 2). The manager allocates a number of nodes (2). Then, the network controller configures the network for the MPI job (3). The controller first discovers the network topology connecting the allocated nodes using the Link Layer Discovery Protocol (LLDP) protocol [27]. The discovery step identifies the racks, assigns them serial identification (ID) numbers, and discovers which nodes are in each rack (4). Then, the 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 the node and rack leaders, as well as the rank and rack ID of all other processes.

Group creation. When an MPI application creates a new communication group (6) the leader of the first rack (i.e., rack 0) informs the controller of the new group. As in the bootstrap process, the 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 controller also creates, using OpenFlow, a multicast tree in every rack, and creates a multicast tree for rack leaders across racks.

IV. Evaluation

In this section, we analytically compare *COOL* collective operations with OpenMPI and MPICH. We focus our evaluation on four metrics: the number of steps an operation takes, 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.

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*) that uses the *parallel* pattern within and across racks. Section IV.E discusses a *COOL*-based design that uses the binomial tree and the recursive doubling patterns.

A. Operation Latency

Each step includes a communication phase, which consists of a concurrent exchange of data with one or more processes, and a computation phase for MPI_Reduce and MPI_Allreduce. Table 1 summarizes our results (section II.B and section III.C present a sketch of our analysis). Table 1 shows the total number of steps that various implementations of the collective operations require, and shows the number of steps that use the inter-rack links (Table 2 explains the parameters used in Table 1). Classical implementations take from $\log N$ to N steps to complete with $\log r$ to N of the steps using the network across racks. *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, as it confines the cross-rack communication to a single step.

Obviously, a step in the *parallel* pattern has a higher 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 IV.E. Nevertheless, if the amount of communication and computation that leaders perform becomes a concern, implementers can reduce this overhead by choosing a different pattern (e.g., binomial tree) for communication within or across racks (Section IV.E presents one example).

B. Exchanged Messages

Table 1 shows the number of messages exchanged in total and across racks. Section II.B and section III.C present a sketch of our analysis. 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 fewer than *r* messages, which is the absolute minimum required for exchanging data between *r* racks. This impressive result is due to two design decisions: first, the cross-rack communication is only done between *r* rack leaders, and second, IP-level multicasting is used for all multicast phases in MPI_Bcast, MPI_Allreduce, and MPI_Allgather, leading to a significant reduction in the number of messages sent. Here, we count a multicast message as a single sent message, although its overhead is higher than that of a single message. The following subsection 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, something classical MPI implementations cannot do (discussed at the beginning of this section), the binomial tree pattern can minimize cross-rack communication. On the other hand, the binomial tree pattern significantly increases the number of steps that the operation takes to log *N* steps.

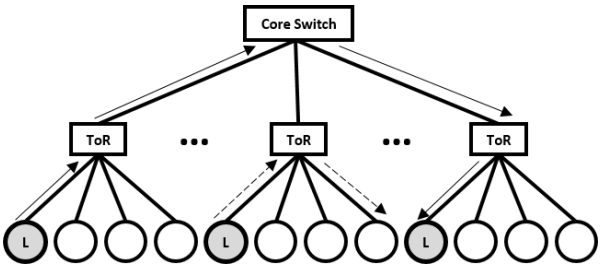

Figure 3. Network model. Tree topology with a single core switch. Solid arrows illustrate the path for a message across racks, dashed arrows illustrate the path for a message within a rack.

C. Network Load

We compare the total network load generated by the different operation-pattern combinations. 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 a single core switch connecting all racks (Figure 3). For instance, a message from one node to another node in the same rack uses two links (dashed arrows in Figure 3), whereas a message across racks traverses four links (solid arrows in Figure 3). Our analysis is conservative; a typical data center network is more complex. Hence, a single message will traverse more core links than in our model, which amplifies the difference between *COOL* and the classical implementations.

