Summer 8-12-2014

Towards an MPI-like Framework for Azure Cloud Platform

Sara Karamati

TOWARDS AN MPI-LIKE FRAMEWORK FOR AZURE CLOUD PLATFORM

by

SARA KARAMATI

Under the Direction of Dr. Sushil K. Prasad

ABSTRACT

Message passing interface (MPI) has been widely used for implementing parallel and distributed applications. The emergence of cloud computing offers a scalable, fault-tolerant, on-demand alternative to traditional on-premise clusters. In this thesis, we investigate the possibility of adopting the cloud platform as an alternative to conventional MPI-based solutions. We show that cloud platform can exhibit competitive performance and benefit the users of this platform with its fault-tolerant architecture and on-demand access for a robust solution. Extensive research is done to identify the difficulties of designing and implementing an MPI-like framework for Azure cloud platform. We present the details of the key components required for implementing such a framework along with our experimental results for benchmarking multiple basic operations of MPI standard implemented in the cloud and its practical application in solving well-known large-scale algorithmic problems.

INDEX WORDS: High-performance computing, Windows Azure, MPI, Cloud computing

TOWARDS AN MPI-LIKE FRAMEWORK FOR AZURE CLOUD PLATFORM

by

SARA KARAMATI

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2014

TOWARDS AN MPI-LIKE FRAMEWORK FOR AZURE CLOUD PLATFORM

by

Sara Karamati

Committee Chair: Sushil K. Prasad

Committee: Rafal Angryk

Yi Pan

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

August 2014

ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my advisor, Dr. Sushil Prasad, for his support and guidance throughout this research. I would like to thank Dinesh Agarwal for his contribution in the publication of "Towards an MPI-like framework for the Azure cloud platform" in the CCGrid 2014 conference. I would like to thank my colleagues at DiMoS group for sharing their knowledge and experience with me and their help in my research.

My deepest appreciation goes to my husband, Reza, who supported me in many ways and made me feel strong. This work could not be done without his support, understanding, and love. Most of all, I thank my parents, my sister, and my brother for their endless love, support, and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Motivation

For several decades, computer industry enjoyed increasing number of transistors on a chip with a proportional increase in the clock frequency with no drastic limitations on the electrical and thermal power. Last decade marked a shift in computing industry and the clock frequency could no longer increase because of increasing temperature of the CPUs and limitations in the cooling systems. The clock frequency barrier along with the advent of mobile computing ushered us in the era of multi-core low-power architectures aided by accelerator units like general-purpose GPUs and specialized hardware —including hardware-accelerated codecs, digital signal processors, etc. In this period, software developers should actively optimize their applications by taking advantage of the special capabilities of the underlying hardware (e.g., cache-aware algorithms, SIMD operations, off-loading work to GPU, etc.) in order to improve speed and energy efficiency of their applications. Furthermore, with the increasing bandwidth and reliability of the Internet, scaling software systems as massive distributed services has become more and more ubiquitous. These massive systems (networked servers and data-centers distributed all over of the world also known as the cloud) work hand-in-hand with the consumer devices in order to deliver quality services. Quality of these services relies on the software that utilizes the capabilities of the cloud and consumer hardware efficiently.

While the advent of cloud computing has provided researchers with a computing platform with unprecedented scale, its adoption for high-performance computing has been limited by the difficulty in employing cloud-based resources. Although the Infrastructure as a Service (IaaS) facilities commonly provided by commercial cloud computing companies promise portability of applications by installing customized software to mimic the capabilities of a virtualized compute

cluster node, IaaS cannot demonstrate essential requirements for high-performance computing such as low-latency networks and fault-tolerant computing.

The steep learning curve involved in understanding the very peculiar and non-uniform architectures and runtime environments of various cloud platforms discourage HPC community to adopt it as an alternative platform. In order for a large-scale penetration of cloud computing platforms into the HPC community, cloud vendors will have to offer easier approaches to utilize these platforms for scientific research.

One of the popular programming environments followed by the HPC community is the message passing interface (MPI) [1]. Various implementations of MPI standard are available in the market proving the effectiveness of MPI standard for parallel and distributed application development. While MPI has been a popular choice for traditional parallel and distributed platforms, its current implementations are not pragmatic for cloud computing platforms [2]. Creating an MPI-like framework for cloud platforms thus is a non-trivial problem. Cloud platforms most often come with their own set of APIs and hence porting legacy parallel and distributed applications require a great deal of engineering.

In order to help these applications run on cloud platforms with as little effort as it would be to run on any other platform, it is essential to bring the same frameworks that these applications use to cloud platforms. Unlike traditional parallel and distributed platforms, cloud platforms have better fault tolerance and recovery due to their relatively stable operation. The traditional frameworks could exploit this feature to simplify their fail-proofing mechanisms. Other key strengths of the cloud can be exploited for coarse-grained, long-running applications including variable pricing, on-demand allocation/deallocation and scalability to tradeoff budget vs. performance, budget vs. time constraints, etc. It is not a question of efficient implementation of ex-

isting standards on a new platform, but rather, it requires careful adaptation in tune with the significantly different computing infrastructure.

1.2 Introduction to Windows Azure

Windows Azure is a Microsoft web service that provides flexible cloud platform for building, deploying, and managing applications. It allows users to reliably host and scale out their applications, store and manage data in many different ways, and provides messaging capabilities for distributed applications development.

Unlike traditional clusters, cloud platform can easily scale by increasing/decreasing the capacity of individual nodes through hardware upgrading/downgrading (i.e., changing memory capacity or the number of CPU cores, etc.) or by adding/releasing nodes (also known as horizontal scaling) [3]. In order to benefit from vertically scaling the cloud application, in addition to reasonable hardware, sufficiently capable software is also required that can take advantage of the available hardware.

Horizontal scaling pattern can minimize cost by releasing some of the allocated resources, when they exceed the demands of application and increase the resources as throughput falls below default expectations. Efficient utilization of the cloud resources is important since cloud platform follows pay-as-use model. Scaling activities can be automated programmatically by monitoring specific performance metrics such as memory usage, CPU utilization, and average queue length, and so on. For applications with variable or unknown workload, the cloud elastic feature can be used to adjust the resources according to the demand and consequently leading to cost savings by releasing unused resources.

In the following section, we discuss key components of Windows Azure that are used in this thesis.

1.3 The Components of Windows Azure

- **Virtual Machines (VMs):** Windows Azure virtual machines are configurable and maintainable servers in the cloud. These scalable computing resources can be set up with software and services on Windows Server or Linux-based operating systems. Because of the control on the configuration and recycling existing virtual machine images, Azure virtual machines are suitable options for migrating legacy codes and applications to the cloud.
- **Cloud Service:** Azure cloud service is designed for developing multi-tier applications on a platform consisting of one or more compute roles. Convenience of deploying multiple roles makes this service a good option for applications requiring distributed processing and flexible scaling. Azure cloud service supports two kinds of roles namely web role and worker role. Unlike worker role, web role runs IIS so they can be used for front-end web applications. At the cost of limited control in comparison to virtual machines, Azure cloud service assures maintaining infrastructure, patching operating system, and restoring from hardware and service failures.
- **Cloud Storage:** Azure storage service is classified into three categories based on the characteristics of the stored data. Blob storage is ideal for storing large amount of unstructured data. It can contain hundreds of gigabytes of data. Table storage stores structured non-relational data. A single table can hold a collection of entities with different set of properties. Queue storage is the perfect means to pass messages between Azure roles. It can contain an unlimited number of messages each with maximum size of 64KB.
- **Service Bus:** Azure Service Bus provides three different communication mechanisms for messaging: queues, topics, and relays. Service Bus queue and topic are one-way durable

and asynchronous messaging components that store messages until they are consumed by the receiver. The main difference between queue and topic is in the number of receivers of a message. While a message in a queue can be received by only one possible subscriber, a topic allows the message to be received by multiple subscribers that satisfy specific criteria. Service Bus relay service provides direct communication between sender and receiver over a TCP channel. Unlike queue and topic, relay service supports bi-directional messaging.

Table 1.1 compares the features of the Windows storage queue and Service Bus queue. The data in this table are collected from the advertised information about these services by Microsoft [4].

Table 1.1 Comparison of Windows Azure queues and Service Bus queues

Comparison Criteria	Windows Azure Queues	Service Bus Queues
Ordering guarantee	No	Yes - First-In-First-Out (FIFO)
Receive behavior	Non-blocking	Blocking with/without timeout (offers long polling, or the "Comet technique") Non-blocking (through the use of .NET managed API only)
Maximum message size	64 KB	256 KB
Maximum queue size	100 TB	1, 2, 3, 4 or 5 GB
Maximum message TTL	7 days	Unlimited
Maximum number of queues	Unlimited	10,000
Maximum throughput	Up to 2,000 messages per second	Up to 2,000 messages per second
Average latency	10 ms (with TCP Nagle disabled)	100 ms
Queue transaction cost	$0.01 (per 10,000 transactions)	$0.01 (per 10,000 transactions)
Billable operations	All	Send/Receive Only

		(no charge for other operations)
Idle transactions	Billable (querying an empty queue is counted as a billable transaction)	Billable (a receive against an empty queue is considered a billable message)
Storage cost	$0.14 (per GB/month)	$0.00
Outbound data transfer costs	$0.12 - $0.19 (depending on geography)	$0.12 - $0.19 (depending on geography)

1.4 Challenges of Cloud HPC

Several researchers have studied the practicality of running tightly coupled and MPI-style applications in the cloud environment. These studies evaluate the performance of MPI applications on different cloud platforms including Amazon EC2 [2, 5-9] and Microsoft Windows Azure [10, 11]. Most studies use classical MPI benchmarks such as NAS, NPB, HPL, and CSFV to compare the performance of MPI on public cloud platforms. Others evaluate the feasibility of running large-scale applications on the cloud such as low-order coupled atmosphere-ocean simulation [5] and biomedical applications [11], matrix multiplication, K-means Clustering [8] . All studies confirm there is a strong correlation between the application communication time and application overall performance on the cloud platform. These studies show that the lack of high-bandwidth, low-latency interconnects as well as virtualization overhead has large effect on the performance of HPC applications on the cloud. Jackson et al. [6] report a significant variability in performance on Amazon cloud platform due to virtualization in the cloud environment and consequent resource sharing and contention.

This project is an ongoing work in our research group. cloudMPI first introduced in dissertation titled “Scientific High Performance computing (HPC) Applications on the Azure Cloud Platform” by Dinesh Agarwal [12]. In this dissertation, the initial implementation of cloudMPI

is considered. The previous version only considered the point to point routines for cloudMPI using Azure queue storage. The interface of cloudMPI was different from conventional MPI and it involved some effort to convert legacy MPI applications to cloudMPI.

