Network Performance Aware MPI Collective Communication Operations in the Cloud

Yifan Gong, Bingsheng He, Jianlong Zhong

Abstract—This paper examines the performance of collective communication operations in Message Passing Interfaces (MPI) in the cloud computing environment. The awareness of network topology has been a key factor in performance optimizations for existing MPI implementations. However, virtualization in the cloud environment not only hides the network topology information from the users, but also causes traffic interference and dynamics to network performance. Existing topology-aware optimizations are no longer feasible in the cloud environment. Therefore, we develop novel network performance aware algorithms for a series of collective communication operations including broadcast, reduce, gather and scatter. We further implement two common applications, N-body and conjugate gradient (CG). We have conducted our experiments with two complementary methods (on Amazon EC2 and simulations). Our experimental results show that the network performance awareness results in 25.4% and 28.3% performance improvement over MPICH2 on Amazon EC2 and on simulations, respectively. Evaluations on N-body and CG show 41.6% and 14.3% respectively on application performance improvement.

Index Terms—Cloud Computing, MPI, Collective Operations, Network Performance Optimizations

1 Introduction

Cloud computing has emerged as a popular computing paradigm for many distributed and parallel applications. Message Passing Interface (MPI) is a common and key software component in distributed and parallel applications, and its performance is the key factor for the network communication efficiency. This paper investigates whether and how we can improve the performance of MPI in the cloud computing environment.

Since collective communications are the most important MPI operations for the system performance [13], [14], [17], this paper focuses on the efficiency of MPI collective communication operations. Network topology aware algorithms have been applied to optimize the performance of collective communication operations [13], [28], [26], [14], [17]. Most of the studies adopt tree-based algorithms, since the network topology is often tree-structured. The essential idea of those algorithms is to obtain the topology information with hardware or software mechanisms, and then to map the MPI processes to machines according to the underlying topology.

While topology aware algorithms work well in traditional clusters, they are no longer feasible in the cloud environment. Due to the virtualization and system management issues, the topology information is not available or not effective to use for optimization. First, virtualization hides the network topology from users. Virtualization offers a uniform interface to users, without exposing the real configurations of the underlying hardware. Second, cloud environments do not offer administrator privileges on accessing hardware and software under the virtualization layer. Such privileges are usually required when getting the network topology information [13], [28]. Due to the first and second issues, some reverse engineering techniques in topology discovery [19], [9] are not applicable in the cloud environment. Third, due to the cloud system dynamics such as virtual machine consolidation [27] and dynamic network flow scheduling [1], the static topology information is not sufficient for representing the network performance. Note, most topology inference methods [15], [25], [2] are used to identify the static network topology.

We have studied the network performance of Amazon EC2. Our studies show significant network performance unevenness (i.e., the performance varies significantly for different virtual machine pairs). Moreover, we have made a significant observation that the network performance of two virtual machines in Amazon is not symmetric. This result is consistent with the previous study on Internet network [22].

All those factors make existing topology aware algorithms unfeasible in the virtualized cloud environment, and new algorithms should be invented for network performance aware optimizations. The network performance unevenness indicates that we should schedule the communication for machine pairs with low network performance in an optimal manner in order to minimize the overall cost. On the other hand, the asymmetry of the network performance requires an advanced way of modeling and utilizing network performance characteristics. Specifically, we have identified two major problems in the MPI collective operation design. First, we need a model to capture the network performance among different machine pairs in the cloud environment. The model should facilitate network performance aware algorithms for collective operations with different message sizes. The second problem is how to take advantage of the network performance model to develop collective communication algorithms to address asymmetry.

• *Yifan Gong, Bingsheng He and Jianlong Zhong are with School of Computer Engineering, Nanyang Technological University, Singapore, 639798.*
E-mail: GONG0029@e.ntu.edu.sg, bshe@ntu.edu.sg, jzhong2@e.ntu.edu.sg

We develop the latency and bandwidth matrices to capture the network performance of all virtual machine pairs in a set of virtual machines. The *network performance matrix* for a particular message size is calculated based on the latency and bandwidth matrices. Based on the network performance matrices, we propose a *network performance hierarchy* to capture the network performance among a set of virtual machines in the cloud. In particular, we calculate the closeness of any two virtual machines, and group them according to the network performance to indicate their closeness. The grouping is performed with a hierarchical approach in order to gracefully adapt to the network performance unevenness.

Based on the network performance hierarchy, we develop our network performance aware collective communication operations. We specifically consider two categories of operations: (a) broadcast and reduce: the message size is constant for the entire operation; (b) gather and scatter: the message size increases as the communications are close to the root process. Given the network performance hierarchy, we develop novel algorithms for those operations according to the message-changing pattern. The links are carefully chosen for optimizing the data transfer time, and the communications with similar network performance are scheduled for maximized overlapping. We compare our collective operations and applications (N-body and CG) with their counterparts with the widely used MPI implementation MPICH2. We also use simulation to evaluate collective operations on a large scale of machines.

On Amazon EC2, our optimized collective operations are 13.5–38.2% faster than their counterparts in MPICH2; on simulation, our optimized collective operations are 25.4–31.2% faster than their counterparts in MPICH2 when the number of process increases from 32 to 2048. In our experiments on Amazon EC2, our network performance aware algorithm improves N-body and CG by 41.6% and 14.3%, respectively. The runtime overhead of network performance hierarchy construction is amortized with iterative MPI communications in the applications.

The rest of the paper is organized as follows. We introduce the preliminary and review the related work on cloud computing and MPI in Section 2. In Section 3, we present our empirical study of network performance in Amazon EC2. We present network performance model and algorithm design in Section 4, followed by the experimental results in Section 5. Finally, we conclude this paper in Section 6.

2 Preliminary and Related Work

In this section, we briefly introduce the preliminary and the related work that are closely related to our study.

2.1 Cloud Computing

The cloud environment has different hardware and software designs, in contrast with traditional clusters. Due to the significant scale, the network environment in the cloud differs to traditional clusters. The current cloud practice is to use the commodity switch-based tree structure to interconnect the servers [12], as illustrated on Figure 1. Machines are first grouped into *rack*s, and then racks are connected with higher-level switches. The key problem of the tree topology is the network bandwidth of any machine pairs is not uniform, depending on how the switch connecting the two machines. For example, the intra-rack bandwidth is much higher than the cross-rack bandwidth.

