Performance and monetary cost optimizations for HPC applications in the cloud

Gong, Yifan

2016

Gong, Y. (2016). Performance and monetary cost optimizations for HPC applications in the Cloud. Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/69201>

<https://doi.org/10.32657/10356/69201>

NANYANG
TECHNOLOGICAL
UNIVERSITY

**PERFORMANCE AND MONETARY COST
OPTIMIZATIONS FOR HPC
APPLICATIONS IN THE CLOUD**

GONG YIFAN

**Interdisciplinary Graduate School
IGS-NEWRI
2016**

PERFORMANCE AND MONETARY COST OPTIMIZATIONS FOR HPC APPLICATIONS IN THE CLOUD

GONG YIFAN

**Interdisciplinary Graduate School
IGS-NEWRI**

A thesis submitted to Nanyang Technological University
in fulfillment of the requirement for the degree of
Doctor of Philosophy

2016

Abstract

Benefit from cloud computing, high performance computing (HPC) tasks can be performed on many virtual machines instead of a physical cluster. The users can move HPC tasks from private clusters to public clouds. The unique virtualized environment calls for careful examination on existing HPC software stacks for efficiency. Because of the pay-as-you-go nature, performance and monetary cost optimizations are significant in not only improving the productivity but also reducing the total ownership cost. Our research focuses on both of them for network intensive applications like scientific computing and big data.

For performance optimizations, we first evaluate the performance of Message Passing Interfaces (MPI), including broadcast, reduce, gather and scatter, in cloud. For existing MPI implementations, the information of network topology becomes very important in performance optimizations. However, in the cloud environment, virtualization not only hides the network topology information from the users, but also causes traffic interference and dynamics to network performance. In this case, topology-aware optimizations are not useful. Therefore, we develop novel network performance aware algorithms for MPI collective communication operations. We further implement two common applications, namely conjugate gradient (CG) and N-body. We have conducted our experiments with two complementary methods (on Amazon EC2 and simulations). Our evaluation results demonstrate 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.

Furthermore, we find network interference is a norm in the cloud networking, and has a large performance impact on network-intensive applications that are deployed and run in virtual clusters. Many existing network optimizations that rely on topology information or stable network performance in physical clusters are no longer effective for virtual clusters as we mentioned above. So we propose a novel network performance model to decouple the constant and volatility components from the dynamic network performance of virtual clusters. The notions of constant and volatility represent the long-term and transient network performance factors of the virtual cluster, respectively. We develop

the model based on RPCA (Robust Principal Component Analysis). In order to adapt the significant changes in the network performance, we leverage theoretical properties of RPCA. This seemingly simple model brings predictability and optimization opportunities to virtual clusters. With the predicted constant and volatility, we are able to determine the effectiveness of network optimizations in the virtual cluster, e.g., if the volatility is too large, network optimizations are useless. Our experiments on both simulations and Amazon EC2 show the effectiveness of our model, and significant performance improvement of our approach on guiding the network optimizations.

For monetary cost optimizations, we observe that Amazon EC2 spot instances give us a chance to reduce monetary costs of MPI-based HPC applications compared with on-demand instances only. So we formulate a monetary cost optimization problem for MPI applications and leverage both on-demand and spot instances to guarantee deadline constraint. We developed a cost model guided approach to combine checkpoint and replication in an adaptive manner. We implement our model on Amazon EC2 and evaluate its efficiency with NPB benchmarks. We also utilize one real-world application, LAMMPS. The experimental results show that 1) due to the dynamics of spot price, it is necessary to adaptively choosing checkpoint and replication techniques and 2) our approach can reduce the monetary cost than the state-of-the-art algorithm [1] by 20% on average (up to 40%) while satisfying the deadline requirement.

Acknowledgments

Firstly, I express my thanks to my previous supervisor Dr. Bingsheng He, my current supervisor Dr. Xueyan Tang, my co-supervisor Dr. Xiaosheng Qin and my mentor Dr. Law Wing-Keung Adrian for their continuous, remarkable and patient guidance throughout my four year's study. Their expertise and invaluable advice has left a profound impact on my work, especially the way I think of and solve new problems.

I would like to thank my friends and colleagues in IGS-NEWRI and SCSE for their friendship and help along the way. Finally, my work is supported by the scholarship from NEWRI and IGS(Interdisciplinary Graduate School).

Contents

Abstract	i
Acknowledgments	iii
List of Figures	vii
1 Introduction	1
1.1 Background for Cloud Computing	1
1.2 Motivation	4
1.3 Contributions	6
1.4 Structure of the Thesis	7
2 Background and Related Work	9
2.1 Cloud Dynamic Study	9
2.1.1 Performance Dynamics Study	9
2.1.2 Price Dynamics and Spot Pricing Model	10
2.2 Performance and Monetary Cost Optimization	14
2.2.1 Network Performance Aware Optimization	14
2.2.2 Monetary Cost Optimizations	17
2.3 Relevant Techniques	18
2.3.1 Robust Principal Component Analysis	18
2.3.2 Fault Tolerance in MPI	19
3 Network Performance Aware MPI Collective Communication Operations in the Cloud	21
3.1 Problem Overview	21
3.2 Algorithm Design	23
3.2.1 Network Performance Matrices	23
3.2.2 Network Performance Hierarchy	24

3.2.3	Network Performance Aware Operations	28
3.3	Evaluations	32
3.3.1	Experimental Setup	32
3.3.2	Results on Collective operations	33
3.3.3	Results on Long-term Behavior	37
3.3.4	Results on Applications	38

4 Finding Constant From Change: Revisiting Network Performance Aware Optimizations on IaaS Clouds 41

4.1	Problem Overview	41
4.2	Problem Definition	42
4.3	RPCA-based Approach	46
4.3.1	Approach Design	46
4.3.2	Implementation Details	49
4.4	Evaluations	50
4.4.1	Experimental Setup	51
4.4.2	Calibrating Overhead	54
4.4.3	Parameter Study	55
4.4.4	Results with Amazon EC2	56
4.4.4.1	Results on basic applications	56
4.4.4.2	Results on real-world applications	59
4.4.4.3	Detailed Performance Comparison	59
4.4.5	Simulations on Large-scale Cluster	61