1.5 HPC Frameworks on Azure

MapReduce is a widely used programming model that provides good performance in the cloud platform. Several studies have evaluated the performance of MapReduce in the cloud platform. Microsoft Daytona [13] is an iterative MapReduce runtime optimized for data analytics and machine learning built on Microsoft Windows Azure. In Daytona architecture a single master instance is used to perform the scheduling of applications and tasks and handling of failures. The master assigns each map and reduce task to a slave instance. The instances communicate directly through a TCP connection. Twister4Azure [14] is another iterative MapReduce runtime for Windows Azure cloud. This runtime environment uses Azure queues for map and reduce tasks scheduling, Azure tables for metadata and monitoring data storage, Azure blob storage for data storage and the Window Azure compute worker roles to perform the computations. A multi-level data caching mechanism is also used in the Twister4Azure runtime to mitigate the latency issues inherent in the cloud services.

Pregel.NET [15] is a bulk synchronous parallel (BSP) framework for graph processing customized for the Microsoft Windows Azure cloud. Pregel.NET uses the Pregel architecture design [16] for BSP graph processing. In this model, each worker role holds the distributed graph partitions and performs vertex-centric tasks on their own partition of the graph. The storage queues are used for message transmission between vertices on different compute nodes as well as synchronization between instances.

AzureBlast [10] is a parallel implementation of BLAST library on Windows Azure. Blast is a sequence comparison tool widely used in bioinformatics applications. To run Blast on multiple instances, the query-segmentation data-parallel pattern is adopted. Given an input file, which contains a number of query sequences, AzureBlast distributes partitioned input sequences between Azure compute instances to be executed. Once all workers process their assigned partition, the results are merged and become available on the blob storage.

1.6 Organization of the Thesis

In this thesis, we discuss the key components of implementing a message passing framework on Azure cloud platform and provide design guidelines derived from our experiments. The goal of this thesis is to theoretically and experimentally investigate the cloud environment for high-performance computing and to identify the strengths of the Azure cloud platform to efficiently map MPI to this platform. The rest of this thesis is organized as follows:

The design of an MPI-like framework for application development on Azure cloud platform is presented in Chapter 2. An efficient implementation of a proof-of-concept MPI-like framework, cloudMPI, on the Windows Azure cloud platform is provided in Chapter 3. This chapter details the implementation of MPI-like communicators and primary communication primitives for selective point-to-point and collective operations. Detail of experimental results, their evaluation, and the environment setup are discussed in Chapter 4. A study of the practicality and efficiency of the cloudMPI framework by porting widely-used applications to this platform is presented, as well as benchmarking results that assess the current communication efficiency and overhead for short and long messages for Azure API and cloudMPI framework. The conclusion is presented in Chapter 5 where the future direction for this research is also discussed.

1.7 Related Publication

Part of this work was presented in the 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing and accepted for publication in the conference proceedings [17].

2 CONSIDERATIONS FOR PORTING THE MPI STANDARD TO CLOUD

In this section, we discuss the requirements of the components of an MPI-like framework and the challenges that make it an interesting research problem. The framework design is expected to behave as close to traditional MPI framework as possible, yet it can benefit from the scale and dynamicity of cloud computing as well as its fault tolerance.

2.1 Design of MPI Primitives

The point-to-point communication is the most commonly used communication pattern in MPI. There are two categories for MPI point-to-point communications: 1) blocking and 2) non-blocking. These categories provide a number of communication modes: synchronous (MPI Ssend, MPI Issend), buffered (MPI Bsend, MPI Ibsend), ready (MPI Rsend, MPI Irsend), and standard (MPI Send, MPI Isend). Each mode uses different mechanisms to send messages to target nodes offering trade-offs for synchronization overhead, system overhead, buffer space, and reliability. In the following, we focus on different options for communications on Azure cloud environment and consider MPI communication modes of transferring data for them [18].

2.1.1 Point-to-Point Communications

- **Blocking buffered asynchronous communication using Azure queue storage:** In Azure cloud infrastructure, all queue services are accessible from any compute node. This capability makes it possible to use the intra-node communication mechanism to implement MPI methods for messaging among processes that are either on the same compute node or on remote nodes over the cloud. Nemesis communication subsystem [19] for MPICH uses shared memory queue for intra-node messaging. This communication subsystem uses free queues to prevent starvation of senders or receivers and assures efficient utilization of shared memory. Since cloud environment does not have the space limitation

of shared memory, the free queue used in Nemesis implementation can be removed without performance fall-off. Our design goal here is to use Windows Azure queue storage to communicate messages between Azure compute instances. Each compute instance has a receive queue that it needs to poll in order to receive messages from other instances. To send a message to a compute instance, the message is inserted into the queue of the receiving instance. The size of the message sent to the queue cannot exceed 64KB. Therefore, large data are sent to blob and its unique id, which identifies the location of the data in the blob, is sent as a message to the receivers queue. The Azure queue storage does not guarantee a first-in-first-out (FIFO) ordered delivery. Therefore, a mechanism such as local queue is needed to compensate for this deficiency. The receive process would follow these steps: 1) Check the local queue for desired message; 2) If message is found, use it as needed; otherwise, 3) Poll the Azure queue; 4) If message is found use it as needed; otherwise, 5) Put the unexpected message in the local queue and go to step 3.

Communication over queue offers the advantages of asynchronous and buffered communication modes. In comparison to synchronous mode, it is safe because it is not dependent on the order of send and receive operations. Furthermore, senders and receivers do not have to be available at the same time; therefore, if the receiving instance fails, it receives messages reliably as soon as it is available. A visibility timeout can be assigned to a message from the queue and the message will reappear in the queue if it has not been deleted by the end of the timeout period. This feature provides fault tolerance for the application by ensuring that no messages will be lost during instance failure. Additionally, queue service eliminates synchronization overhead by buffering the message in the queue. The flexibility of the communication can be extended using Azure blob storage when the

message is larger than the maximum message size that queue service can handle (typically 64 KB). This effectively eliminates the pitfall of using buffer mode for messages larger than available buffer space which in traditional MPI programs will generate an error and the program will by default exit.

Another alternative to the local queue is using table storage. The sender of a message will store the message in a table instead of a queue. Messages can be automatically tagged with a time-stamp to enforce strict ordering at the time of retrieval. However, table storage does not have the fault tolerance boasted by Azure queue storage service. If a node fails after taking a message out of the table storage, there is no automated recovery or reappearing of the message in the table storage.

- **Blocking buffered asynchronous or blocking synchronized messaging for short messages using Service Bus relay:** The first design scheme, the message passing via Azure storage, is slow for short messages. Therefore, an alternative scheme optimized for short messages can be designed. In this communication scheme, the Service Bus relay service is used to communicate between two compute instances over the TCP channel. In this communication mechanism, each cloud node hosts Windows Communication Foundation (WCF) services for communication operations. Whenever an instance decides to send a message to another instance, it invokes the send service of the receiver instance and passes the desired message as the argument for the service call. This method has two variations. The first variation provides buffered and asynchronous communication and the second variation supports unbuffered and synchronous communication. In the first variation, whenever an instance decides to send a message to another instance, it invokes the send service of the receiver instance and passes the desired message as the argument for

the service call. The send service puts the message inside a local queue in the receiver side. As a result, a receiver instance can receive a message by checking its local queue. In the second variation, the send service call waits until a receiver instance provides a matching receive. In this variation, the send service call (receive routine) first triggers an event and then waits for an event from the receiver (send service call).

- **Blocking buffered FIFO communication using Azure Service Bus queue:** The third design scheme can use Azure Service Bus queue for communication. This design is similar to one that uses Azure queue except for following finer details. In comparison to Azure storage queue, which only supports messages of type string and binary array, Service Bus queue can support messages of any type. Therefore, it obviates the need for an explicit type conversion. While the Service Bus queue supports the maximum message size of 256KB (in comparison to 64KB for Azure storage queue), the maximum queue size is only 5GB (which is 100 TB for Azure Storage queue). The disadvantage of this service is that its latency is ten times greater than that of Azure storage queue. Since Service Bus queue supports first-in-first-out ordered delivery, it alleviates the out-of-order delivery for messages sent from the same instance.
- **Non-blocking communication:** Asynchronous operations or multiple threads can be used to implement non-blocking send and receive calls. In asynchronous send and receive, which is supported with Service Bus queue, the next statement executes before the previous send or receive request is completed. The asynchronous send and receive instantiates a delegate that invokes a method when the operation is completed; so, this callback method can inform MPI Wait routine of the finished operation. Another design variation

uses multiple threads in order to send and receive messages. In this variation, a new thread is launched for every send and receive request.

2.1.2 One-sided Communications

The Service Bus relay service can be utilized to implement one-sided communication methods. In MPI one-sided communication (introduced in MPI-2 standard [20]), only one of the processes initiates the data transmission on sender and receiver processes. Since a Service Bus relay communication supports request-reply communication, where sender makes calls to service operations and waits for a response from the service without explicit participation of the receiver, it can be a suitable candidate for implementing MPI one-sided operations. The sender instance calls a service and waits for a response from the service. For MPI GET the service operation response is the requested data; for MPI PUT the request is to put the input data in the requested location; and for MPI Accumulate the service call combines the service call input with the data already present in the receiver part.

2.1.3 Collective Communications

MPI collective communication consists of three groups:

- **Barrier synchronization:** The goal of this group is to synchronize all the processes within a communicator. All the instances in a communicator put a message inside the barrier queue when they reach the synchronization point and then wait for an event. One of the instances inside the communicator (master) polls the barrier queue until it receives *k* messages, where *k* is the number of instances in the communicator. After that, the master instance calls a service in all of the other instances in the communicator in order to unlock them.

Another design option is to use Table storage for synchronization between nodes. Each worker role adds an entity with a message containing worker ID to the table storage, and then waits for the master node to change its message entity value to the number of processes in the communicator. The master node waits until all processes add an entity to the table, and then sends a signal to processes by updating their entity values to the number of processes in the communicator.

- **Data movement collective operations:** Broadcast, gather, and scatter operations are examples of this type of operations. The operations of this group can be implemented using the point-to-point send and receive operations. For example, the broadcast operation can be implemented by sending a message to the queues of all other compute instances in the communicator. The time of sending to all other instances can be overlapped using asynchronous send operation or multithreading. The other option is to use Service Bus topics and subscriptions. Service Bus topics and subscriptions offer a one-to-many communication pattern. In this method, there is a topic for each communicator. All the compute instances in the communicator are subscribed to the topic. When a compute instance sends a message to a topic, the message is available to each subscribed instance. Also, subscribed instances can define a filter for received messages. For example, they can filter messages so that they receive only messages from the senders other than themselves (i.e., the sender ID differs from that of the subscribed instance).
- **Global computation:** The reduction operation is one of the operations of this group. One method for reduction operation is to send the data of all compute instances to the receiver instance. Then, the receiver instance applies the reduction operation on the received data. The receiver instance distinguishes these data from the other data in the queue using a

type of messaging called session enabled messaging where all the data related to reduction operation are given identical session ID. Therefore, the receiver receives messages with the same session ID, consecutively.