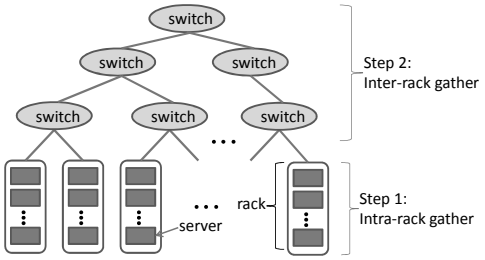


Fig. 1. Topology-aware gather in a tree-topology network

Virtualization is the enabling technology for resource sharing and consolidation in the cloud. Example virtualization systems include VMWare, Xen and Hyper-V. Virtualization hides the details on the underlying hardware. As we mentioned in Section 1, virtualization prohibits the network topology aware optimizations for MPI.

The hardware and software issues make the cloud environment different from traditional clusters. Researchers need to carefully examine the existing system optimizations and may invent new techniques to exploit the unique features of the cloud environment.

Network topology inference techniques have been investigated in the traditional environments [15], [25] and cloud environment [2]. We refer readers to a survey [4] for more details on classic techniques for network topology discovery and inference. The information given by basic diagnostic tools like *traceroute* is incomplete in the virtualized cloud like Amazon EC2. Many topology inference methods [15], [25], [2] are based on end-to-end measurements. Our notion of network performance matrix is also based on end-to-end measurements. Closely related to our study, Orchestra [8] applies flow control mechanisms to minimize two common operations (shuffle and broadcast). Different from Orchestra, our network performance optimizations are in the application layer, and are specifically designed for MPI collective operations.

2.2 MPI

MPI is a de facto standard for distributed and parallel programs running on computer clusters or supercomputers. Since MPI was first implemented in 1992, it has been implemented and optimized on different computing environments, e.g., multi-core processors [18], [20], [11], [5], wide area network [17], and Infiniband networks [13], [28]. Our idea of grouping is partially inspired by the grouping algorithms in those previous studies.

Before reviewing the related work on MPI, we first introduce several terminologies. *Communicator* is an MPI object which connects a group of processes in an MPI session. The size of the communicator is the number of processes in it, and each process is given an identifier (*rank*) 0, 1, ..., N-1, where N is the size of the communicator.

A number of network topology aware collective communication algorithms have been developed [13], [28], [26], [14], [17], [16]. They are targeted at different kinds of network architectures. Instead of relying on network performance knowledge, Burger et al. [3] optimized multicast by adapting to the current achievable bandwidth ratios, without any knowledge of the variable network performance. We refer readers to a survey [6] for more details on collective communication algorithms. Figure 1 illustrates the latest algorithm implemented for gather on InfiniBand networks. In the algorithm, there are two major steps, namely intra-rack gather and inter-rack gather. First, a process is selected as the rack leader, and the leader processes independently perform intra-switch gather operations. Second, once the rack leaders have completed the first phase, the data is gathered at the root through an inter-rack gather operation performed over the rack leader processes. Experiments show that the topology awareness significantly improves the performance [13], [28]. However, those algorithms do not take the message size change during the scatter and gather operations into consideration. The message size increases as the communication is close to the root process, while their algorithms use low-bandwidth inter-rack links for those communications. Moreover, they require the network topology information which is not available in the cloud environment. Thus, we need network performance aware algorithms for MPI collective operations in the cloud. A preliminary version of this work has been published in a poster paper [10].

3 A Study in Amazon EC2

To understand the network performance in the cloud environment, we start with the investigation on Amazon EC2. We conduct our study with point-to-point communications among all machine pairs for a set of virtual machines, with the following issues in mind. First, how does the network performance of the point-to-point communication among two virtual machines evolve as time goes by? Second, network performance statistics for a set of virtual machines are important considerations for collective communication operation design. We first apply the standard statistics analysis such as variance, max/min and cumulative percentages for point-to-point communications, and then study the *Symmetry* of the network performance. This property is rooted at the network topology of traditional cluster environments, and require careful revisits on the virtualized network environment.

Since the raw performance is in a continuous numerical domain, it is *not* intuitive to directly analyze the symmetry. We perform *discretization* on the raw performance prior to analysis. Discretization can also reduce the memory consumption of network performance matrix (described in Section 4), usually representing a discretized number with a smaller number of bytes. We use the minimum value in the pair-wise average network performance as the base. The base reflects the smallest latency or the highest bandwidth. We use latency as an example. Suppose the minimum value is min. Our discretization is to translate a measurement value into a level value. We define the discretization as follows: a measurement x is at level l, if and only if $l \leq \frac{x}{k \cdot min} < (l+1)$, where k is an adjusting factor for discretization.

After discretization, we have a more elegant definition on *symmetry*. Denote the level for the network performance (latency or bandwidth) from machine A to machine B to be $L(A, B)$. The network performance between two machines A and B is *symmetric* if and only if $L(A, B) = L(B, A)$.

3.1 Experimental setup

Amazon provides virtual machines (or *instance* in Amazon's terminology, or simply machines) with different amount of storage and RAM as well as different CPU capabilities in different prices. We use three kinds of standard on-demand instances: *small*, *medium* and *cluster* (Cluster Compute Quadruple Extra Large Instance). All the instances are acquired from US East (N. Virginia) data center of Amazon.

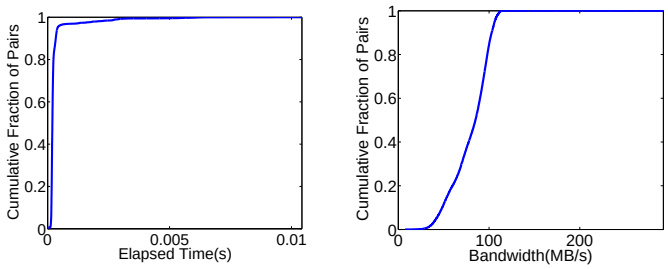
We use the point-to-point communication (MPI_Send) in MPICH2 v1.4 [21], which is a high-performance and widely portable implementation for MPI. We measure the latency and the bandwidth as two key network performance metrics. The latency is the elapsed time of sending a one-byte message and the bandwidth is calculated from the elapsed time of sending 8MB data. The elapsed time for sending a message is measured from an existing MPI benchmark tool named SKaMPI [23]. Specifically, we use the function Pingpong_Send_Recv, which calls MPI_Send followed by MPI_Receive.