5 Monetary Cost Optimizations for MPI-Based HPC Applications on Amazon Clouds: Checkpoints and Replicated Execution 65

5.1	Problem Overview	65
5.2	Problem Formulation	67
5.2.1	Model Overview and Assumptions	68
5.2.1.1	Model Overview	68
5.2.1.2	Assumptions	68
5.2.2	Problem Definition	69
5.2.2.1	Estimating Expected Monetary Cost	70
5.2.2.2	Estimating Expected Execution Time	72

5.3	Model Optimization	72
5.3.1	On-demand Type Selection	73
5.3.2	Two-Level Optimization	75
5.3.2.1	Difficulty Analysis	75
5.3.2.2	Algorithm Details	76
5.3.3	Update Maintenance	79
5.3.4	Implementation Details	81
5.4	Evaluations	83
5.4.1	Experimental Setup	83
5.4.2	Parameter Study	85
5.4.3	Results on Amazon EC2	86
5.4.3.1	Comparison with state-of-the-art algorithm	86
5.4.3.2	Comparison with heuristic algorithms	90
5.4.3.3	Detailed study for SOMPI	91
5.4.4	Results with Simulation	92
5.4.4.1	Failure Rate Function and Model Accuracy	92
5.4.4.2	Comparison with individual fault-tolerance mechanisms	93
6	Conclusion and future work	97
6.1	Conclusion	97
6.2	Future Work	98
6.2.1	Optimization on Exascale Cloud Environments	99
6.2.2	Optimization in Geo-distributed Cloud	99
6.2.3	Optimization for Scientific Workflows in IaaS Cloud	100
	Bibliography	101

List of Figures

1.1	MPI Broadcast and Binomial Tree Algorithm	4
1.2	A tree-topology network	5
2.1	CDF of pair-wise network bandwidth for 196 medium instances in Amazon EC2 . .	10
2.2	Temporal behavior of the point-to-point performance between two medium instances: latency (left) and bandwidth (right).	11
2.3	The gap of the point-to-point communication between two medium instances: latency (left) and bandwidth (right).	11
2.4	Spot price history of Amazon EC2 over three days	12
2.5	Histogram of m1.medium spot price in us-east-1a zone in four consecutive days . . .	12
2.6	An example of weight matrix as all-link network performance and FNF tree structure	16
3.1	Network performance aware reduce operation	29
3.2	The construction of the communication tree in gather: (a) construction by distance; (b) network performance hierarchy (top) and the result communication tree (bottom).	31
3.3	The network performance hierarchy for 64 medium instances	34
3.4	Performance improvement for 64 medium instances	35
3.5	CDF of individual process elapsed time for scatter	35
3.6	The distribution of broadcast and scatter performance with different virtual clusters. .	36
3.7	Probability density distribution with Monte Carlo simulations (broadcast and scatter)	37
3.8	Long-term performance of Bcast and Scatter	38
3.9	Performance comparison of applications on 64 medium instances: N-Body and CG. .	39
4.1	An example of calculating $N_A = N_D + N_E$ with RPCA	48
4.2	Overhead of calibrating temporal performance matrix	55
4.3	Overall performance comparison for the three applications on Amazon EC2 (all nor- malized to the average of Baseline)	57

4.4	Overall performance comparison with baseline for the different number of instances on Amazon EC2	58
4.5	Performance comparison of real-world applications on 196 medium instances: N-Body and CG.	58
4.6	The impact of N_E in the virtual cluster	60
4.7	Overall performance comparison with $Norm(N_E)=0.2$	62
4.8	The impact of network interference on $Norm(N_E)$	62
4.9	Performance Comparison in ns-2-based simulations	62
5.1	Overview of the optimization algorithm.	73
5.2	Changing trends of the failure rate function $f_i(P_i, t_i)$ and the expected spot price $S_i(P_i)$ in us-east-1a zone.	78
5.3	Monetary cost comparison in different deadline	95
5.4	Monetary cost comparison with heuristic algorithms	96
5.5	Monetary cost in different Deadline: BT, FT and BTIO	96
5.6	Monetary cost comparison with fault-tolerance mechanisms	96

Chapter 1

Introduction

This chapter first introduces the background for cloud computing. Then we present the major motivations, including diverse cloud offerings, network heterogeneity, network interference and cloud prices dynamics. Finally, we present the contributions, followed by the organization of the thesis.

1.1 Background for Cloud Computing

Performance and monetary cost are two important features problem for HPC applications in IaaS clouds. We first explain two concepts, including Infrastructure-as-a-Service (IaaS) clouds and Message Passing Interface (MPI).

IaaS Clouds. Infrastructure-as-a-service (IaaS) clouds become a popular computing infrastructure for many distributed applications. For example, many scientific applications have been deployed in Google Compute Engine, Windows Azure and Amazon EC2, including life sciences [2, 3], physics [4, 5, 6, 7], and others listed in Amazon case studies [8].

Virtualization is the enabling technology for resource sharing and consolidation in IaaS clouds. Example virtualization systems include VMWare, Xen and Hyper-V. Virtualization hides the details on the underlying hardware. Based on virtualization techniques, IaaS cloud offers computing resources as virtual instances. The providers provide different types of instances. The instances have different capabilities, including network bandwidth, I/O speed, RAM size and CPU speed to satisfy

Type	Network	CPU cores	Disk(GB)	Mem(GB)	Price
m1.small	Low	1	160	1.7	0.06
m1.medium	Moderate	2	410	3.75	0.12
m1.large	Moderate	4	840	7.5	0.24
m1.xlarge	High	8	1680	15	0.48

Table 1.1: Capabilities and prices(\$/hr) of on-demand instances from us-east-1a region in Amazon EC2

different demands. According to different users' requirements, they are able to select the most suitable instance type.