Table 3. Load analysis. The table presents the total network load, network load across racks, and the maximum load on processes at any step in the operation. *COOL-P-P* uses the parallel pattern for communication within and across racks. *COOL-B-R* uses binomial tree pattern within a rack and recursive doubling pattern across racks.

Op.	Pattern	Network Load		Max. Proc. Load	
		Total	Across racks	Leader	Other
Reduce	*COOL-P-P*	$2N + 2r - 2$	$2r - 1$	$\max(r, k)$	1
	Binomial tree	$2N + 2r - 4$	$2r - 2$	1	
	Rabenseifner	$2(N \log N + N \log r + N + r - 2)$	$2N \log r + 2r - 2$	2	
Allreduce	*COOL-P-P*	$3N + 2r^2 - 2r$	r^2	$\max(r, k)$	1
	COOL-B-R	$3N + 4rlogr - 2r$	$2r \log r$	2	1
	Ring	$2(N-1)(N+r)$	$2r\,(N-1)$	2	
	Recursive doubling	$2N \log N + 2N \log r$	$2N \log r$	2	
	Rabenseifner	$4N \log N + 4N \log r$	$4N \log r$	2	
Gather	*COOL-P-P*	$2N + 2r - 2$	$2r - 1$	$\max(r, k)$	1
	Binomial tree	$2N + 2r - 4$	$2r - 2$	1	
Allgather	*COOL-P-P*	$3N + 2r^2 - 2r$	r^2	$\max(r, k)$	1
	Ring	$2(N-1)(N+r)$	$2r\,(N-1)$	2	
	Recursive doubling	$2N \log N + 2N \log r$	$2N \log r$	2	
	Bruck	$2N(\log N + \log r) + 2N - 2r$	$2N \log r + 2N - 2r$	2	
Bcast	*COOL-P-P*	$N + 2r + 2$	r	1	1
	Binomial tree	$2N + 2r - 4$	$2r - 2$	1	
Scatter	*COOL-P-P*	$2N + 2r - 2$	$2r - 2$	$\max(r, k)$	1
	Binomial tree	$2N + 2r - 4$	$2r - 2$	1	

Table 3 presents the total load generated and the load on the network across racks. It suffices to compare the various entries in Table 3 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 in MPI_Reduce, MPI_Scatter, and MPI_Gather, 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.

D. Process Load

The *parallel* pattern 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. Table 3 shows the maximum load generated on a process at any given step of the operation. The load is defined as the number of messages sent and received in a step. As not all processes have the same role in *COOL*, rack leaders see a higher load than other processes do. Table 3 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 can still be effectively performed by modern cloud machines since *k* and *r* are typically small (a few tens). For instance, a configuration of $k = 32, r = 32$, with 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 can choose a different pattern (e.g., binomial tree) for communication within a rack or across rack leaders (discussed next).

E. COOL 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*. Excluding the parallel pattern, which increases the load on leader processes, Table 1 shows that the binomial tree pattern is the best pattern for the reduce operation, and recursive doubling is the best pattern for allreduce. We combine these two patterns to create the *COOL*-Binomial-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* (Table 1 and Table 3) 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 $logN + 1$ total steps with *log r* of them using the core network, and increases the number of messages exchanged across racks by a factor of *log r* (Table 1). On the other hand, *COOL-B-R* reduces the network load across racks by a factor or *r/2log r* and the load on the leader processes to only two messages (Table 3).

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.

F. Summary

COOL small subgroups allow the usage of the *parallel* pattern. Our evaluation shows that the *parallel* pattern can bring significant benefits: *COOL-Parallel-Parallel* composition completes any collective operation in three steps, reduces the cross-rack communication to the absolute minimum, and sizably reduces the network load. These improvements come at the cost of a modest increase in load on leader processes.

Furthermore, our evaluation demonstrates that *COOL* is flexible. This flexibility allows implementers to explore the performance/overhead tradeoffs present in communication patterns. For instance, the MPI_Allreduce, with the *parallel* pattern within and across racks, reduces the number of steps and the number of messages, but increases the load on leader processes. Alternatively, combining binomial tree and recursive doubling 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 tradeoffs or select the best pattern for every level of the data center infrastructure.