For N virtual machines, we need N iterations of calibrations in order to get all pair-to-pair performance. In each iteration, $\frac{N}{2}$ pairs are measured with MPI_Send in both directions. When the number of instances is 128, the total calibration overhead is usually smaller than 100 seconds. While this calibration approach is simple and effective, it will introduce network interference by itself. Fortunately, the impact of self-induced competing traffic should be ignorable. On the one hand, the data center is usually large enough in the scale of tens of thousands of servers, and the interference of the virtual cluster for the MPI application (in the scale of hundreds of virtual machines in our studies) should be small. On the other hand, the calibration result can effectively guide the optimization on MPI collective operations, as shown in the experiments. As a sanity check, we perform the bandwidth calibration by conducting the send/receive operations one by one, and have got similar results.

3.2 Results

Due to the space limitation, we briefly present the major findings, and the detailed results can be found in Appendix A of the supplementary file. Overall, we have observed the unique features of network performance: unevenness and asymmetry of pair-wise network performance.

First, the short-term performance for a machine pair seems ad hoc, and the long-term network performance tends to be relatively stable for the same machine pair (forming a band on latency and bandwidth). However, we clearly observe the difference in the network performance among different virtual machine pairs. Figures 2(a) and 2(b) show the distribution



(a) Latency (1B) (b) Bandwidth

Fig. 2. The distribution of pair-wise latency/bandwidth in 200 medium instances

for the latency and the bandwidth on 200 medium instances on Amazon EC2. This result is consistent with the previous studies [7], [24].

Second, pair-wise network performance is asymmetric. After the discretization, we found neither the pair-wise latency nor the pair-wise bandwidth is symmetric. When the number of medium instances is 128, there are 58.2% and 18.2% of asymmetric virtual machine pairs in terms of latency and bandwidth, respectively. One possible cause of this asymmetry is due to the different routes taken by pair-wise communications in the virtualized network.

4 Algorithm Design

Due to unevenness and asymmetry of pair-wise network performance in the cloud, there is no short-cut in obtaining the network performance statistics, and we need to consider all the pair-wise network performance for a set of virtual machines. Moreover, some machine pairs have very low performance compared with others, which should be carefully scheduled in the algorithm design. Therefore, we develop the *network performance matrices* to estimate the pair-wise network performance, and the *network performance hierarchy* to quantify the closeness of different virtual machines in terms of network performance. Based on the network performance hierarchy, we develop our collective communication algorithms for optimizing their completion time.

4.1 Network Performance Matrices

Our empirical study on Amazon reveals that we need to capture all the pair-wise network performance for a set of virtual machines. In particular, we use two matrices, namely latency matrix and bandwidth matrix, to model the pair-wise network performance. These two matrices are obtained through experimental calibration, as the method described in Section 3. Based on the latency matrix and the bandwidth matrix, we derive the network performance matrix for an arbitrary message size.

Definition. Given a set of virtual machines (the set cardinality is N), we define the three matrices as follows.

- Latency matrix, L[0, 1, ..., $N-1$][0, 1, ..., $N-1$]. $L_{i,j}$ represents the average latency from machine i to machine j. $L_{i,i}$ is initialized with zero.
- Bandwidth matrix, B[0, 1, ..., $N-1$][0, 1, ..., $N-1$]. $B_{i,j}$ represents the average bandwidth from machine i to machine j. $B_{i,i}$ is initialized with ∞.
- For a given message size, m, we define the network performance matrix M[0, 1, ..., $N-1$][0, 1, ..., $N-1$], where $M_{i,j}$ is estimated to be $L_{i,j} + \frac{m}{B_{i,j}}$ ($M_{i,i}$ is initialized with zero). This estimation is commonly used in estimating the network performance of point-to-point communication. More details about this estimation model can be found in the previous study [29].

We perform discretization on those matrices. In the remainder of this paper, we refer M to be the ones after discretization. This study focuses on handling fix-sized messages, and it is straightforward to extend our models and algorithms to variable-sized messages. In the remainder of this paper, we use the message size m for calculating M.

4.2 Network Performance Hierarchy

Due to the asymmetric network performance matrix, we design a data structure to guide our optimizations of carefully scheduling the slow links and maximizing the data transfer overlapping. Since network topology has been demonstrated to be effective in optimizing MPI collective operations, we should quantify the closeness of the virtual machines for similar functionality of the network topology awareness. In particular, we define the closeness of two virtual machines in terms of network performance, and further leverage the network performance matrix to define the network performance hierarchy.

Our definition on the network performance hierarchy is inspired by the tree-like topology in data centers. The intra-rack network performance is good and almost the same for any machine pair within the rack. The inter-rack network performance gradually decreases as the level of the lowest common switch for the machine pair becomes higher. Similarly, we can perform grouping on the virtual machines according to the network performance matrices, which allows us to quickly differentiate the fast and the slow links. Additionally, this grouping should form a hierarchy, gradually adapting to the network performance difference between virtual machine pairs.

Algorithm 1 Network performance hierarchy construction

1: Calculate the network performance matrix M based on L, B and m (the message size);
2: Let the iteration counter $iter$ be zero;
3: Denote the set of groups generated in the iteration $iter$ be $\mathbb{G}_{iter}$;
4: Let $\mathbb{G}_0$ be the set of groups, where each group consists of one machine;
5: **while** $|\mathbb{G}_{iter}| > 1$ **do**
6: Perform grouping on $\mathbb{G}_{iter}$, and obtain $\mathbb{G}_{iter+1}$ (Algorithm 2);
7: Increase $iter$ by one;
8: The network performance hierarchy is formed by $\mathbb{G}_0$, $\mathbb{G}_1$, ..., $\mathbb{G}_{iter}$;

Closeness definition. We use a boundary value to quantify how "close" on the network performance between a machine and a group of machines is. For a group of machines, $\eth$, and a boundary value, b, we define a machine p is *b-close* to $\eth$,

if and only if $\forall p' \in \eth$, $max(M_{p',p}, M_{p,p'}) \leq b$. If a process p is b-*close* to $\eth$, we can add p to $\eth$ and form a new group, subject to the boundary value b. The definition considers all the machines in the group. In this definition, we use the maximum of the links in two directions to capture the asymmetry in the worst case.