Cloud has evolved into an economic market. Compared with the traditional cluster and grid computing environments, IaaS cloud offers on-demand virtual machines in the *pay-as-you-go* manner. Anyone with a credit card can buy the computational resources and storage from public cloud providers. Table 1.1 shows the capabilities and prices from four types of on-demand instance provided by Amazon EC2 in US east region. Amazon EC2 utilizes the hourly pricing model, which means the instances are charged by hours. Besides on-demand instances, which charge the users with a fixed rate, Amazon EC2 offers spot instances, whose prices are mainly decided by the supply and demand in the market. To use spot instances, users are required to set the appropriate bid price, which they are willing to pay. Once the instance is launched, the bid price is fixed. If bid price is higher than the spot price, the instance can be successfully launched and run; otherwise it waits. Amazon publishers update the spot price periodically and launch the waiting instances whose bid prices exceed the current spot price and terminate the instances whose bid prices are lower. Usually, the fixed price of on-demand instances is much higher than the spot price of the same type. For these features, leveraging spot instances appropriately can greatly reduce the monetary cost. However, once the spot price is higher than the bid price, spot instance will be terminated(i.e., out-of-bid events). It can lead to fault-tolerance problem.

MPI. Recently, we have witnessed many emerging high performance computing (HPC) applications developed and hosted in the cloud [9, 8, 10, 11]. Message Passing Interface (MPI) is the key

(a) Example of MPI Broadcast

(b) Binomial Tree Algorithm

Figure 1.1: MPI Broadcast and Binomial Tree Algorithm

programming paradigm for developing HPC applications [12, 13]. Many large-scale scientific applications are MPI-based applications, and recently have been migrated to public clouds [14, 15, 9].

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.

MPI supports point-to-point and collective communications. The former function involves the communication between two specific processes, and the latter one involves the communication within a *Communicator*.

MPI collective operations includes four major operations, namely broadcast, reduce, gather and scatter. Broadcast (MPI_Bcast) takes data from one process and sends it to all other processes. A reverse operation is reduce (MPI_Reduce) which takes data from all processes, performs aggregation, and stores the results on one process. Scatter (MPI_Scatter) is to divide a large array into some smaller parts that equal to the number of processes and sends each process a piece of the array according to the rank id. Gather (MPI_Gather) is the opposite to scatter. It receives data stored in small arrays from all the processes and concatenates it into a large array in rank order.

We introduce the details of broadcast as an example. In the broadcast, one process sends the data to all the other processes. The communication pattern of broadcast is shown in Figure 1.1(a). In

our example, process 0 is the root process, which has the data. The other processes will receive the copy of data. In MPI Broadcast, binomial tree, which is shown in Figure 1.1(b) is one of the basic communication tree structures. In this example, process 0 (root) sends the copy of data to process 1 in the first step. After this step, both process 0 and process 1 have the copy of data. In the second step, process 0 sends the data to process 2 while process 1 sends the data to process 3. Finally, the first four processes send the data to the rest four processes and the broadcast has been completed.

We utilize the same pattern in Figure 1.1(a) to demonstrate MPI_Scatter. Instead of sending all the data to all the other processes, MPI_Scatter first divide the data into eight parts. The root process keeps the first part of the data and then sends the i -th part to process $i-1$, where $i = 2, 3, \dots, 8$.

1.2 Motivation

Although the IaaS clouds have brought great opportunities for the HPC applications, there are many research problems which need to be carefully considered. Performance and monetary cost optimizations are two important problems. Cloud providers often provide different instance types with different computational capabilities and prices, which gives us opportunity to improve the performance and reduce monetary cost for HPC applications. However, the performance and monetary cost optimizations are complex problems on IaaS cloud due to several reasons.

Diverse cloud offerings. First, there are multiple clouds adopting different offerings for the users. For example, Google Compute Engine charges by minute while Amazon EC2 charges by hours. Second, IaaS clouds provide many different kinds of resources to the users. For example, Amazon EC2 have on demand, spot and reserved instances. Furthermore, for each kind of instances, there are more than twenty different types, which have diversified capabilities and prices, can be selected by the users.

Network heterogeneity. The cloud environment has different hardware and software designs, in contrast with traditional clusters. The commodity switch-based tree structure is widely used for

Figure 1.2: A tree-topology network

the current cloud practice to interconnect the servers [16], as illustrated on Figure 1.2. Machines are grouped into *racks* and racks are connected with switches in the higher-level. The problem for such topology is that network bandwidth for any machine pairs is not uniform, which depends on how the switch linking the machines. For example, the cross-rack bandwidth is much lower than the intra-rack bandwidth. The heterogeneous network structure makes the performance optimization complex.

Network interference. Many performance optimizations directly utilize several measurements from network performance. Some recent studies [17, 18] make optimization decisions based on only a few *ad-hoc* measurements on the end-to-end performance. However, as IaaS cloud environment is designed as a shared infrastructure, the virtual machine consolidation [19], dynamic network flow scheduling [20] and some other users' application execution lead to the performance dynamics. In this case, such direct use of measurements is inherently affected by dynamic network and is inaccurate to reflect the performance. Due to the network interference and significant performance variance in IaaS clouds, the design of performance optimizations are even more complex.

Cloud prices dynamics. The cloud, which is an economic market, has dynamic prices. For example, Amazon EC2 provides spot instances and their prices are decided by market. Many previous monetary cost optimization approaches for HPC applications utilize static cost. They are not suitable

in the dynamic cloud environment. In such environment, how to set bid price and how to improve the reliability of executing HPC applications need to be carefully considered.

1.3 Contributions

In this thesis, we have explored three projects for performance and monetary cost optimizations. The contributions of this thesis are summarized as follows:

Performance Optimization. For performance optimizations, we first solve the network performance hierarchy problem and develop novel algorithms for collective communication operations including broadcast, reduce, gather and scatter. 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.

Furthermore, we solve the network interference problem. In order to utilize network performance aware optimizations on IaaS clouds, we try to decouple the constant component from the dynamic network performance while minimizing the difference between the network performance and the constant component. We propose a novel RPCA-based approach to improve the effectiveness of network performance aware optimizations. The evaluation results demonstrate that our approach can largely improve the performance, reducing the average elapsed time of broadcast, scatter of MPI and topology mapping by 20-40% and 8-20% over the baseline approach and the approach based on direct use of the network measurements.