2.2 Interface and Hardware Configuration

The steep learning curve faced by developers who want to write applications for the cloud has a lot to do with the interface provided by cloud vendors. We have worked with a number of cloud vendors and we invariably found the interface to be overwhelming. We firmly believe that the nomenclature and accessibility must be abstracted out to reduce the complexity and to allow developers to seamlessly work with the cloud platforms. A terminal-based interactive shell, which provides easy to use bindings, can provide developers with means to administer the basic configurations of their project. The terminal shell can connect to developers account on the cloud by asking their configuration settings at first launch and from there it can behave like the well-known terminal Putty to execute commands on the cloud setup. Windows Azure currently allows this but there is no easy way to accomplish this without going through a cumbersome exercise.

A lightweight terminal could allow somewhat similar yet minimalistic, familiar, and simple interface. Based on the account information (credentials), the user could be automatically configured for that machine. This terminal should also provide developers with commands to install required packages and spin-off the Azure roles (VMs) as necessary. There should also be a user friendly way to configure the packages based on the MPI application's requirements. Developers only need to port their legacy MPI code and/or develop new cloud-based MPI code. The code deployment can therefore be friction-less as all the cloud related configurations are already set with necessary libraries and files required for their cloud-based MPI application. Using the

terminal, developers can specify the number of running instances and change them on the go and deploy their application to the cloud.

2.3 Software Configuration

The MPI APIs intended for use on the cloud environment should be similar to traditional MPI in order to reduce the cost of porting. However, C# and PHP, which are the default implementation languages for the Azure platform, along with the configuration of Azure roles brings up some implementation challenges. In traditional MPI, pointers to initial address of the send/receive buffer are passed to MPI functions. However, C# or PHP do not encourage using pointer arithmetic due to type safety and security concerns. This may cause a problem as pointer arithmetic is required to reference the first element of the send/receive buffer.

3 IMPLEMENTATION

The cloudMPI framework is based on the object-oriented principles consisting of the following classes: 1) cMPI, 2) cMPIMessage, 3) Datatype, 4) Comm. These classes, collectively, implement basic MPI routines.

The cMPI class is the core of the implementation. This class offers the methods to facilitate the communication among MPI nodes. All members of this class are declared static.
The cMPI Message class packs the data to be transferred in a message as well as the other attributes of a message that can be used to distinguish messages at the receiver end. A message includes following fields: data, source, tag, msgId and isSmallMessageField. isSmallMessage field is used by the program to determine the location of the data (queue or blob). For large messages this field is set to false. msgId stored in this message is used for large messages and contains the location of the data in the blob.

The class Datatype contains the type information of the data elements of the array that is to be sent/received in the MPI communication operations. In order to provide seamless operation with traditional MPI routines, we use the standard MPI data types, as shown in Fig. 2.1. Current implementation supports all primitive types provided by the C# language; however, it can easily be extended to support any user defined data type.

Comm class: If multiple communicators are required, there will be one queue per node for each communicator. Default COMM WORLD communicator is defined as a static member in the cMPI class.

3.1 Implementing MPI Point-to-Point Operations on Azure Cloud Platform

The basic point-to-point communication operations implemented in cMPI class are send and receive methods. These two methods are declared as follows:

```
int Send(Object buffer, int offset, int count, Datatype type, int dest, int
tag, cMPI Comm MPI COMM)

int Recv(ref object buffer , int offset, int count, Datatype type, int
source, int tag, cMPI Comm MPI COMM)
```

The arguments of these methods closely correspond to conventional MPI send and receive routines except for an additional offset parameter. Unlike the traditional MPI which uses pointers to specify buffer index, the cloudMPI methods get additional offset argument to indicate the data position in the buffer. The buffer wraps the data that is to be sent in a generic object type. The buffer is actually an array of any serializable object. However, the type of the data wrapped by this object should be consistent with the type argument, which is one of the required arguments in Send/Recv methods.

Figure 3.1 shows an MPI sample code in C and its equivalent code for cloudMPI. As shown in this figure, the cloudMPI API is very similar to the C bindings of the conventional MPI. cMPI.Init initializes a channel of communication and assigns a list of nodes to this channel. One queue, dedicated to this channel, is created per node after the initialization. The need to allocate nodes to a channel allows users to broadcast a message to all of the nodes in a communicator. This routine gets Azure storage string as input, which contains the necessary parameters required to access developer storage account in the Windows Azure environment. The receive buffer should be defined as object data type as shown in Fig. 3.1. cMPI.Finalize is used to release the resources occupied by the application.

C Language - Blocking Message Passing Routines Example

```
int numtasks, rank;
int sendBuf[4]={1, 2, 3, 4};
int recvBuf[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  MPI_Send(sendBuf, 4, MPI_INT, 1, 1, MPI_COMM_WORLD);
}
else if (rank == 1) {
  MPI_Recv(recvBuf, 4, MPI_INT, 0, 1, MPI_COMM_WORLD, &Stat);
}
MPI_Finalize();
```

CloudMPI - Blocking Message Passing Routines Example

```
int numtasks, rank;
object sendBuf=new int[4]{1, 2, 3, 4};
object recvBuf=new int[4];

cMPI.Init(connectionString);
cMPI.Comm_size(cMPI.COMM_WORLD, out numtasks);
cMPI.Comm_rank(cMPI.COMM_WORLD, out rank);

if (rank == 0) {
  cMPI.Send(sendBuf, 0, 4, MPI_INT, 1, 1, cMPI.COMM_WORLD);
}
else if (rank == 1) {
  cMPI.Recv(ref recvBuf, 4, MPI_INT, 0, 1, cMPI.COMM_WORLD);
}
cMPI.Finalize();
```

Figure 3.1 Comparison of traditional MPI code with cloudMPI

Figure 3.2 shows the send and receive mechanism between sender and receiver instances. The send routine packs the data to be transferred as well as the other attributes in a message of type cMPI Message. These attributes can be used to distinguish messages at the receiver instance. Then the message is serialized to binary format and converted to byte array (Azure queues can store messages of type string or byte array). Finally a CloudQueueMessage is created from the byte array and sent to the receiver instance queue. On the other side, the receiver routine monitors its queue for new message. The new message is deserialized to an object of type cMPI Message. Then the receiver instance can retrieve required information from the received message.

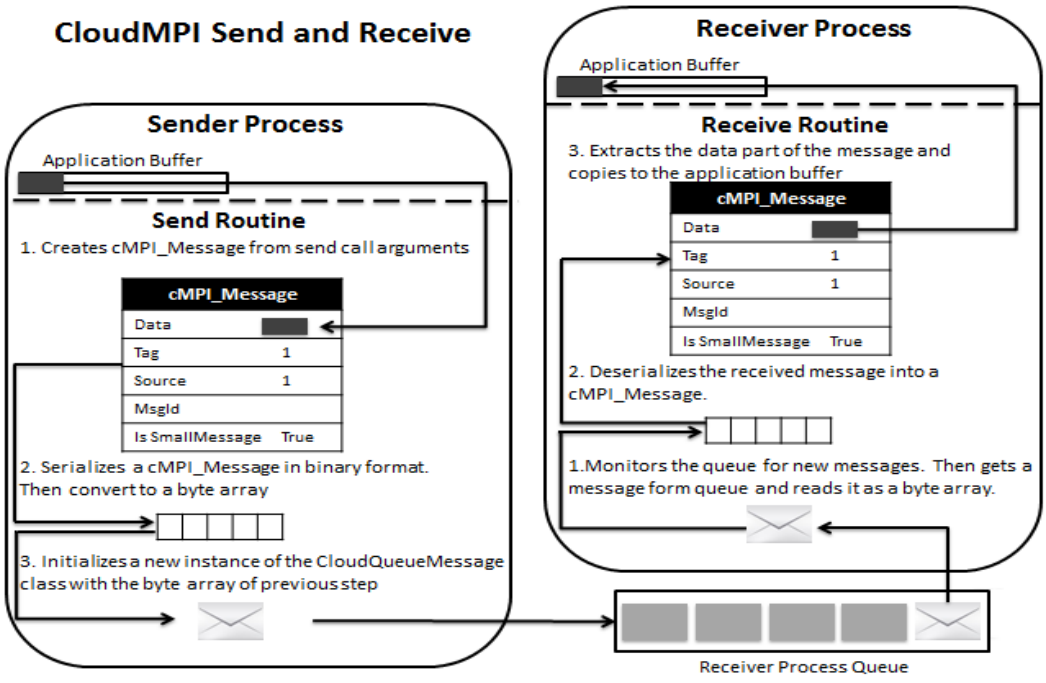

Figure 3.2 Send and receive mechanism in cloudMPI via Azure

3.2 Implementing MPI Collective Operations on Azure Cloud Platform

Broadcast: In the broadcast operation, a root node sends a message to all other nodes in the communicator. cloudMPI supports three implementations of data broadcasting methods: 1) using a combination of Azure queue storage and Azure blob storage; 2) using a combination of Service Bus topic and Azure blob storage; 3) a hypercube-based broadcasting algorithm using Azure storage services (queue and blob). The first approach uses the point-to-point send and receive operations to implement the broadcast operation following the scheme shown in Fig. 3.3 (left). The root node uses the send routine inside a loop to broadcast a message to all other nodes in the communicator. Then, other nodes in the communicator get the message using the receive routine. In the second approach, a Service bus topic is used to broadcast a message between the communicator instances. All instances get subscribed to this topic inside the init routine. When a message is sent to a topic by the root instance, it is then made available to each instance in the

communicator to be received. As the size of the message passes the maximum message size (i.e. 64 KB for storage queue and 256KB for Service Bus topic), the blob storage is used in both methods to communicate the broadcast data and the queue and topic messages indicate the blob ID of the data stored in the blob storage.