Grouping. Our goal of grouping is to construct a hierarchy to minimize the worst communication time from the top down. However, calculating the optimal hierarchy is an NP-hard problem (similar to the bin-packing problem). Due to the exponential complexity, it is commonly infeasible to calculate the optimal solution, especially when the number of virtual machines is large. Therefore, we choose a greedy method to construct the hierarchy. The grouping process generates a network performance hierarchy, as illustrated in Algorithm 1.

The basic idea is to form a number of non-overlapping groups with similar network performance to each other. As we gradually relax the closeness boundary, small groups are combined into larger ones. In large scale processing, there may be many groups, and we need multiple levels of grouping. The parent-child relationship among groups forms a tree hierarchy. As the closeness boundary is gradually increased, the hierarchical grouping in our algorithm gracefully adapts to the unevenness in the pair-wise network performance. In order to create a more balanced tree structure, we also need to adjust the number of machines in each group. This is captured by a parameter on a set of groups called *span*, which is defined to be the difference in the number of machines between the largest group and the smallest group. If the span is zero, the tree is balanced. Taking a set of groups as input, the grouping method at each level works in Algorithm 2.

If we look at the grouping from the perspective of the network performance matrix, we essentially perform row exchanges on the network performance matrix such that the sub-matrix along the diagonal satisfies the b-close requirement. For example, when $b = 0$, grouping will result in the sub-matrices along the diagonal with zeros.

Algorithm 2 Grouping on group set $\mathbb{G}$ and obtain new group set $\mathbb{G}'$.

```
1: Denote b to be the boundary value for grouping (initialized with zero);
2: Initialize 𝔾′ such that |𝔾′| = 0;
3: while |𝔾′| > |𝔾|/2 or |𝔾′| = 0 do
4:    Clear 𝔾′;
5:    for any machine p ∈ ð, ∀ð ∈ 𝔾 do
6:       Add p to an existing group ð′ ∈ 𝔾′ if p is b-close to ð′;
7:       if such a group does not exist then
8:          Create a new group for p, and add it to 𝔾′;
9:    Increase b by 1;
10: while the span of 𝔾′ is larger than T_span do
11:    Adjust 𝔾′
12: return 𝔾′;
```

To limit the number of levels in the hierarchy, we make sure that the number of groups generated in a level is no more than one half of the number of groups in the previous level. Thus, the maximum number of levels in the network performance hierarchy is no longer than $\log_2 N$, where N is the number of virtual machines involved in the collective communication. Note, this estimation has excluded the intra-machine communication, since the intra-machine bandwidth is often over orders of magnitude higher than the network bandwidth. Algorithm 1 starts with $\mathbb{G}_0$, where each group consists of one single machine.

Let us present more details on the grouping algorithm (Algorithm 2). There are two important variables for grouping (b for grouping and T_{span} for span). For b, a small boundary value generates the groups with high closeness. However, this may generate too many groups. For example, if there are too few zeros in the network performance matrix, setting $b = 0$ could result in most of the groups containing only one machine. Thus, we gradually increase the boundary value by 1 (Line 9) and generate the groups until the number of generated groups is no larger than one half of the number of groups in the input. Thus, we can calculate the grouping complexity, which is $\sum_{i=0}^{log_2 N} (\frac{N}{2^i})^2 \sim O(N^2)$. Also, the storage consumption is $O(N^2)$. For T_{span}, we try to balance the tree structure while maintaining the grouping closeness. If the span of the group is larger than T_{span}, we adjust the grouping by moving one machine from the largest group to the smallest group until the span is no larger than T_{span}.

There is one thing worth noting in the grouping process. The network performance matrix is not symmetric and a machine may be feasible to be added to multiple groups. That is the major difference with the network topology awareness, where machines are statically assigned to the racks. On the other hand, since a machine can be added to multiple groups, there is an opportunity in making group sizes more balanced. In the grouping process, we seek all the qualified group candidates for the machine. Next, we always add the machine to the group candidate which has the smallest number of processes. Thus, each new machine is more likely to be added to the smaller group. In practice, this simple approach can result in much more balanced group sizes (i.e., a smaller span for the groups).

The network performance hierarchy shows the closeness of the network performance between any two machines. The network performance of two processes is measured by the network performance of their corresponding virtual machines. Thus, the network performance of any two machines can be represented by the level of the lowest common ancestor in the network performance hierarchy. We define this level to be the *distance* between the two machines. We also use the *distance* to measure the distance of two processes (i.e., the distance of the two machines that the two processes are in). The lower level means a smaller distance and thus a higher network performance. For example, the processes which belong to the same group at the bottom level of the network performance hierarchy are the ones within the same virtual machine, and they have a distance of zero.

Finally, we note that the network performance hierarchy can also be applicable to traditional cluster environments. When all the pair-wise network performances are the same, the grouping procedure generates one group. For the cluster

with tree topology, network performance hierarchy naturally matches the network topology.

Algorithm 3 Network performance aware reduce

1: Calculate the network performance hierarchy $\mathbb{G}_0$, $\mathbb{G}_1$, ..., $\mathbb{G}_l$ (l is the number of levels in the network performance hierarchy) based on L, B and m (the message size);
2: Let the iteration counter $iter$ be zero;
3: Pick a group leader process for each machine in $\mathbb{G}_0$ and create a sub-communicator for the processes in each machine;
4: Each group performs the reduce operation independently;
5: Increase $iter$ by one;
6: **while** $iter < l$ **do**
7: Pick a leader for the group of leader processes in the same group in $\mathbb{G}_{iter}$;
8: Create a sub-communicator for each group;
9: Each group performs the reduce operation independently;
10: Increase $iter$ by one;

4.3 Network Performance Aware Operations

We exploit the network performance hierarchy to develop the network performance aware collective operations.

Broadcast and Reduce. Since broadcast and reduce are two symmetric operations, they have similar network performance aware algorithms, and we focus on the description of the reduce operation. The algorithm for our network performance aware reduce algorithm is illustrated in Algorithm 3.