Monetary Cost Optimization. For monetary cost optimizations, we mathematically formulate the monetary cost optimization problem with performance guarantees for MPI applications in the Amazon EC2 spot market. We can find the near optimal solutions for the problem with our proposed numerical methods. According to problem-specific features, we further utilize several dimensionality reduction techniques to reduce the optimization overhead while preserving the solution quality. Finally, we implement our optimization method on Amazon EC2 based on OpenMPI [21] using the

BLCR library [22]. We evaluate our method on a number of applications, including LU, BT, SP, FT, IS, BTIO in NPB [23] benchmarks and a real-world application, LAMMPS [24]. Evaluation results indicate that our approach can reduce 20% of total monetary cost on average (up to 40%) compared with the state-of-the-art algorithm [1].

1.4 Structure of the Thesis

The report is organized into five chapters.

Chapter 1 introduces the background, motivations and contributions of our study.

Chapter 2 introduces the cloud dynamic study and reviews the related work on performance and monetary cost optimizations and relevant techniques, including RPCA and fault-tolerance in MPI.

Chapter 3 presents our study for the network performance aware MPI collective communication operations in the cloud. We design two kinds of algorithms for MPI collective communication operations and obtain good improvement in efficiency.

Chapter 4 presents a novel network performance model to decouple the constant and volatility components from the dynamic network performance of virtual clusters.

Chapter 5 presents the monetary cost optimization project. We first formulate an optimization problem and then build a novel cost model to minimize expected monetary cost.

Chapter 6 concludes our current studies and discusses the future work.

Chapter 2

Background 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 Dynamic Study

2.1.1 Performance Dynamics Study

We have performed a detailed study on the long-term network performance of a virtual machine pair in Amazon EC2.

We first perform end-to-end measurements on Amazon EC2. Figure 2.1 shows the pair-wise network bandwidth for 196 medium instances on Amazon EC2. The network bandwidth changes a lot among machine pairs. The mean and standard deviation are 115.0 (MB/s) and 40.2 (MB/s). We found that some pair-wise bandwidths are more than 500 MB/s. The reason is that some virtual instances are allocated to the same physical node. Previous studies have observed similar results of significant network performance variations among machine pairs on other cloud providers like Azure and Rackspace [25].

Then we have made interesting observations on the long-term network performance of a virtual machine pair. First, while the network performance from consecutive measurements forms a clear band, it is almost unpredictable at a certain time. This is the combined effect from the constant and

Figure 2.1: CDF of pair-wise network bandwidth for 196 medium instances in Amazon EC2

volatile components of the network performance. The one-week consecutive measurements shown in Figures 2.2(a) and 2.2(b) conform to this observation. Second, there can be significant network performance changes, due to the start/termination of other network intensive applications or virtual machine migrations. The gap shown in Figure 2.3 demonstrates this phenomenon. This is a long-term change in the constant, and our approach should be designed to handle this change.

2.1.2 Price Dynamics and Spot Pricing Model

Amazon EC2 offers different types of virtual instances with different computational capabilities and prices. There are multiple pricing models in the cloud [26], such as on-demand, spot and reservation. We focus on the on-demand and spot pricing models in our work.

Different from on-demand pricing model where users pay a fixed price for unit time of instance usage, the spot price changes along time. To use spot instances, users are required to set the appropriate bid price they want to pay. The bid price is fixed once the instance is launched. If bid price is higher than the spot price, the instance can be successfully launched and run; otherwise it waits. Amazon updates the spot price periodically and launches the waiting instances whose bid prices exceed

(a) Latency

(b) Bandwidth

Figure 2.2: Temporal behavior of the point-to-point performance between two medium instances: latency (left) and bandwidth (right).

(a) Latency

(b) Bandwidth

Figure 2.3: The gap of the point-to-point communication between two medium instances: latency (left) and bandwidth (right).

(a) us-east-1a zone

(b) us-east-1b zone

Figure 2.4: Spot price history of Amazon EC2 over three days

Figure 2.5: Histogram of m1.medium spot price in us-east-1a zone in four consecutive days

the current spot price and terminate the instances whose bid prices are lower than that. We have statistically analyzed the spot price history and found that, 1) the spot price varies in both temporal and spatial dimensions and it is hard to predict the exact price in the future; 2) however, the probabilistic distribution of the spot price is stable in a short time.

Spot price variance. The spot price has shown variances in both spatial and temporal dimensions. We study the spot price history of m1.large and m1.medium instance types in two availability zones from Amazon EC2. Figure 2.4 illustrates the variation of the spot prices over three days.

Temporal variation. The spot price is not static, but changes along the time. The change in the spot price can be huge. Figure 2.4(a) shows that the spot price of m1.medium instances in the zone of us-east-1a increases from less than \$0.1 to around \$10 at the time of 10 hours. More importantly, the variation of the spot prices is not constant. The spot price can be unchanged for some time (e.g., spot price of m1.medium in the zone of us-east-1a during 20 to 40 hours, highlighted with A in Figure 2.4(a)) and changing dramatically for some other time (e.g., spot price of m1.medium in the zone of us-east-1a during 50 to 60 hours, highlighted with B in Figure 2.4(a)). Thus, it is generally difficult to predict the spot price exactly, even in the very near future.

Spatial variation. On the spatial dimension, we have the following observations. First, the spot price variations of different instance types are different. For example, as shown in Figure 2.4(a), the spot price of m1.medium changes abruptly during 50 to 60 hours while the price of m1.large is unchanged. We also find that, the spot price of a more powerful instance can be cheaper than a less powerful instance type at some time (e.g., m1.large and m1.medium in Figure 2.4(a)). Second, the spot price variations of the same instance type in different availability zones are different. For example, the spot price of m1.medium instances changes largely in the us-east-1a zone and is quite low and unchanged in the us-east-1b zone at all time. It is possible to use spot instances of different types or in different availability zones to replicate the MPI executions.

Stable spot price distribution. Although the spot price is variant, the probabilistic distribution of the spot price can be **stable** in a short time. Figure 2.5 shows the spot price histogram of m1.medium

instances in the zone of us-east-1a in four consecutive days. The spot price distributions in the four days are very close to each other. It is feasible and desirable to utilize the history of spot price to estimate probabilistic distribution of the spot price in a short time.