V. Related Work

Many previous efforts focused on optimizing MPI collective operations. MPICH [14], Nemesis [28], and hierarchical collectives [29] optimize collective operations between processes collocated on the same node using node-local cache and shared memory. COOL uses similar optimizations to optimize node-local groups. Furthermore, Mirsadeghi et al. [30] propose a heuristic to dynamically reorder process ranks to match the operation's communication pattern with the supercomputer architecture. Our evaluation indicates that this approach is not optimal, our evaluation shows 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 [31, 29] explore a hierarchical design of some collective operations, such as MPI_Reduce and MPI_Allreduce. However, these explorations do not cover all of the collective operations, and they are not optimized for the data center architecture. Another close effort is the recent preliminary exploration of using SDN capabilities to optimize specific collective operations on fat-tree interconnects [32]. This effort focuses on optimizing a single collective operation (e.g., broadcast and reduce) without considering a comprehensive architecture for MPI collectives in the cloud.

Several studies explore building rack-aware systems. For example, location-awareness have been suggested in the context of distributed file systems [33], data processing engines [34], and big data applications [35]. However, to the best of our knowledge, this is the first effort to consider a rack-aware MPI design.

VI. Conclusion and Future Work

We present *COOL*, a simple yet generic structure for MPI collective operations in the cloud. *COOL* exploits the inherent access locality between collocated MPI processes on the same node or in the same rack and reduces cross-rack communication. To demonstrate the feasibility of our approach, we present a system design that embodies *COOL* and implements frequently used collective operations. Our

analytical evaluation indicates that our design reduces communication across racks to the bare minimum. Compared with classical implementations, our design reduces cross-rack communication by a factor of *log N* to *N* for most operations, and it reduces the network load by a factor of two to *k* in most cases. The cost for this performance improvement is a modest increase in the communication load on leader processes.

We are currently extending MPICH to provide an implementation for collective implementations using *COOL*. Our current effort focuses on extending MPICH with a network controller, extending the Hydra [36] management components, and adding new *COOL*-based implementations to the MPI collective library.

VII. REFERENCES

[1] Q. He, S. Zhou, B. Kobler, D. Duffy and T. McGlynn, "Case study for running HPC applications in public clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010.

[2] J. J. Rehr, F. D. Vila, J. P. Gardner, L. Svec and M. Prange, "Scientific computing in the cloud," *Computing in science & Engineering*, vol. 12, pp. 34-43, 2010.

[3] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. 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*, vol. 5, pp. 358-371, 2017.

[4] M. Rak, M. Turtur, U. Villano and L. Pino, "A Portable Tool for Running MPI Applications in the Cloud," in *Intelligent Networking and Collaborative Systems (INCoS), 2014 International Conference on*, 2014.

[5] M. Parashar, M. AbdelBaky, I. Rodero and A. Devarakonda, "Cloud paradigms and practices for computational and data-enabled science and engineering," in *Computing in Science & Engineering*, 2013.

[6] R. da Rosa Righi, V. F. Rodrigues, C. A. Da Costa, G. Galante, L. C. E. De Bona and T. Ferreto, "Autoelastic: Automatic resource elasticity for high performance applications in the cloud," in *IEEE Transactions on Cloud Computing*, 2016.

[7] K. Yelick, S. Coghlan, B. Draney, R. S. Canon and others, "The Magellan report on cloud computing for science," *US Department of Energy, Washington DC, USA, Tech. Rep*, 2011.

[8] J. Bashor, "Can Cloud Computing Address the Scientific Computing Requirements for DOE Researchers? Well, Yes, No and Maybe," nersc, [Online]. Available: http://www.nersc.gov/news-publications/nersc-news/nersc-center-news/2012/can-cloud-computing-address-the-scientific-computing-requirements-for-doe-researchers-well-yes-no-and-maybe/. [Accessed 8 May 2018].