The basic idea is to exploit the non-overlapping groups with similar network performance to each other in the network performance hierarchy, and to create a sub-communicator for each group to perform partial reduction. The sub-communicator allows taking advantage of an existing optimized MPI implementation. Within each group, we consider the processes are close to each other. Thus, an existing MPI implementation without network performance awareness is already sufficient. In our experiment, we simply call MPI_Reduce in MPICH2 to perform reduce in each sub-communicator. We choose a group leader process from each group, and all the leaders form an upper level for the reduce operation. Figure 3 illustrates a two-level reduce operation, where multiple sub-communicators are created according to the network performance hierarchy.

Our algorithm carefully schedules the links for communications. Particularly, it exploits the links with relatively high network performance and avoids the slow links across different groups. We select the group leader such that it can have better network performance with other group leaders. However, this can be a complex and time-consuming process, since we have many candidates to choose from. Thus, we adopt a greedy approach to select the group leader. Given a group $\eth$, we select p ($p \in \eth$) as the leader such that $\sum_{p' \in \eth} M_{p',p}$ is minimized. That means, the group leader minimizes its total worst case pair-wise network performance. The leader process is the root in the sub-communicator and only in-bound links are considered.

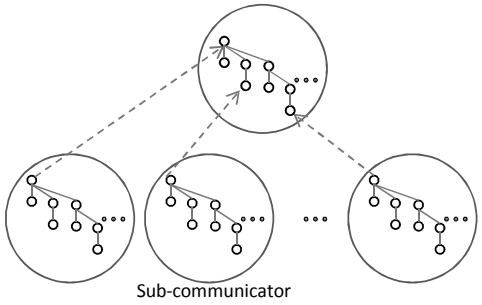


Fig. 3. Network performance aware reduce operation

The root process of the reduce operation needs special care, since the message is finally sent to the root process. To avoid this extra cost, we assign the root process as the group leader at all levels of the hierarchy. In our implementation, we can change the order of the processes in the group such that the root process has the rank of zero.

Gather and Scatter. Since gather and scatter are two symmetric operations, we have designed a similar network performance aware algorithm for scatter, and we focus on the description of gather.

Algorithm 4 Network performance aware gather

1: Calculate the network performance hierarchy (denoting its root to be R_h) based on L, B and m;
2: Let the root process of the gather be $root$;
3: Let the current distance $dist$ be zero;
4: *Construct*($root$, R_h, $dist$);
5: Perform the gather operation according to the communication tree with the root at $root$;

Procedure: *Construct*($root$, R_h, $dist$)
Input: The root process of the gather, $root$, the root of the network performance hierarchy, R_h
Description: The recursive procedure for constructing the communication tree.

1: Denote UP to be the set of processes whose distance to $root$ is no larger than $dist$;
2: Denote $DOWN$ to be the set of subtrees whose root has a distance to $root$ equal to $dist + 1$;
3: Adjust $DOWN$ to match UP such that $|UP| = |DOWN|$;
4: **for** each process $u \in UP$ **do**
5: Pick an unattached subtree t from $DOWN$;
6: **if** t has more than one node **then**
7: Let the root of t be d such that u and d has the best network performance among the nodes in t;
8: *Construct*(d, R_h, $dist + 1$);

In gather, the message size increases as the message approaches the root process. This message size change prohibits us to use the same design as the broadcast and reduce operations. Instead, we need to carefully schedule the links with low network performance in order to adapt to the message size change in the gather operation.

Algorithm 4 illustrates our network performance aware algorithm for gather. The basic idea behind the algorithm is

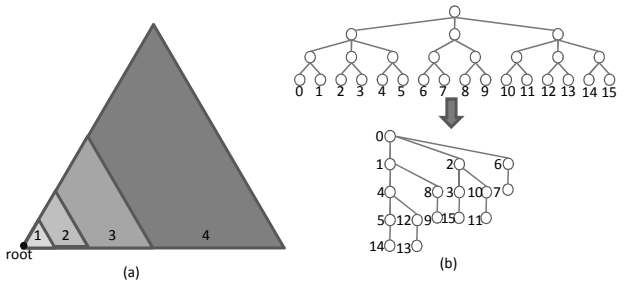


Fig. 4. The construction of the communication tree in gather: (a) construction by distance; (b) network performance hierarchy (top) and the result communication tree (bottom).

to exploit the distance property of the network performance hierarchy. We arrange the communication links according to the increasing order of their distance to the root process, as illustrated in Figure 4 (a). This scheduling mechanism well adapts to the message size change and the network performance variation among machine pairs in the gather operation. In particular, it has two distinct features. First, the links with a smaller distance to the root are arranged to be close to the root process, and those links have higher network performance according to the network performance hierarchy. This is good for larger messages near the root process. Second, the links with the same distance are executed in parallel, and their data transfers can be well overlapped. The impact of overlapping is significant, especially for those links with low performance. Additionally, our approach distributes the message into multiple links associated with different subtrees, to avoid the large message sent by a single leader process per rack in the previous studies [13], [28].

Let us describe more details on link scheduling. We start with the root process as the root for the communication tree. The recursion essentially builds the tree level by level. The detailed flow is given by the Procedure *Construct* in Algorithm 4. At level $dist$, we attach all the subtrees in $DOWN$ to all the processes in UP. In the network performance hierarchy, we denote UP to be the set of processes whose distance to the root process is $dist$, and $DOWN$ to be the set of subtrees whose root has a distance to root $dist$+1. Since UP and $DOWN$ may have different sizes, we adjust $DOWN$ until these two sets have the same cardinality (i.e., $|UP| = |DOWN|$). The adjustment repeats the following operations. (1) Splitting when $|UP| > |DOWN|$. We pick the subtree with the largest number of nodes in $DOWN$ (denoted as $oldSubTree$), and split it into two subtrees (removing the root of the subtree and combining the forests if necessary). Thus, we substitute $oldSubTree$ with the two new subtrees in $DOWN$, and thus $|DOWN|$ is increased by one. (2) Combining when $|UP| < |DOWN|$. We combine the two smallest subtrees in $DOWN$ into one. Through splitting and combining, we make sure $|UP| = |DOWN|$ and each subtree in $DOWN$ is attached to a single process in UP.