2.2 Performance and Monetary Cost Optimization

2.2.1 Network Performance Aware Optimization

Network performance aware optimizations have long been a hot research topic to optimize distributed applications on traditional network environments (e.g., local clusters and grids [27, 28, 29, 30, 31, 32, 33]). The essential idea of those optimizations is to obtain the network performance metrics (mainly latency and bandwidth of all communication links in the cluster), and then to optimize overall performance by carefully selecting the links for communications. Many studies rely on the network topology, and some of them assume the *a-priori* knowledge of all-link (or pair-wise) network performance in the (virtual) machines. However, without understanding and capturing the long-term network performance, the existing studies in the cloud [17, 18] simply adopt the methods in cluster/grid environments according to several *ad-hoc* measurements/calibrations.

We use two basic applications – MPI collective operations [28, 29, 30] and topology mapping [31] as examples to demonstrate network performance aware optimizations with the knowledge of pair-wise network performance. We select the two applications for their sensitive to the communication link selection.

MPI collective operations. Network performance optimizations are very important for the performance of collective communications of MPI [28, 29, 30]. Our study focuses on the four basic collective operations, namely broadcast, reduce, gather and scatter, which have been introduced in Chapter 1. Binomial tree is one of the basic communication tree structures for collective operations.

Due to the unevenness of pair-wise network bandwidth among the virtual machines [34, 25], we need to carefully choose the link in constructing the communication tree in MPI collective operations.

It causes significant performance loss if the links are wrongly chosen.

Given all pair-wise network performance, the traditional approach can build an efficient communication tree without process migration so that the communication links can better utilize the network bandwidth. In this case, we assume that each process has been successfully allocated to the right machine. Given the all-link network performance for a set of machines, Banikazemi et al. developed a near-optimal greedy algorithm named Fastest-Node First (FNF) to construct a binomial tree [31]. The basic idea is described as below. We assume that each machine has only one process, the root of the communication tree is fixed and the extension to multiple processes per machine is straightforward. The algorithm works in multiple iterations. In each iteration, it maintains two sets of machines, S and U , to represent the machines that have been selected in the tree structure and have not been selected, respectively. Initially, S consists of the root for the binomial tree and U consists of all the machines under study except the root. In each iteration, for each machine s in S , we pick a machine in U having the best network performance with m , according to the pair-wise network performance. Then, the resultant machine (denoted as r) is removed from U immediately and is added to S after this iteration. That creates an edge in the tree from s to r , where s and r represent sender and receiver, respectively. When considering a machine in S , the selection order is according to the order added into S . The algorithm stops when U becomes empty.

A running example is given in Figure 2.6(a). On the left, it is the weight matrix for pair-wise network performance. A smaller weight indicates that the network performance is better. We assume Machine 1 is the root. In first iteration, Machine 1 is the sender and Machine 3 is chosen as the receiver for its smallest weight. Now, S consists of machines 1 and 3. In the next iteration, we choose Machine 2 to receive message from Machine 1 and Machine 6 to receive message from Machine 3. On the right of Figure 2.6(a) shows the resultant tree structure by the FNF algorithm. The total weight of the longest path is five.

Topology Mapping. Assigning a set of tasks to machines such that the task communication efficiently utilizes the physical links in the network is called topology mapping [35]. In our work,

(a) A running example of FNF

(b) The revised example of FNF

Figure 2.6: An example of weight matrix as all-link network performance and FNF tree structure

we utilize the Greedy Heuristic Algorithm [35]. Basically, the task with the largest data volume to transfer is mapped to the machines with the highest total bandwidth of all its associated links. Therefore, all-link network performance is important for guiding this mapping. There are some other topology mapping techniques [36, 37], which aim at optimizing for latency communication time. However, we only utilize the most commonly used approach for bandwidth sensitive operations as an example. Our approach can be extended to adopt on such kinds of techniques easily. In fact, our approach is proposed to efficiently and effectively utilize performance aware optimizations, which is independent to either bandwidth-optimized or latency-optimized algorithms.

The algorithm is described as follows. Again, we assume that each machine has only one process, and the extension to multiple processes per machine is straightforward. The algorithm assumes two inputs: (1) a task graph G : a vertex representing a task and the edge represents data transfer between two tasks (its weight represents the data volume for the communication); (2) a machine graph H : a vertex representing a machine and the edge representing the connectivity between two machines (its weight represents network bandwidth between the two machines).

Given the two directed graphs G and H , the algorithm determines the mapping between G and H . Now let the weight of vertex v in the graph (either G or H) be the sum of weights for all edges

associated with v . The algorithm starts at the heaviest vertex v_0 in H , chooses the heaviest vertex s_0 in G and maps v_0 to s_0 . Next, the algorithm maps v 's heaviest neighboring vertices in H to the neighboring vertices in G with the heaviest connections. The mapping process finishes when all the vertices in H have their mappings to distinct vertices in G .

2.2.2 Monetary Cost Optimizations

Cost-aware optimizations have generated fruitful research results for decades in various areas such as grid computing [38, 39], databases [40] and Internet [41]. The pay-as-you-go scheme attracts many interests in optimizing the monetary cost for various applications in the cloud.

The importance of spot instances on cost optimizations has attracted many research interests on studying the spot price and utilizing it for cost efficiency. Song et al. [42] proposed a bidding strategy to maximize the revenue of cloud brokers on utilizing spot instances for computation. Mazzucco et al. [43] also designed a bidding policy and resource allocation policies to maximize the revenue of SaaS providers. But they only consider optimizing the monetary cost but ignore the reliability. Guo et al. [44] discussed bidding policy of spot instance for highly available services. But they does not focus on MPI-based applications. Taifi et al. [45] proposed a formal model to guide the design of checkpoint for MPI-based applications. But they only considered checkpointing technique. Marathe et al. [1] exploit replicated executions for HPC applications on Amazon EC2.