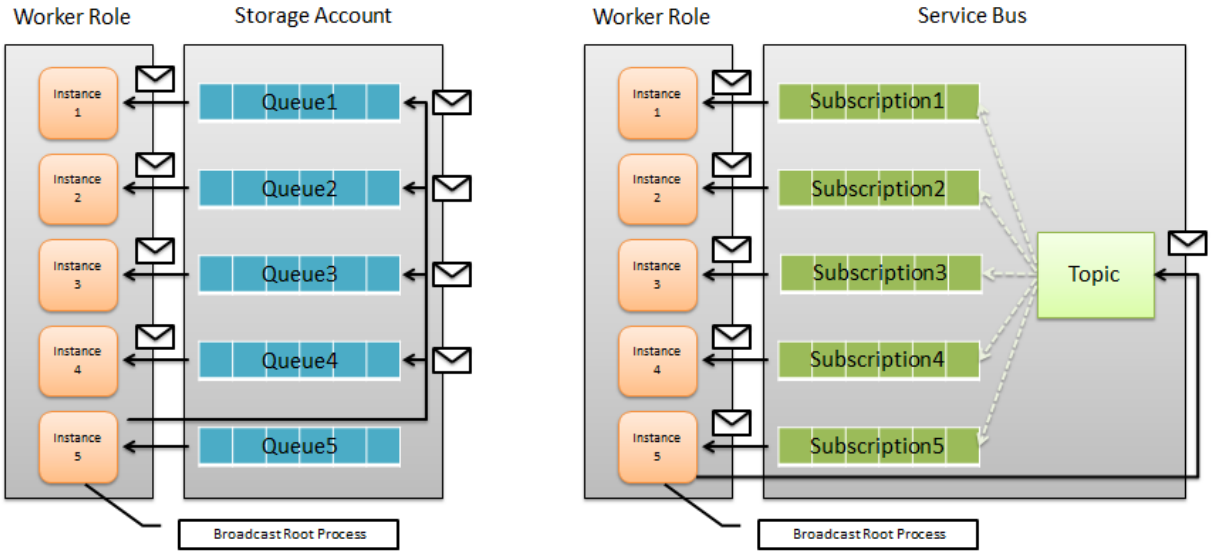

Figure 3.3 Implementation of broadcast using storage queue (left) and Service Bus topic (right)

The third implementation is a hypercube-based broadcast algorithm that uses the storage queue and blob service for data communication. In this communication scheme, the broadcast operation is performed in multiple steps. At each step, a number of nodes act as the message sender; and their neighbor nodes (in an imaginary hyper-cube arrangement) along a single direction of the hyper-cube act as the receiver nodes. In the beginning of the broadcast sequence, the root node sends the message to one of its neighbors and the chain process of broadcast gets initiated. As a result, in the nth time step, 2^n nodes receive the message. In other words, for a communicator of size d, the broadcast operation is performed in $\log(d)$ time steps.

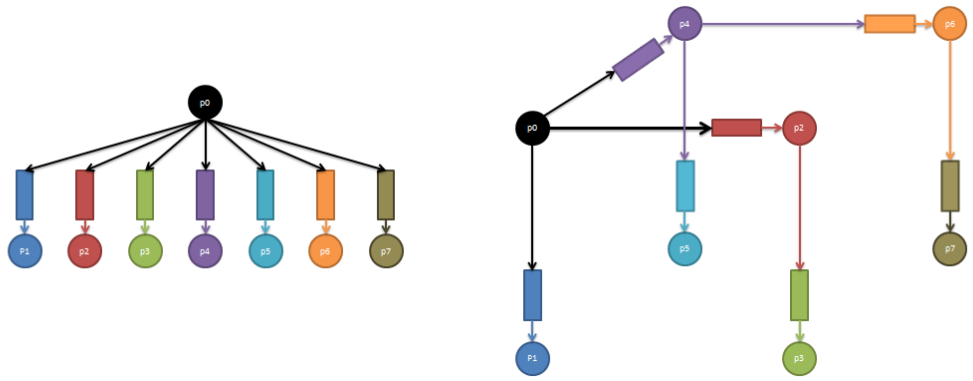

Figure 3.4 Broadcast naive implementation versus hypercube-based implementation

Scatter and Gather: In the scatter operation, a root node sends the segments of a message of size *m* to different nodes, and in the gather operation, the root node receives the segments of a message of size *m* from different nodes and stores them based on the order of the sending nodes rank. In MPI gather/scatter operations, the length of the message received from or sent to other nodes is the same for all nodes. The scatter and gather routines are implemented using the point-to-point send and receive operations. To scatter a message between nodes of a communicator, first, root node extracts *N* segments of the data from the send buffer and sends each segment to the receive queue of the corresponding nodes so that node 1 gets the first segment, node 2 gets the second segment, etc. Then, other nodes in the communicator will wait for the message from the root processor. To gather a message from other nodes of a communicator, first, each node except for the root node sends its data to the root node queue. Then, root node receives the data from the queue and stores them in a local buffer based on the rank order. cloudMPI supports two versions of the scatter method. The only difference between these two versions is that one uses Service Bus queue; and the other, queue storage for communication. Similar to the broadcast routine, the blob storage is used for the transmission of large messages.

For the scatter routine, we also implemented a multi-threaded version of the scatter operation. In this implementation, threads run in parallel to send the messages to other instances' queues.

```
if (root == myId)
{
     Parallel.For(0, size(MPI_COMM), i => {
          if (root != i)
               Send(sendbuf, (i*sendcount)+sendoffset, sendcount, sendtype, i, 0, MPI_COMM);
     });
     Array.Copy(sendbuf as Array, sendoffset+(myId*sendcount), recvbuf as Array, recvoffset, sendcount);
}
if (root != myId)
{
     Recv(ref recvbuf, recvoffset, recvcount, recvtype, root, 0, MPI_COMM);
}
```

Figure 3.5 Multi-threaded scatter operation

Barrier: This operation is used to synchronize operations in MPI nodes. This routine blocks nodes until all the nodes within the communicator reach the synchronization point. We use table storage to implement the barrier method in cloudMPI. All the instances in a communicator insert an entity with a property equal to their ID to the barrier table when they reach the synchronization point and then wait until the master node updates their property. A master node inside the communicator monitors the barrier queue until it receives *k* entity messages, where *k* is the number of instances in the communicator. After that, the master instance updates the property of all the entities to the number of instances in the communicator. Therefore, other processors become unlocked after seeing the change in their property value.

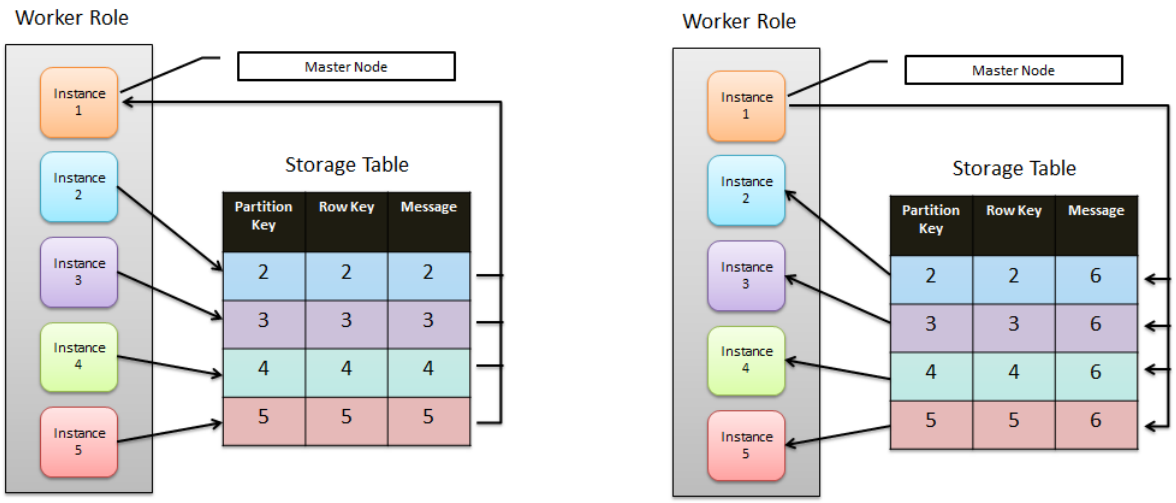

Figure 3.6 Barrier operation implementation

Reduction (MPI_SUM): The reduction operations collect and reduce the values of the send buffer of the different nodes to a single value. For the purpose of this thesis, only MPI_SUM operation of the MPI reduction is implemented. This can be easily extended to other reduction operations. The reduction routine is implemented using the point-to-point send and receive operations. This operation is similar to the gather routine, and the only difference is that the reduction operation computes the reduce operation over elements received from the gather routine.

4 COMMUNICATION BENCHMARKS AND EVALUATION

4.1 Test Environment

In this section, we evaluate the performance of cloudMPI over the Microsoft Azure cloud environment. In order to evaluate the communication performance of cloudMPI, we employ the traditional messaging benchmarks for MPI [21]. Not only we convert these benchmarks for cloud platform using cloudMPI framework, we also create a cluster of virtual machines running a flavor of Linux operating system to run the original MPI benchmarks on barebone machines on the cloud. In both platforms, compute nodes are set to be Azure “small” virtual machine with 1.75GB of memory.

In order to setup our two-processor MPI virtual machine over the barebone cloud infrastructure, a series of configuration steps are needed beyond installing Linux. These steps are instructive for the Azure environment, and hence briefly described below. Master and slave virtual machines have to be created and configured properly. These configurations include installing MPI, configuring network for each machine, loading and compiling the program on each node, etc. We start with creating a master virtual machine with proper installation of the MPI. In our experiments, we use openMPI, which is a popular implementation of the MPI standard. In order to provide SSH connectivity with the slaves, a public/private key pair is generated and the SSH configuration files are modified accordingly. The next step is to create a slave node. MPI installation steps should be performed on this machine as well. To provide the ability to connect to this machine via SSH from the master node, the SSH server should be installed. The key pair generated in the master node is used here to setup the SSH connection. In this step, the program can be loaded on the master and slave nodes and compiled using the MPI package.

To create the desired number of slave nodes, we first save an image of the currently created slave node and then create multiple instances of this image using a shell script and the API provided by the Azure platform. Since each slave instance that gets created this way has a random port number for its SSH connectivity, we use additional shell API commands to query these instances to get their SSH port numbers. These port numbers are then added to the master node's SSH configuration file. Furthermore, the host names supplied in the shell script to create slave nodes are also used to create the MPI configuration file on the master node. MPI uses random TCP port numbers. Therefore, it is difficult to establish the TCP/IP connection between master and slave nodes as this type of communication between virtual machines is not supported by simple Azure network. In order to overcome this problem a virtual network needs to be created on the Azure platform and all the nodes needs to be added to this network. In order to load new code on slave nodes, simple shell scripts can be used to load/compile the code remotely on slave nodes.

4.2 Latency

A ping-pong latency test is often used to characterize the MPI communication performance. The benchmarks measure the network latency between two compute nodes. Node 1 sends a message to node 2, waits for node 2 to receive the message, and then receives a return message from node 2.

For the cloudMPI-based implementation, the benchmarks perform the same operations as for the network benchmarks, but the messages are not sent point to point. Instead, each message is stored in the storage by the sender and the receiver receives the message by reading it from the cloud storage. Thus, there is additional latency introduced in the system that could degrade the performance.