[9] E. TAYLOR, "DOE's cloud computing project wins "Best Use of HPC in the Cloud" award," argonne national laboratory, [Online]. Available: https://www.anl.gov/articles/does-cloud-computing-project-wins-best-use-hpc-cloud-award. [Accessed 8 May 2018].

[10] "Amazon High Performance Computing (HPC)," [Online]. Available: https://aws.amazon.com/hpc/. [Accessed 28 01 2018].

[11] "The Power of Bare Metal, the Flexibility of Cloud," Penguin Computing, [Online]. Available: https://www.penguincomputing.com/pod-hpc-cloud/. [Accessed 8 May 2018].

[12] M. P. I. Forum, "MPI: A Message-Passing Interface Standard," Message Passing Interface Forum, 1994.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine and others, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2004.

[14] W. Gropp, E. Lusk, N. Doss and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, vol. 22, pp. 789-828, 1996.

[15] J. Milano, P. Lembke and others, IBM system Blue Gene solution: Blue Gene/Q hardware overview and installation planning, IBM Redbooks, 2013.

[16] "Cray XC40™ Series Specifications," [Online]. Available: https://www.cray.com/sites/default/files/resources/cray_xc40_specifications.pdf. [Accessed Jan 2018].

[17] 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 and others, "Blue Gene/L torus interconnection network," *IBM Journal of Research and Development*, vol. 49, pp. 265-276, 2005.

[18] 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 *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011.

[19] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter and others, "Early evaluation of IBM BlueGene/P," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[20] L. A. Barroso, J. Clidaras and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, pp. 1-154, 2013.

[21] R. Rabenseifner, "Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512," in *Proceedings of the message passing interface developer's and user's conference*, 1999.

[22] R. Rabenseifner, "Automatic mpi counter profiling," in *42nd CUG Conference*, 2000.

[23] O. S. Specification, *Open Networking Foundation (ONF). Technical Specification*, 2015.

[24] R. Thakur, R. Rabenseifner and W. Gropp, "Optimization of collective communication operations in MPICH," *The International Journal of High Performance Computing Applications*, vol. 19, pp. 49-66, 2005.

[25] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1143--1156, 1997.

[26] CISCO, "Data Center: Load Balancing Data Center Services SRND," March 2004. [Online]. Available: https://learningnetwork.cisco.com/docs/DOC-3438. [Accessed 2018].

[27] *IEEE Standard for Local and Metropolitan Area Networks– Station and Media Access Control Connectivity Discovery,pp. 1–204, 2009.*

[28] D. Buntinas, G. Mercier and W. Gropp, "Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, 2006.

[29] H. Zhu, D. Goodell, W. Gropp and R. Thakur, "Hierarchical collectives in mpich2," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2009.

[30] S. H. Mirsadeghi and A. Afsahi, "Topology-Aware Rank Reordering for MPI Collectives," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[31] K. Hasanov and A. Lastovetsky, "Hierarchical redesign of classic MPI reduction algorithms," *The Journal of Supercomputing*, vol. 73, no. 2, pp. 713-725, 1 February 2017.

[32] H. Morimoto, K. Dashdavaa, K. Takahashi, Y. Kido, S. Date and S. Shimojo, "Design and Implementation of SDN-enhanced MPI Broadcast Targeting a Fat-Tree Interconnect," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, 2017.

[33] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Project*, vol. 53, 2008.

[34] B. Palanisamy, A. Singh, L. Liu and B. Jain, "Purlieus: locality-aware resource allocation for MapReduce in a cloud," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011.

[35] M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. O'Hallaron, J. Cipar, E. Krevat, J. Lopez, M. Stroucken and G. R. Ganger, "Tashi: location-aware cluster management," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, 2009.

[36] "Hydra Process Management Framework," [Online]. Available: https://wiki.mpich.org/mpich/index.php/Hydra_Process_Management_Framework. [Accessed Feb 2018].