2.3 Relevant Techniques

2.3.1 Robust Principal Component Analysis

PCA is one of the most widely used approaches for dimensionality reduction. However, the accuracy of PCA is prone to noise or gross errors in the input data. Robust Principal Component Analysis (RPCA) [46] was proposed to improve the robustness of PCA under noisy or error measurements. The basic idea is to recover a low-rank matrix from a series of corrupted measurements and to minimize

the noise component that is sparse and unknown. Suppose A is a data matrix, D is a low-rank matrix and E represents a sparse matrix. RPCA is proposed to solve the following problem, where $\|E\|_0$ is the zero norm of E .

$$\begin{aligned} & \text{minimize} && \text{rank}(D) + \lambda \|E\|_0 \\ & \text{subject to} && A = D + E \end{aligned}$$

RPCA has been widely used in computer vision. It can be used to solve many important applications like face recognition, latent semantic indexing and video surveillance [46]. In a video surveillance application, we need to distinguish the activities which occur in the background. Particularly, we consider each video frame as a row of the data matrix A . RPCA is used to divide A into two components: D representing the information of the background and E including the moving objects which may appear accidentally in each video frame. D is a low-rank matrix because the background is stable across the video frames of a reasonably long period. In the scenario of network performance in the IaaS cloud, we expect to find the constant component that can represent the long-term performance in the dynamic network environment. Our problem has the analogy to RPCA in computer vision.

There are many algorithms which are proposed to solve this optimization problem (e.g., [47, 48]). We choose the approach by Ji et al. [48] (their implementation [49]), which is a polynomial-time algorithm and has strong performance guarantees on the error.

2.3.2 Fault Tolerance in MPI

There are two types of commonly adopted techniques for fault tolerance in MPI, namely checkpointing and replicated execution.

Checkpointing, a failure recovery approach, has been widely used. It allows the applications to restart from where failure occurs instead of restarting from the beginning. There are two kinds

of checkpoint restart techniques for MPI applications, including coordinated checkpointing and uncoordinated checkpointing. The coordinated checkpointing requires global coordination where all processes are synchronized and save states consistently for future recovery. Moussa Taif et al. [45] discuss how to improve the performance of MPI on spot instance by coordinated checkpoint. The uncoordinated checkpointing requires less coordination and each process takes its checkpoints independently. This approach induces less overhead than coordinated checkpointing, but some techniques, for example message logging, are needed to avoid the domino effect. Logging all the messages takes up a huge storage volume. Guermouche et al. [50] proposed an uncoordinated checkpointing protocol for deterministic MPI-based applications. A survey of rollback-recovery techniques is shown in [51]. In our algorithm, we select coordinated checkpointing technique. When we run an MPI application on the same spot instance type, the processes are terminated at the same time and coordinated checkpointing technique is more efficient.

Primary-backup and active replication [52] are two major classes of replicated execution techniques. Primary-backup replication utilizes one replica as the primary, which receives invocations from client processes and returns responses, and uses the other replicas as backups. The backups, which interact with the primary, do not interact directly with the client process. Active replication, which is also known as the state-machine approach, gives all replicas the same role without the centralized control of the primary-backup technique. The replication technique is adopted by many existing studies. For example, MPI/FT [53], P2P-MPI [54] and VolpexMPI [55] have provided transparent fault tolerance by replicating MPI processes. In our model, the client processes do not communicate with the replicas, and the active replication is more suitable.

Chapter 3

Network Performance Aware MPI Collective Communication Operations in the Cloud

In this chapter, we present our work of the optimization algorithms on Network Performance Aware MPI Collective Communication Operations in the Cloud.

3.1 Problem Overview

While topology aware algorithms work well in traditional clusters, they are no longer feasible in the cloud environment. Due to the system management and virtualization issues, the topology information is not available or not effective to use for optimization. First, the network topology is hidden by virtualization from users. Virtualization offers a uniform interface to users, without exposing the real configurations of the physical hardware. Second, cloud does not provide administrator privileges, which are usually required in getting network topology information [29, 30], on accessing hardware under virtualization layer. Due to the first and second issues, some reverse engineering techniques in topology discovery [56, 57] are not applicable in the cloud environment.

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 cost. Furthermore, 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.

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* is calculated by the latency and bandwidth matrices. Based on the network performance matrices, we propose *network performance hierarchy* to capture the network performance among a set of virtual instances 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 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.

3.2 Algorithm Design

Due to unevenness and asymmetry of pair-wise network performance in the cloud, there is no shortcut 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.

3.2.1 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. 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, $L[0, 1, \dots, N-1][0, 1, \dots, N-1]$. $L_{i,j}$ represents the average latency from machine i to machine j . $L_{i,i}$ is initialized with zero.
- Bandwidth matrix, $B[0, 1, \dots, N-1][0, 1, \dots, N-1]$. $B_{i,j}$ 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 $M[0, 1, \dots, N-1][0, 1, \dots, N-1]$, 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 [27].

We perform discretization on those matrices. We refer M to be the ones after discretization. This study focuses on handling fix-sized messages, and it is easy to extend our model to variable-sized messages. We use the message size m for calculating M .

3.2.2 Network Performance Hierarchy

Due to the asymmetric network performance matrix, we design a data structure to guide our optimizations of carefully reordering 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.

Closeness definition. We use a boundary value to quantify how “close” on the network performance between a machine and a group of machines. For a group of machines, $\tilde{O}$, and a boundary value, b , we define a machine p is b -close to $\tilde{O}$, if and only if $\forall p' \in \tilde{O}, \max(M_{p',p}, M_{p,p'}) \leq b$. If a process p is b -close to $\tilde{O}$, we can add p to $\tilde{O}$ and form a new group, subject to the boundary value b .

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 $\mathbb{G}_0, \mathbb{G}_1, \dots, \mathbb{G}_{iter}$;
-

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. Due to the exponential complexity, it is commonly infeasible to calculate the optimal solution. Therefore, we choose a greedy method to construct the hierarchy. The grouping process generates a network performance hierarchy, as illustrated in Algorithm 1.