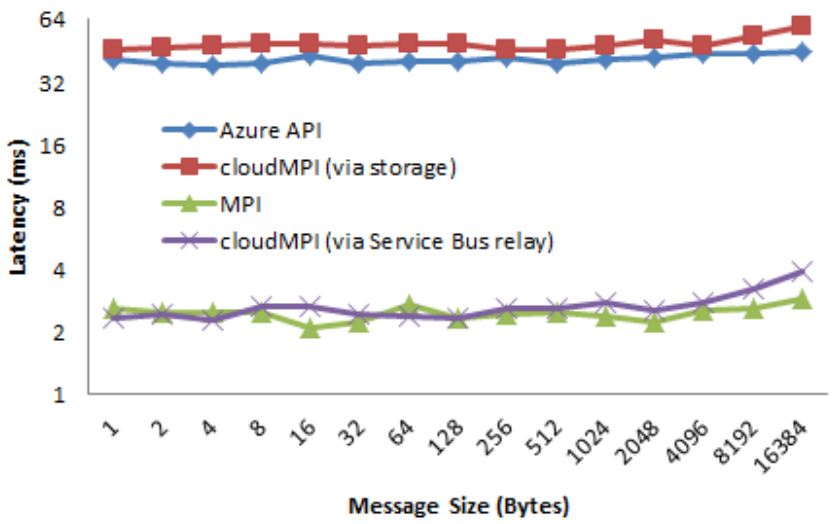

Figure 4.1 Benchmarking MPI performance on Azure cloud environment for small messages

Figure 4.1 shows the benchmark results for short messages. The cloudMPI via Service Bus relay and traditional MPI displays comparable performance in terms of communication latency. Since these two methods do not have the additional overhead of reading and writing from the cloud storage, their performance is superior to the other two methods. To improve the performance for short messages (i.e., message size < 1,460 bytes) useNagleAlgorithm property was turned off.

For communication by large messages, Service Bus relay cannot be used; therefore, as shown in Fig. 4.2, for large messages, we are comparing the benchmark results for the Azure cloud platform for two communication methods, namely MPI and cloudMPI. It is clear that traditional MPI benchmarks on the cluster of virtual machines outperform other implementations. Interestingly, while MPI performs up to 50× better than cloudMPI via storage for short messages, the difference keeps reducing as the message size increases, finally reducing to a difference of less than 3.5×. Comparing the results for short and large messages suggests that for a wide range

of message sizes cloudMPI can be reasonably competitive with traditional MPI while providing the convenience, reliability, and massive scalability of the cloud platform.

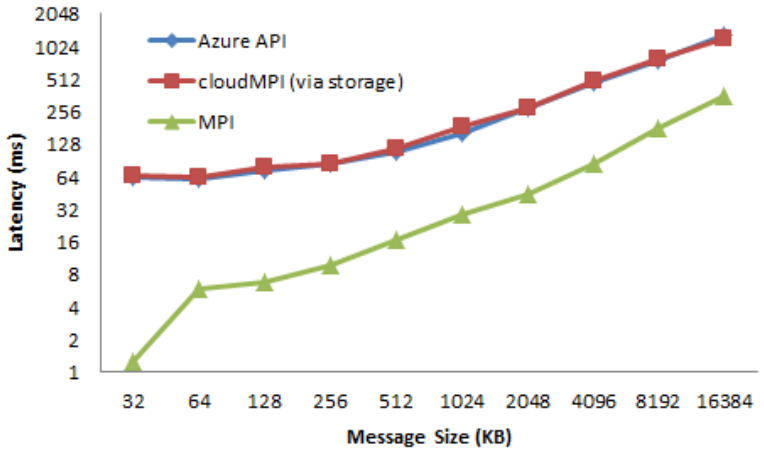

Figure 4.2 Benchmarking MPI performance on Azure cloud environment for large messages

4.3 Performance of Collective Algorithms

We executed the performance tests using our benchmark code on the implementation of barrier, broadcast, and scatter collective operations using cloudMPI and MPI on the cloud clusters. We then analyzed the performance results and the optimal implementation of various collective operations.

4.3.1 Barrier Performance

The benchmark measures the average time to complete a barrier routine repeated for 100,000 times. We run this benchmark on a cluster with 16 single-core nodes (i.e., Azure small nodes). As shown in Fig. 4.3, the benchmark results demonstrate significant performance gap on the average latency between cloudMPI and MPI, especially when the number of compute nodes

increases. The average latency of barrier operation for MPI is almost 49 times as fast as the cloudMPI for a cluster with 16 nodes. When the number of compute nodes is small, the performance gap is smaller but still significant.

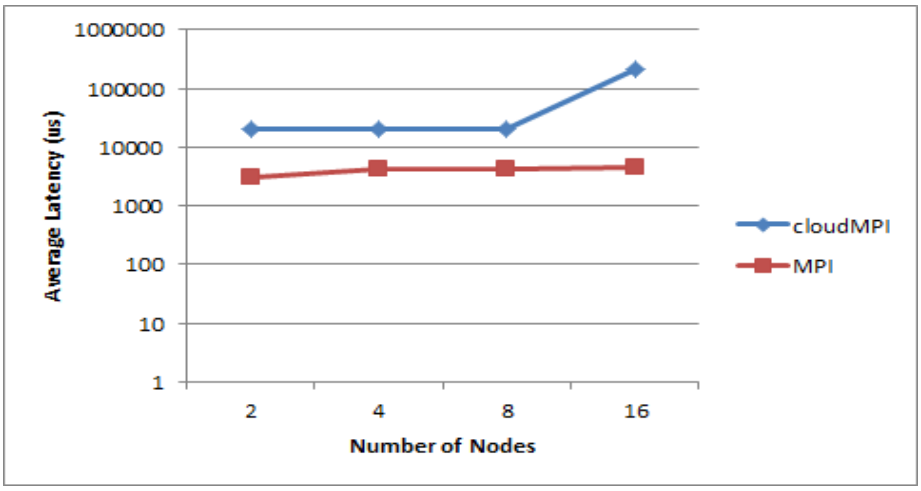

Figure 4.3 . Benchmarking performance of barrier operation on Azure cloud environment

4.3.2 Broadcast and Scatter Performance

Broadcast and scatter benchmarks measure the completion time of the routine averaged over all the compute nodes in the communicator (including both the sender and receivers). We compare the performance of the cloudMPI with that of the traditional MPI. For both scatter and broadcast operations, the result of benchmarks indicates that the traditional MPI outperforms cloudMPI implementations for messages with small size. As the message size increases, the performance gap between cloudMPI and MPI significantly shrinks. Experimental results for cloudMPI scatter operation indicate the implementation based on queue storage is faster than the implementation based on Service Bus queue. For broadcast operation with node sizes of 2 and 4, using queue storage yields faster runtime than using Service Bus topic. As shown in Fig. 4.4,

broadcast operation on 8 nodes — with large enough messages (i.e., message size ≥ 524,288 bytes) — performs faster for the implementation based on Service Bus topic compared to the one using queue storage.

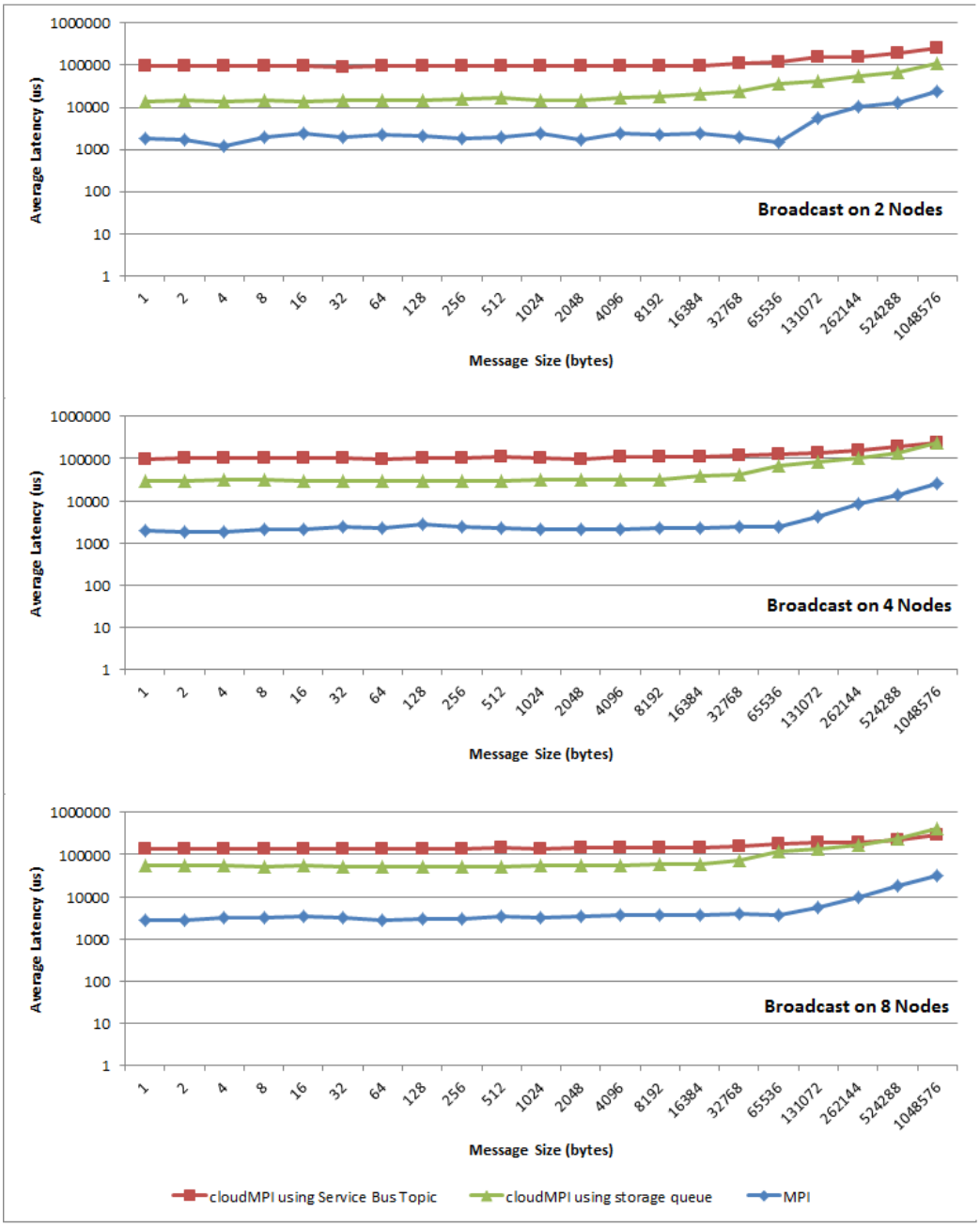

Figure 4.4 Benchmarking performance of broadcast operation on Azure cloud environment