Figure 4 (b) illustrates an example of the gather operation on 16 virtual machines. The example shows the scenario of MPI process i running on virtual machine i. The network performance hierarchy has four levels. Process 0 is the root process for the gather operation. The processes in the result communication tree is arranged according to the distance to the root node. Note, we have split the subtree consisting of Processes 14 and 15 during the attaching process in Level 4.

5 Evaluations

In this section, we present our experimental results on our network performance aware collective communication algorithms.

5.1 Experimental Setup

We use two complementary approaches to evaluate the efficiency of our algorithms (denoted as CMPI). One approach is to evaluate our algorithms on Amazon EC2 with a reasonable scale of virtual machines, and the other is to perform simulation such that the evaluation can be performed on a much larger scale. The experimental setup of Amazon is the same as those in Section 3.

On both real cloud environments and simulation, we compare our network performance aware algorithms with MPICH2 [21]. MPICH2 in most cases chooses the binomial tree algorithm. Note, it may also choose different algorithms according to the message size. For example, when the message size is larger than 1MB, MPICH2 chooses the ring algorithm for broadcast. The major metric in our study is the completion time of the collective operation. We use an existing MPI benchmark called SKaMPI [23] to measure the completion time.

For space interests, we present the simulation setup and results in Appendix B of the supplementary file. On simulations, our optimized collective operations are 25.4–31.2% faster than their counterparts in MPICH2 when the number of process increases from 32 to 2048.

For each kind of operation, we evaluate the impact of message sizes, the number of virtual machines and the number of processes per virtual machine. The *default* setting is: 64 medium instances and 200 processes. The message size is 1MB per process. The root process is randomly chosen. We present the results under the default setting, specified otherwise. We set $k = 4$ and $T_{span} = 1$ for the discretization and grouping algorithm on Amazon EC2. We study the impact of k and T_{span} and other parameters in Appendix B of the supplementary file.

On the same virtual cluster, we obtain the network performance matrix when the experiment starts, and use it afterwards unless the network performance significantly changes. We re-calibrate the matrix and re-build the tree if we find that the performance differs significantly from our estimation. In our experiments, we set the difference to be 20% by default.

Since broadcast and reduce are dual operations for gather and scatter, respectively, we observe similar results on those dual operations. We focus on the results for broadcast and scatter in this section. More results for gather and reduce,

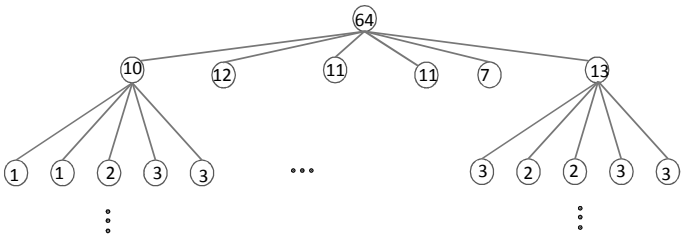


Fig. 5. The network performance hierarchy for 64 medium instances

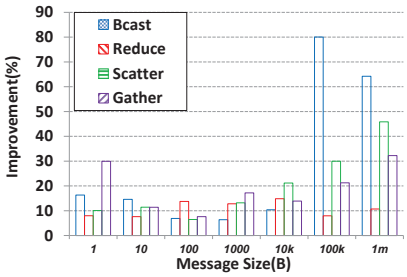


Fig. 6. Performance improvement for 64 medium instances

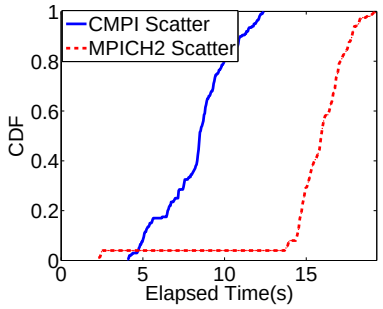
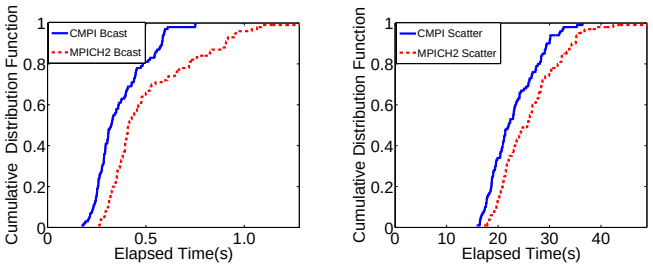


Fig. 7. CDF of individual process elapsed time for scatter



(a) Broadcast (b) Scatter

Fig. 8. The distribution of broadcast and scatter performance with different virtual clusters.

evaluations on cluster instances and sensitivity studies can be found in Appendix B of the supplementary file.

5.2 Results on Collective operations

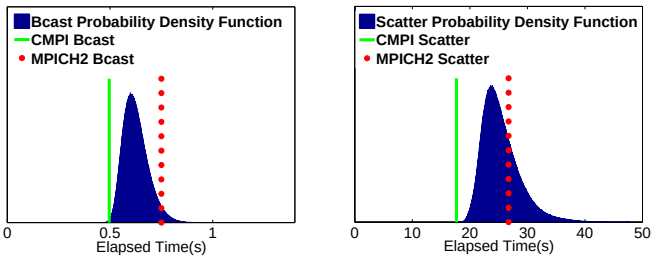
Network performance hierarchy. Figure 5 shows a snapshot of the network performance hierarchy for the 64 medium instances under the default setting. The number of levels in the hierarchy is four. The number in the node represents the number of virtual machines in the group. The group sizes are rather balanced. Note, the network performance hierarchy usually varies for different sets of virtual machines; its structure is rather stable if the set of virtual machines is fixed.

Overall comparison. We focus on comparing the performance of the collective operation themselves since they are usually used as components in applications. We leave the study on the impact of the construction overhead in the application evaluation. Figures 6 shows the performance comparison of the four operations with different message sizes. On the four collective operations, our network performance aware algorithms significantly outperform their counterparts in MPICH2. It indicates the importance of the network performance awareness in the cloud environment. The improvement is relatively larger for larger message sizes. The average improvement is 38.2%, 13.5%, 22.7% and 27.3% on broadcast, reduce, scatter and gather, respectively. Note that there is a huge gap in Bcast when the message size is 10^5 and 10^6 bytes. That is because the MPICH2 Bcast chooses the ring algorithm, instead of the binomial tree algorithm. The ring algorithm turns out to be even slower than our algorithm.