The essential idea is to form some 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 as the difference of 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  $\mathbb{G}'$  such that  $|\mathbb{G}'| = 0$ ;
3: while  $|\mathbb{G}'| > \frac{|\mathbb{G}|}{2}$  or  $|\mathbb{G}'| = 0$  do
4:   Clear  $\mathbb{G}'$ ;
5:   for any machine  $p \in \mathbb{G}$ ,  $\forall \tilde{\mathcal{O}} \in \mathbb{G}$  do
6:     Add  $p$  to an existing group  $\tilde{\mathcal{O}}' \in \mathbb{G}'$  if  $p$  is  $b$ -close to  $\tilde{\mathcal{O}}'$ ;
7:     if such a group does not exist then
8:       Create a new group for  $p$ , and add it to  $\mathbb{G}'$ ;
9:   Increase  $b$  by 1;
10: while the span of  $\mathbb{G}'$  is larger than  $T_{span}$  do
11:   Adjust  $\mathbb{G}'$ 
12: return  $\mathbb{G}'$ ;

```

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 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 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 $\sum_{i=0}^{\log_2 N} \left(\frac{N}{2^i}\right)^2 \sim$

$O(N^2)$. 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 calculate 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 $\mathbb{G}_0, \mathbb{G}_1, \dots, \mathbb{G}_l$ (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;
-

3.2.3 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 essential 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 3.1 illustrates a two-level reduce operation, where multiple sub-communicators are created according to the network performance hierarchy.

Figure 3.1: Network performance aware reduce operation

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 $\mathcal{G}$, we select p ($p \in \mathcal{G}$) as the leader such that $\sum_{p' \in \mathcal{G}} M_{p',p}$ 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.

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 reorder the processes in the group such that the root process has the rank of zero.

Gather and Scatter. As gather and scatter are symmetric operations, we design a similar network performance aware algorithm for scatter, and we focus on the description of gather.

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 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: $Construct(d, R_h, dist + 1)$;
-

Algorithm 4 illustrates our network performance aware algorithm for gather. The essential idea is 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 3.2 (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 [29, 30].

Figure 3.2: The construction of the communication tree in gather: (a) construction by distance; (b) network performance hierarchy (top) and the result communication tree (bottom).

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 3.2 (b) shows 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 hier-

archy has four levels. Process 0 is root process for 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.

3.3 Evaluations

3.3.1 Experimental Setup

We utilize 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.

On both real cloud environments and simulation, we compare our network performance aware algorithms with MPICH2 [58]. MPICH2 in most cases chooses the binomial tree algorithm. Note, it may also choose different algorithms according to the different message size. For instance, MPICH2 chooses the ring algorithm for broadcast when the message size is larger than 1MB. The major metric in our study is the completion time of the collective operation. We use an existing MPI benchmark called SKaMPI [59] to measure the completion time.

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 study the impact of message sizes, number of virtual machines and 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.

Figure 3.3: The network performance hierarchy for 64 medium instances

On the same virtual cluster, we obtain the network performance matrix when the experiment starts, and use it afterwards unless the network performance has changed significantly. 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 focus on the results for broadcast and scatter in this section. We observe similar results on the dual operations.

3.3.2 Results on Collective operations

Network performance hierarchy. Figure 3.3 shows a snapshot of the network performance hierarchy for 64 medium instances under the default setting. The number of levels in the hierarchy is four. The number in the node indicates the number of virtual machines in the group. 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 3.4 shows the performance comparison of the four operations with different message sizes. On the four collective operations, our

Figure 3.4: Performance improvement for 64 medium instances

network performance aware algorithms significantly outperform their counterparts in MPICH2. It indicates the importance of network performance awareness in cloud. 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 selects ring algorithm, instead of binomial tree algorithm. Ring algorithm turns out to be even slower than our algorithm.

Figure 3.5 shows the cumulative distribution function (CDF) 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 3.6 shows CDF for 100 measurements

Figure 3.5: CDF of individual process elapsed time for scatter

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 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 3.7. We also show the performance of our approach and MPICH2. Our proposed approach is close to the best of Monte Carlo simulations.

3.3.3 Results on Long-term Behavior

Because of the cloud network performance dynamics, 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.

(a) Broadcast

(b) Scatter

Figure 3.6: The distribution of broadcast and scatter performance with different virtual clusters.

(a) Bcast(Message Size=1MB)

(b) Scatters(Message Size=1MB)

Figure 3.7: Probability density distribution with Monte Carlo simulations (broadcast and scatter)

Figure 3.8: Long-term performance of Bcast and Scatter

Figure 3.8 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 reconstructing 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.

3.3.4 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

(a) CG

(b) N-body(Message Size=1MB)

(c) N-body(#Step=128)

Figure 3.9: Performance comparison of applications on 64 medium instances: N-Body and CG.

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 [58]. 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 [58]) are considered as our future work. 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. 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 3.9(a) shows the comparison studies for CG. We vary the size of vector 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 3.9(b) and 3.9(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%.

Chapter 4

Finding Constant From Change: Revisiting Network Performance Aware Optimizations on IaaS Clouds

4.1 Problem Overview

We have studied the network performance aware optimizations in the previous work. Those studies [17, 18] make optimization decisions based on only a few *ad-hoc* measurements on the end-to-end performance. However, such direct use of measurements is inherently affected by dynamic network and is inaccurate to reflect the long-term performance due to the cloud system dynamics such as virtual machine consolidation [19] and dynamic network flow scheduling [20].

To utilize existing network performance aware optimizations, we attempt to decouple the constant component from the dynamic network performance on IaaS clouds. We further minimize the difference between the network performance and the constant component. The difference can also be considered as *error*, as we utilize the constant component to guide the design of network performance aware optimizations.

Interestingly, this problem can be cast into a common problem in the computer vision, named Robust Principal Component Analysis (RPCA) [46]. RPCA is proposed to solve the following problem: for a data matrix, it is used to identify a low-rank matrix and a sparse matrix with minimized