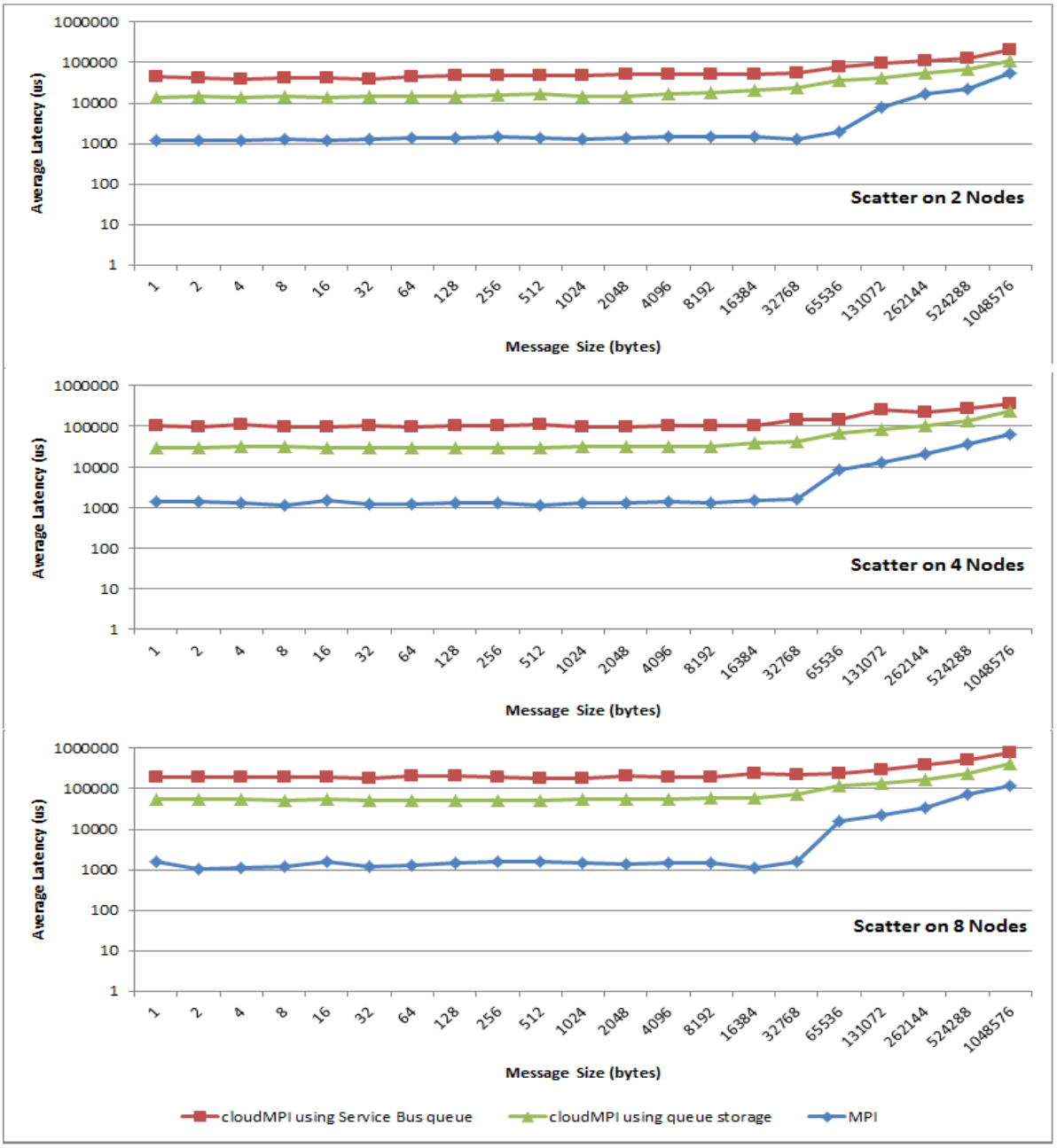

Figure 4.5 Benchmarking performance of scatter operation on Azure cloud environment

Figure 4.6 shows the results of our benchmark test for the scatter operation and compares the performance of MPI and cloudMPI for different number of compute nodes.

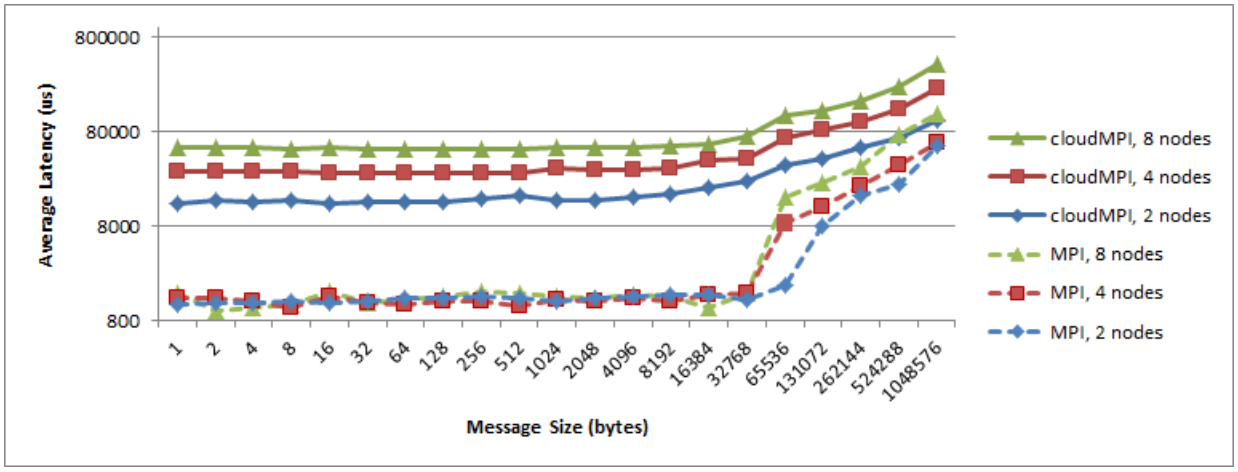

Figure 4.6 Comparing the performance of scatter operation for MPI and cloudMPI

Figure 4.7 compares the performance of the naïve implementation of cloudMPI broadcast (i.e., one sender broadcast the message to every other node in the communicator) with its hypecube-based counterpart.

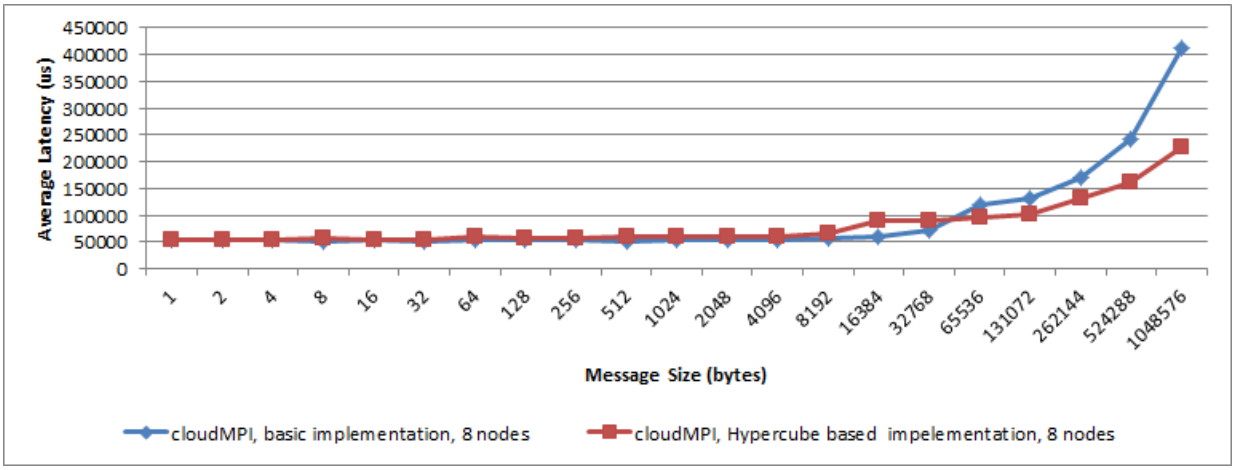

Figure 4.7 Performance of broadcast naïve operation and its hypercube-based counterpart

Finally, as shown in Fig. 4.8, we compare the effectiveness of the multi-thread implementation of the naïve scatter — where the sender routine uses the multi-thread facility of the

Azure platform for its compute nodes — with the simple serial implementation. In the multi-thread version, root process uses multiple threads to send messages to the queues of other processes, simultaneously. As seen in Fig. 4.8, the multi-thread implementation outperforms the serial code by a factor of 1.3 for the message size of 1,048,576 bytes and as the size of the message grows, the disparity between the performances of these two implementations becomes more apparent.

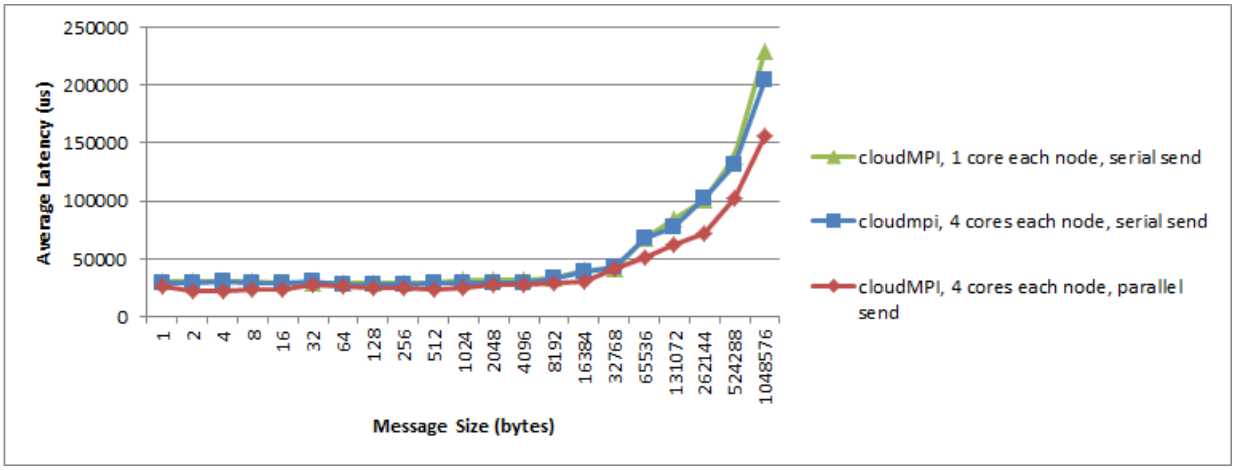

Figure 4.8 Performance of scatter multi-threaded implementation

5 PERFORMANCE OF APPLICATIONS ON THE CLOUD

We analyze the performance implications of cloudMPI for parallel and distributed applications traditionally implemented using MPI. Specifically, we are trying to understand 1) how applications with different communication-to-computation ratios perform using cloudMPI compared to MPI and 2) the steps required to map a traditional MPI implementation to the cloudMPI implementation. To reach our goal, we run our test applications with two different setups: 1) using cloudMPI framework on the Azure worker roles, 2) using traditional MPI on the bare-bone Azure Linux virtual machines configured as a conventional cluster. In both cloud platforms, compute nodes are set to be Azure "small" virtual machines, with 1.75GB of memory.

For our evaluations, we select two MPI applications with different communication and computation requirements, namely, 1) the N-Body particle simulation [22] and 2) Cannon's multiplication algorithm [23, 24].

5.1 N-Body Particle Simulation

The N-Body problem determines the motion of particles over time based on the effect of the forces from other particles. This problem has applications in astrophysics, molecular dynamics, etc. The simulation of this problem is computationally expensive for large N-body systems, which can be sped up by distributing the computationally intensive tasks to multiple compute nodes.