Figure 7 shows the CDF (cumulative distribution function) of elapsed time of individual processes for scatter. CMPI algorithms have much more balanced distribution on the process execution time than MPICH2. The balanced process time distribution indicates the effectiveness of our algorithm on scheduling the links with similar performance and maximizing the impact of overlapping.

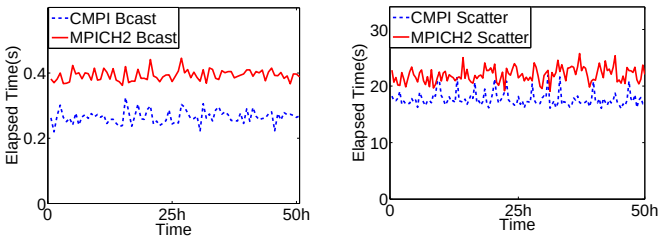
We further study the impact of different virtual machines in the virtual cluster. For each measurement, we restart a virtual cluster of 64 medium instances, measure the execution time for CMPI and MPICH2 and then terminate all the virtual machines. Figure 8 shows CDF for 100 measurements for broadcast and scatter. On different virtual clusters, CMPI outperforms MPICH2 and the execution time of CMPI is more stable.

Finally, we study the effectiveness of our heuristics based optimization, in comparison with other classic optimization methods. The experiments are conducted in simulations. Given the network performance matrix under the default setting, we



(a) Bcast(Message Size=1MB) (b) Scatter(Message Size=1MB)

Fig. 9. Probability density distribution with Monte Carlo simulations (broadcast and scatter)



(a) Broadcast (b) Scatter

Fig. 10. Long-term performance of Bcast and Scatter

implement Monte Carlo simulations to randomly generate the communication tree structure for collective operations. With a large number of simulations, we get the distribution of Monte Carlo simulations and investigate how far our optimization is from the best one in Monte Carlo simulations. With 10^7 times of simulations, we obtain the probability density distribution, as shown in Figure 9. We also show the performance of the proposed approach and MPICH2. Our approach is close to the best of Monte Carlo simulations.

5.3 Results on Long-term Behavior

Due to the network performance dynamics in the cloud, we study the long-term performance behavior of our proposed approach. We conduct back-to-back experiments for our implementation and MPICH2: we first run CMPI, then MPICH2, and then repeat the experiments immediately.

Figure 10 shows the long-term performance comparison between MPICH2 and our approaches for over two days. We find that our optimizations work very well along the time. The performance improvement for the two days is rather stable (30.1% and 17.3% for broadcast and scatter, respectively). The mean and the variance values are (0.27, 4.0e-4) and (0.91, 2.2e-3) for broadcast and scatter, respectively. The variation is usually smaller than 10% of the mean. We conjecture two possible implications on Amazon EC2. First, the network traffic involving those instances is rather stable during the period. Second, there are few or even no occurrences of virtual machine migrations among those instances during the period.

In another set of experiments, we tried re-calibrating the network performance matrix and re-constructing the communication tree for our approach every hour. However, the performance improvement of the maintenance is very small (less than 2% on average). The overhead of maintenance further offsets the performance improvement.

5.4 Results on Applications

To further evaluate the impact of our network performance aware algorithms, we have implemented two applications namely N-body and conjugate gradient (CG). N-Body is an astronomy model, aiming at simulating the movement, position and other attributes of bodies with gravitational forces exerted on one another. The parameters of N-Body include the number of steps for the simulation (#Step) and the number of bodies. CG is an iterative method, with the core operation of sparse matrix vector multiplication (SpMV). CG converges as more iterations are conducted, and we set the convergence condition: $\|r\| \leq 10^{-5} \times g_0$ (r is the residual norm and g_0 is the initial gradient). In both applications, we implement the allgather communication with a gather followed by a broadcast, which is also used in MPICH2 [21]. This simple implementation is sufficient for us to investigate the impact of collective operations in distributed applications. More advanced all-to-all algorithms (such as other algorithms in MPICH2 [21]) are considered as our future work (more future work is discussed in Section 6). During the execution period of both applications, we observe few changes in the network performance matrix, and the network performance hierarchy and the communication tree are constructed once in one execution of the application.

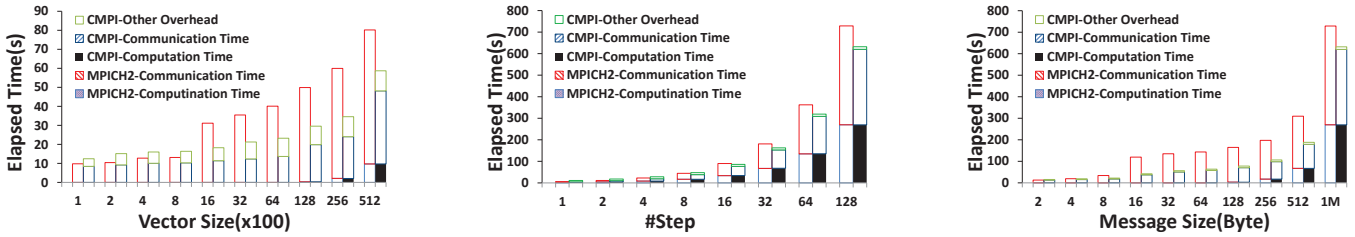
We study the breakdown of the execution time in the application. In particular, we divide the entire application execution time into two parts: computation and communication. For our proposed algorithms, we also present the initialization cost including network performance hierarchy construction and communication tree construction (denoted as "Other Overheads").

Figure 11(a) shows the comparison studies for CG. In this experiment, we vary the vector size from 100 to 51200. We make two observations. First, the CG performance is network-bounded, with communication time contributing over 90% to the total execution time in MPICH2. Second, when the vector size is small, our algorithm is slower than MPICH2-based CG, due to the tree and hierarchy construction overheads. As the vector size increases, more iterations are required for convergence. Network communication time becomes more significant and the network performance aware optimization reduces the network communication time. The performance gain compensates the overhead, with 41.6% performance improvement over MPICH2.