For this experiment, we have used the sample code presented in Ref. [25]. The algorithm used in this experiment implements the following steps to calculate the particle position and velocity at each time step: 1) update positions using velocities, 2) calculate forces, and 3) update velocities. Table 5.1 shows the MPI code in C and its equivalent cloudMPI code. As shown in this figure, the cloudMPI API is very similar to the C bindings of the conventional MPI. Porting

the MPI application to cloudMPI is basically a matter of converting C/C++ code to C# and finding the equivalent syntax for some keywords (such as reading from standard input and writing to standard output) in the Windows Azure environment. We also convert MPI calls to the equivalent calls in cloudMPI. This can be easily achieved using an equivalence table for the function/method calls of MPI and cloudMPI.

Table 5.1 Comparison of N-Body MPI code with cloudMPI

Set up environment

MPI:

```
int n_proc, rank;
MPI_Init( &argc, &argv );
MPI_Comm_size( MPI_COMM_WORLD, &n_proc );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

cloudMPI:

```
int n_proc, rank;
cMPI.Init(connectionString);
cMPI.Comm_size(cMPI.COMM_WORLD, out n_proc);
cMPI.Comm_rank(cMPI.COMM_WORLD, out rank);
```

Set up the data partitioning across processors

MPI:

```
int particle_per_proc = (n + n_proc - 1) / n_proc;
int *partition_offsets = (int*) malloc( (n_proc+1) * sizeof(int) );
for( int i = 0; i < n_proc+1; i++ )
    partition_offsets[i] = min( i * particle_per_proc, n );

int *partition_sizes = (int*) malloc( n_proc * sizeof(int) );
for( int i = 0; i < n_proc; i++ )
    partition_sizes[i] = partition_offsets[i+1] - partition_offsets[i];
```

cloudMPI:

```
int particle_per_proc = (n + n_proc - 1) / n_proc;
int[] partition_offsets = new int[(n_proc + 1)];
for (int i = 0; i < n_proc + 1; i++)
    partition_offsets[i] = Math.Min(i * particle_per_proc, n);

int[] partition_sizes = new int[n_proc];
for (int i = 0; i < n_proc; i++)
    partition_sizes[i] = partition_offsets[i + 1] - partition_offsets[i];
```

Allocate storage for local partition

MPI:

```
int nlocal = partition_sizes[rank];
particle_t *local = (particle_t*) malloc( nlocal * sizeof(particle_t) );
```

cloudMPI:

```
int nlocal = partition_sizes[rank];
Particle[] local = new Particle[n];
```

Initialize and distribute the particles

MPI:

```
set_size( n );
if( rank == 0 )
    init_particles( n, particles );
MPI_Scatterv( particles, partition_sizes, partition_offsets, PARTICLE, local,
nlocal, PARTICLE, 0, MPI_COMM_WORLD );
MPI_Allgatherv( local, nlocal, PARTICLE, particles, partition_sizes, parti-
tion_offsets, PARTICLE, MPI_COMM_WORLD );
```

cloudMPI:

```
set_size(n);
if (rank == 0)
    init_particles(n, particles);

cMPI.scatterv(particles, 0, partition_sizes, partition_offsets, ref local, 0,
nlocal, 0, cMPI.COMM_WORLD);
cMPI.AllGatherv(local, 0, nlocal, ref particles, 0, partition_sizes, parti-
tion_offsets, cMPI.COMM_WORLD);
```

Simulate a number of time steps

MPI:

```
for( int step = 0; step < NSTEPS; step++ )
{
    //  compute all forces
    for( int i = 0; i < nlocal; i++ )
    {
        local[i].ax = local[i].ay = 0;
        for (int j = 0; j < n; j++ )
            apply_force( local[i], particles[j] );
    }

    //  move particles
    for( int i = 0; i < nlocal; i++ )
        move( local[i] );

    //  collect all global data locally (not good idea to do)
    MPI_Allgatherv( local, nlocal, PARTICLE, particles, partition_sizes, par-
tition_offsets, PARTICLE, MPI_COMM_WORLD );


    //  save current step if necessary
    if( fsave && (step%SAVEFREQ) == 0 )
        save( fsave, n, particles );
}
```

cloudMPI:

```
for (int step = 0; step < Constants.NSTEPS; step++)
{
    //  compute all forces
    for (int i = 0; i < nlocal; i++)
    {
        local[i].ax = local[i].ay = 0;
        for (int j = 0; j < n; j++)
            apply_force(local[i], particles[j]);
    }

    //  move particles
    for (int i = 0; i < nlocal; i++)
        move(local[i]);

    //  collect all global data locally (not good idea to do)
    cMPI.AllGatherv(local,0, nlocal, ref particles,0, partition_sizes, parti-
tion_offsets, cMPI.COMM_WORLD);

    //  save current step if necessary
    if ((step % SAVEFREQ) == 0)
        save(sw, n, particles);
}
```

The simulation results for both platforms (i.e., MPI and cloudMPI) with different number of particles and different number of nodes are shown in Figs. 5.1 and 5.2. In this application, the cloudMPI implementation uses the Service Bus queue and the Service Bus topic for small messages and falls back to the blob storage for large messages (> 256kB). It should be noted that for the node size of 1, we use a serial implementation of the problem for the both MPI and cloudMPI cases and no message-passing mechanism is used. Therefore, comparison of the performance of our code for the single-node case provides a measure of the performance of the hardware used in each node. It can be seen from Figs. 5.1 and 5.2 that the Windows Azure small node (used in the cloudMPI implementation) outperforms the Linux virtual machine small node provided by the Azure platform. This observation is in agreement with previous reports [26]. Therefore, in our comparisons for the multi-node cases, we should be aware of this performance disparity between the two types of the nodes (i.e., Windows worker roles and Linux VMs) and draw our conclusions regarding the performance of the MPI and cloudMPI with this disparity in mind.

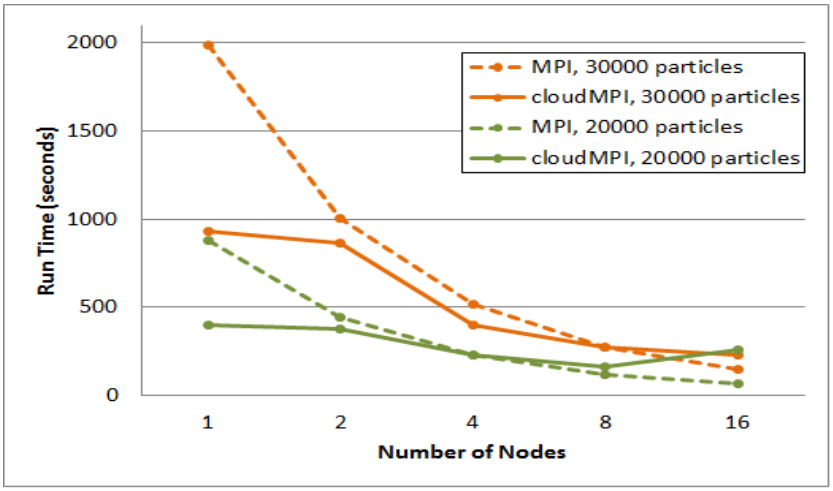

Figure 5.1 Performance of N-Body simulation for 20,000 and 30,000 particles

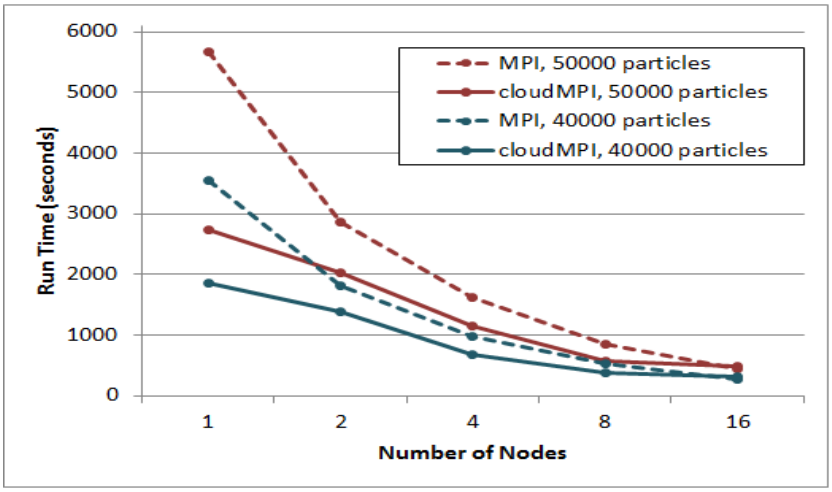

Figure 5.2 Performance of N-Body simulation for 40,000 and 50,000 particles

The algorithm for the N-Body simulation computes all the interactions of N particles on p compute nodes in a load-balanced fashion (i.e., each node performs the computations for N/p particles). Each compute node performs $O(N^2/p)$ force evaluations in each time step, whereas the communication scales linearly with N [$O(N)$] for each compute node. As shown in Fig. 5.1, cloudMPI outperforms MPI for small number of nodes (e.g., < 4 nodes) as the computation time overtakes the communication time. Figure 5.3 shows the performance of the two implementations with respect to the number of particles for a fixed number of nodes. As seen in this figure, as the computation to communication ratio increases for sufficiently large N, the cloudMPI implementation outperforms the traditional MPI.

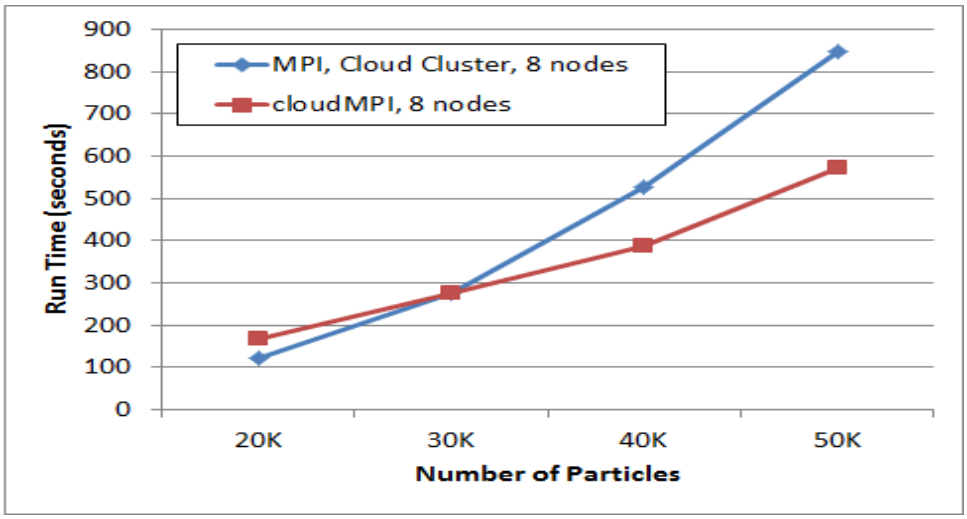

Figure 5.3 Performance of N-Body simulation on 8 nodes

5.2 Cannon's Multiplication Algorithm

Cannon's algorithm distributes square submatrices of size n/p of original two $n \times n$ matrices A and B among the p compute nodes. These submatrices are aligned to compute nodes in a way that the corresponding square submatrices (from matrices A and B) at each compute node can be multiplied together locally. The compute nodes are organized in a mesh arrangement, and in each iteration, each compute node shifts its current submatrix of A to its left neighbor compute node and its current submatix of B to the upper neighbor compute node. This shifting operation is performed circularly for the leftmost and the topmost nodes in the mesh. Following this step, each node multiplies the new submatrices and adds it to a result matrix. Therefore, each compute node spends $O(n^3/p)$ time on the computation step (i.e., multiplication of local submatrices) and $O(n^2/\sqrt{p})$ on the communication step (i.e., shifting the submatrices to the neighbor nodes).

We run the cloudMPI implementation of the algorithm for clusters of up to 16 worker role nodes. Figure 5.4 shows the cloudMPI performance on different mesh sizes for a 4800×4800 matrix.

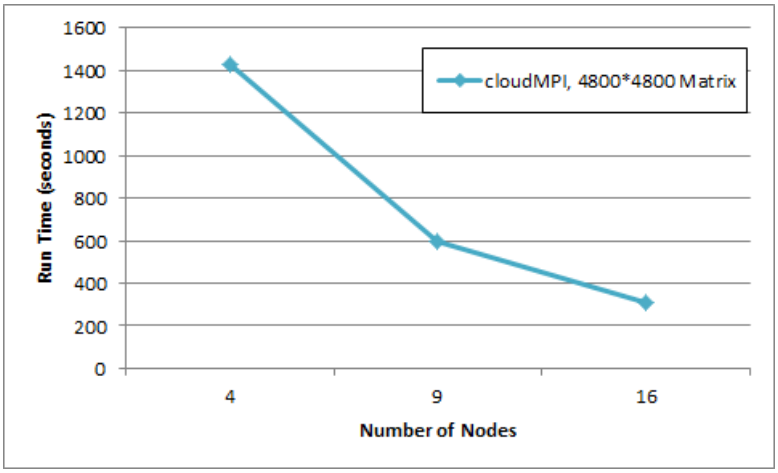

Figure 5.4 cloudMPI performance for a 4800×4800 matrix

Figure 5.5 compares the performance of algorithm for cloudMPI and MPI for different matrix sizes on a cluster with 9 nodes (Azure worker roles for the cloudMPI and Linux VMs for the MPI). In this experiment, cloudMPI implementation uses the Service Bus queue for its communication step. As expected, higher computation to communication ratios results in better performance for the cloudMPI implementation.

As we discussed earlier, the Windows environment of Azure platform outperforms its Linux VMs. Taking this fact into consideration, two conclusions can be made based on the results of our experiments. First, we conclude that the applications' communication intensity is correlated with the performance gap between the cloudMPI and MPI and the MPI implementation outperforms the cloudMPI implementation as the communication intensity of the algorithm increases. Second, we conclude that for the applications with higher computation intensity cloudMPI implementation performance is comparable to and in certain cases better than the performance of the MPI implementation. Therefore, despite lower bandwidth of the cloud environment, cloudMPI can be used to implement CPU intensive applications and provide the scalability, maintenance, and cost advantage of the cloud platform for the users of this platform.

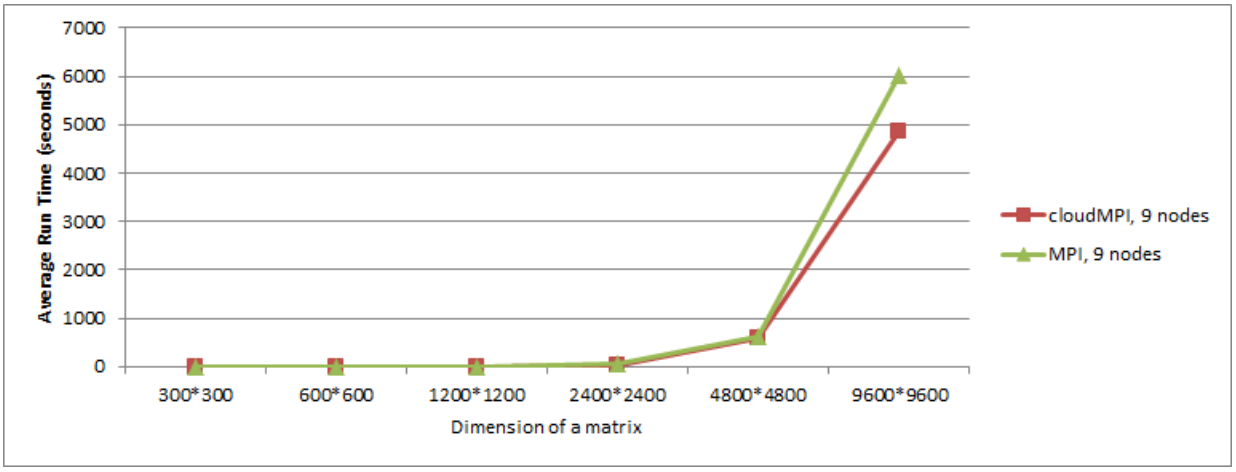

Figure 5.5 Performance of Cannon's algorithm for cloudMPI and MPI

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

To introduce a framework for implementing new MPI-style applications, as well as to port MPI-based legacy applications to the Azure cloud platform was the main focus of this thesis. In this work, we conducted a comprehensive evaluation of the performance of communications on Window Azure. We implemented different MPI point-to-point and collective operations using Windows Azure storage and messaging components and assessed the feasibility of replacing traditional MPI with similar interface tailored for cloud usage for running tightly coupled MPI programs. We tested the cloudMPI using available microbenchmarks for MPI to evaluate the performance of each MPI routine separately and also using two different applications to see the effect of the cloud low bandwidth on the overall computation. Experimental results indicate that the cloudMPI performance is comparable to the traditional MPI as computation overhead exceeds the communication overhead. Therefore, cloudMPI can provide an acceptable performance for applications with high computation to communication ratio. Users can benefit from the advantages of the cloud environment's low cost, scalability, ease of management and job submission, capability of interactive job execution, and instant reconfiguration which are effective factors for HPC users' choice of platforms.

6.2 Future Work

One of the major challenges for migrating HPC applications to the cloud environment is fault tolerance. Some common reasons for failure in the Windows Azure cloud environment are application failure (e.g., poor exception handling in the code), routine maintenance activities, and hardware failure. HPC applications are more prone to failure due to two reasons: 1) As HPC applications usually require a large number of processors and virtual instances and relatively con-

siderable communication links, the probability of failure increases for such applications; 2) Virtual machines, which are used for arranging HPC clusters on the cloud, are more likely to crash as a result of resource sharing and contention. For some failures, local data written to the local virtual machine disks are lost, which consequently causes the application to fail. Therefore, handling transient compute nodes failures is essential for building reliable cloud-native HPC applications.

We also believe that implementing a translator to automatically convert an MPI application to cloudMPI will significantly reduce the overhead of manually porting the legacy MPI code to cloudMPI and could be a valuable research project as a future work.

Providing terminal environment for convenient managing (i.e., initial setup, rescaling, monitoring, etc.) and deploying cloudMPI applications is also another area of focus.

REFERENCES

1. *MPI Documents*. 21 September 2012; Available from: http://www.mpi-forum.org/docs/docs.html.
2. He, Q., 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. ACM.
3. Wilder, B., *Cloud Architecture Patterns: Using Microsoft Azure*. 2012: O'Reilly Media, Inc.
4. Mizonov, V. and S. Manheim. *Windows Azure Queues and Windows Azure Service Bus Queues - Compared and Contrasted*. January 21, 2014.
5. Evangelinos, C. and C. Hill, *Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2.* ratio, 2008. **2**(2.40): p. 2.34.
6. Jackson, K.R., L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright. *Performance analysis of high performance computing applications on the amazon web services cloud*. in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. 2010. IEEE.
7. Zhai, Y., M. Liu, J. Zhai, X. Ma, and W. Chen. *Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications*. in *State of the Practice Reports*. 2011. ACM.
8. Ekanayake, J. and G. Fox, *High performance parallel computing with clouds and cloud technologies*, in *Cloud Computing*. 2010, Springer. p. 20-38.
9. Expósito, R.R., G.L. Taboada, S. Ramos, J. Touriño, and R. Doallo, *Performance analysis of HPC applications in the cloud.* Future Generation Computer Systems, 2013. **29**(1): p. 218-229.
10. Lu, W., J. Jackson, and R. Barga. *AzureBlast: a case study of developing science applications on the cloud*. in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 2010. ACM.
11. Gunarathne, T., T.L. Wu, J.Y. Choi, S.H. Bae, and J. Qiu, *Cloud computing paradigms for pleasingly parallel biomedical applications.* Concurrency and Computation: Practice and Experience, 2011. **23**(17): p. 2338-2354.
12. Agarwal, D. and S. Adviser-Prasad, *Scientific high performance computing (hpc) applications on the azure cloud platform.* 2013.
13. Barga, R.S., J. Ekanayake, and W. Lu. *Project Daytona: data analytics as a cloud service*. in *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. 2012. IEEE.
14. Gunarathne, T., B. Zhang, T.-L. Wu, and J. Qiu, *Scalable parallel computing on clouds using Twister4Azure iterative MapReduce.* Future Generation Computer Systems, 2013. **29**(4): p. 1035-1048.
15. Redekopp, M., Y. Simmhan, and V.K. Prasanna. *Optimizations and Analysis of BSP Graph Processing Models on Public Clouds*. in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 2013. IEEE.
16. Malewicz, G., M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. *Pregel: a system for large-scale graph processing*. in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010. ACM.

17. Agarwal, D., S. Karamati, S. Puri, and S. Prasad. *Towards an MPI-like framework for the Azure cloud platform*. in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2014. IEEE.
18. *MPI Point-to-Point*. Available from: https://www.cac.cornell.edu/VW/MPIP2P/.
19. Buntinas, D., 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. IEEE.
20. Huss-Lederman, S., B. Gropp, A. Skjellum, A. Lumsdaine, B. Saphir, and J. Squyres, *Mpi-2: Extensions to the message-passing interface.* University of Tennessee, available online at http://www. mpiforum. org/docs/docs. html, 1997.
21. Liu, J., B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D.K. Panda, and P. Wyckoff, *Microbenchmark performance comparison of high-speed cluster interconnects.* Micro, IEEE, 2004. **24**(1): p. 42-51.
22. Gropp, W., E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. 1999: MIT press.
23. Cannon, L.E., *A CELLULAR COMPUTER TO IMPLEMENT THE KALMAN FILTER ALGORITHM*. 1969, DTIC Document.
24. Grama, A., *Introduction to parallel computing*. 2003: Pearson Education.
25. *Parallelize Particle Simulation*. Available from: http://www.cs.berkeley.edu/~bvs/cs267_hw2/.
26. Ristov, S. and M. Gusev. *Performance vs cost for windows and linux platforms in Windows Azure cloud*. in *Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on*. 2013. IEEE.