Figures 11(b) and 11(c) show the performance comparison for N-body. We firstly fix the message size as 1M bytes and vary #Step from 1 to 128. Then we fix #Step to be 128 and vary the message size from 2K to 1M bytes. As the message size and #Step increase, the computation and communication play a more important role and the overhead becomes insignificant. CMPI algorithms reduce the network communication time by 33.2%, and the total execution time by 14.3%.

6 Conclusion and Future Work

In this paper, we investigate the performance of MPI collective communication operations, and our empirical studies in Amazon EC2 reveal the unique features of network performance: unevenness, asymmetry and intransitivity of pair-wise network performance. Those findings motivate us to develop the network performance hierarchy for capturing the closeness of any two virtual machines in terms of network performance. Based on the network performance hierarchy, we develop novel algorithms for collective communication operations including broadcast, reduce, gather and scatter. Our experimental results show that the network performance awareness results in



(a) CG (b) N-body(Message Size=1MB) (c) N-body(#Step=128)

Fig. 11. Performance comparison of applications on 64 medium instances: N-Body and CG.

25.4% and 28.3% performance improvement over MPICH2 on Amazon EC2 and on simulations, respectively. Evaluations on N-body and CG show 41.6% and 14.3% respectively on application performance improvement. Our future work is to explore monetary cost optimizations for MPI [30].

ACKNOWLEDGEMENTS

The authors thank the anonymous reviewers for their insightful suggestions. This work is supported by the funding from National Research Foundation through the Environment and Water Industry Programme Office (EWI) on project 1002-IRIS-09. The experiment on Amazon EC2 is supported by an Amazon AWS Research Grant (2011-2012).

REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[2] D. Battr, N. Frejnik, S. Goel, O. Kao, and D. Warneke. Evaluation of network topology inference in opaque compute clouds through end-to-end measurements. In *IEEE CLOUD'11*, pages 17–24, 2011.

[3] M. Burger and T. Kielmann. Collective receiver-initiated multicast for grid applications. *IEEE Trans. Parallel Distrib. Syst.*, 22(2):231–244, Feb. 2011.

[4] R. Castro, M. Coates, G. Liang, R. Nowak, and B. Yu. Network tomography: recent developments. *Statistical Science*, 19:499–517, 2004.

[5] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda. Designing an efficient kernel-level and user-level hybrid approach for mpi intra-node communication on multi-core systems. In *ICPP*, pages 222–229, 2008.

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

[7] T. Chiba, M. den Burger, T. Kielmann, and S. Matsuoka. Dynamic load-balanced multicast for data-intensive applications on clouds. In *CCGRID*, pages 5–14, 2010.

[8] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, pages 98–109, 2011.

[9] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. *SIGMETRICS Perform. Eval. Rev.*, 2005.

[10] Y. Gong, B. He, and J. Zhong. An overview of cmpi: network performance aware mpi in the cloud. In *PPoPP*, pages 297–298, 2012.

[11] R. L. Graham and G. Shipman. Mpi support for multi-core architectures: Optimized shared memory collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140, 2008.

[12] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. *SIGCOMM*, 38(4), 2008.

[13] K. C. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda. Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather. In *10th Workshop on Communication Architecture for Clusters*, 2010.

[14] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *IPDPS*, 2000.

[15] N. T. Karonis, B. Toonen, and I. Foster. Mpich-g2: a grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, May 2003.

[16] A. Karwande, X. Yuan, and D. K. Lowenthal. Cc–mpi: a compiled communication capable mpi prototype for ethernet switched clusters. In *PPoPP*, 2003.

[17] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. Magpie: Mpi's collective communication operations for clustered wide area systems. In *PPoPP*, 1999.

[18] P. Lai, S. Sur, and D. K. Panda. Designing Truly One-Sided MPI-2 RMA Intra-node Communication on Multi-core Systems. In *International Supercomputing Conference (ISC)*, June 2010.

[19] B. Lowekamp, D. O'Hallaron, and T. Gross. Topology discovery for large ethernet networks. *SIGCOMM Comput. Commun. Rev.*, 2001.

[20] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. Mpi collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *CCGRID*, pages 130–137, 2008.

[21] MPICH2. *http://www.mcs.anl.gov/research/projects/mpich2/*.

[22] V. Paxson. End-to-end routing behavior in the internet. *SIGCOMM Comput. Commun. Rev.*, 1996.

[23] R. Reussner, P. S, L. Prechelt, and M. Muller. Skampi: A detailed, accurate mpi benchmark. In *In Vassuk Alexandrov and Jack Dongarra, editors, Recent advances in Parallel Virtual Machine and Message Passing Interface*, pages 52–59. Springer, 1998.

[24] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 2010.

[25] M.-F. Shih and A. Hero. Hierarchical inference of unicast network topologies based on end-to-end measurements. *Trans. Sig. Proc.*, 55(5):1708–1718, May 2007.

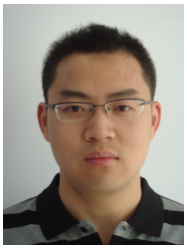
[26] S. Sistare, R. vandeVaart, and E. Loh. Optimization of mpi collectives on clusters of large-scale smps. In *SC*, 1999.

[27] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *HotPower*, 2008.

[28] H. Subramoni, K. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. McLay, K. Schulz, and D. K. Panda. Design and evaluation of network topology-/speed-aware broadcast algorithms for infiniband clusters. In *IEEE Cluster*, 2011.

[29] R. Thakur and R. Rabenseifner. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.

[30] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou. Distributed systems meet economics: pricing in the cloud. In *HotCloud*, pages 6–6, 2010.



Y ifan Gong received the bachelor degree in School of Mathematic from Peking University (2006-2010), and is now a Ph.D. student in Interdisciplinary Graduate School and School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing and cloud computing.

B ingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in Division of Networks and Distributed Systems, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, cloud computing, and database systems.

J ianlong Zhong received the bachelor degree in software engineering from Tianjin University (2006-2010), and is now a PhD candidate in School of Computer Engineering of Nanyang Technological University, Singapore. His research interests include GPU computing and parallel graph processing.