norm, subject to that the sum of the two matrices are equal to the data matrix. Specifically, we develop a novel approach based on RPCA with special design and optimizations for practical use on IaaS clouds, and leverage the theoretical properties of RPCA to find the constant component from the dynamic network performance. We model each row of the data matrix to be one snap-shot of all-link performance for the cloud at a certain point of time, and apply RPCA on that data matrix to obtain the constant component and error as the low-rank and sparse components, respectively.

This seemingly simple design of decoupling the constant component from network performance enables existing or new network performance aware optimizations in virtual clusters. Based on the constant component, conventional network performance optimizations become valid, i.e., we can select the best performing links with the minimized errors. On the other hand, with the error component, we are able to determine the effectiveness of network performance aware optimizations in virtual clusters, e.g., if the error is too large, the network of the IaaS cloud is too dynamic and network performance aware optimizations are useless.

4.2 Problem Definition

We consider the scenario where users apply network performance aware optimizations to their applications running in the virtual cluster, built on the IaaS cloud. The unique cloud network performance features motivate us to understand the cloud network performance of virtual clusters and to investigate how to improve the application performance. This section gives the definitions on our problem.

Network performance. We can measure the performance of each link, and the all-link network performance for a set of N virtual machines (or instances). We denote this set of virtual machines to be a *virtual cluster*.

We adopt the α - β model [27] to model the network performance for each link. In the α - β model, each link is represented with two parameters: the latency (α) and the bandwidth (β) between the two

machines. The transfer time for sending the data of n bytes is estimated to be $\alpha + \frac{n}{\beta}$. This model can be utilized to calculate the transfer time for message with arbitrary sizes.

We define the pair-wise network performance with a matrix (namely *performance matrix*). Since the network performance varies, we extend the performance matrix with the time dimension. In particular, given two virtual machines i and j , the network performance parameters of the link from i to j at time t_0 is denoted as $\alpha_{ij}(t_0)$ and $\beta_{ij}(t_0)$. We represent the network performance of the virtual cluster with two $N \times N$ performance matrices: $L(t_0) = (\alpha_{ij}(t_0))$ and $B(t_0) = (\beta_{ij}(t_0))$ ($1 \leq i, j \leq N$). In those matrices, we require all the pair-wise performance information between any pair of instances in the virtual cluster.

We have two major motivations for developing the performance matrices. Firstly, the performance matrices offer a general performance model which can be used for network performance aware optimizations of different applications. Secondly, individual link performance is significant to the network performance aware optimizations, because they rely on the comparison of the long-term network performance of all links. For example, in Figure 2.6(a), if we change the weight of the link (Machine 1, Machine 3) to be four, the tree structure will be changed. The new structure is shown in Figure 2.6(b) and the total weight of the longest path reaches seven (instead of five in the original case). This demonstrates the importance of individual link performance.

Temporal performance matrix. Performance matrix can only reflect a snapshot of network performance at a certain point of time. It does not necessarily represent the long-term network performance. First, while the network performance from consecutive measurements forms a clear band, it is almost unpredictable at a single point. This is the combined effect from the constant and volatility components of the network performance. Second, we did observe significant network performance changes on Amazon EC2. Thus, our approach should be designed to handle this change.

To capture long-term network performance, we can perform a series of measurements (i.e., performance matrices). Each measurement on all-link network performance in the virtual cluster is one row. We arrange the n rows according to their measurement time and denote the matrix as N_A .

Formally, we define temporal performance matrix (*TP-matrix*) on $[T_0, T_1]$ ($T_0 < T_1$) as follows, where $T_0 \leq t_i \leq T_1$, $t_i \leq t_{i+1}$, $P_{A_{t_i}}$ is the performance matrix of the virtual cluster at t_i . We layout each $P_{A_{t_i}}$ into a vector of N^2 dimensions by the row order. Thus, the *TP-matrix* is a $n \times N^2$ matrix.

$$N_A[T_0, T_1] = \begin{bmatrix} P_{A_{t_0}} \\ P_{A_{t_1}} \\ \vdots \\ P_{A_{t_{n-1}}} \end{bmatrix}$$

Problem definition. Consider the scenario of applying a network performance optimization to a network communication operation (e.g., a collective operation in an MPI application) in the virtual cluster from T_0 to T_1 . Moreover, we consider an *offline* scenario that we have the a-priori knowledge of network performance at each moment from T_0 to T_1 . If we run the network communication operation at time t_i ($T_0 \leq t_i \leq T_1$), we can effectively apply the network performance optimization according to the pair-wise network performance $P_{A_{t_i}}$. However, this offline approach is impossible at practice, since we do not have the a-priori knowledge on the exact network performance in the future.

Without the a-priori knowledge on the network performance, we want to find a constant component from the time-varying network performance so that the difference between the network performance and constant component is minimized. The basic idea is, although we cannot predict the exact network performance of each link, the constant component represents the long-term performance of each link. According to the link performance in the constant component, we can perform the network performance aware optimizations (e.g., a link tends to have better performance if it has a better performance in the constant component). By minimizing the difference, we can obtain the best total performance of running at all t_i that we can achieve with the constant component.

Similar to the format of temporal performance matrix, we define two matrices:

- Temporal constant matrix (*TC-matrix*), where each row gives the estimated pair-wise network performance in the constant component. All rows are the same, and therefore the rank of the

matrix is one. An example TC-matrix N_D is shown below, where all $P_{D_{t_i}}$ are the same.

$$N_D[T_0, T_1] = \begin{bmatrix} P_{D_{t_0}} \\ P_{D_{t_1}} \\ \vdots \\ P_{D_{t_{n-1}}} \end{bmatrix}$$

- Temporal error matrix (*TE-matrix*), where each row gives the estimated pair-wise network performance error. An example TE-matrix N_E is shown below.

$$N_E[T_0, T_1] = \begin{bmatrix} P_{E_{t_0}} \\ P_{E_{t_1}} \\ \vdots \\ P_{E_{t_{n-1}}} \end{bmatrix}$$

Let us formulate our problem in an offline setting. Given the TP-matrix N_A on $[T_0, T_1]$, we define TC-matrix N_D and TE-matrix N_E accordingly. Our problem is formulated as below.

$$\begin{aligned} & \text{minimize} && \|N_E\|_0 \\ & \text{subject to} && N_A = N_D + N_E, \\ & && \text{and } \text{rank}(N_D) = 1. \end{aligned}$$

We note that, our current problem definition is still impractical, since it requires the knowledge of entire N_A . However, interestingly, this optimization problem can be cast into the RPCA problem. In the next section, we present an RPCA-based mechanism with a few calibrations on the virtual cluster, and use them to calculate the constant component for the network performance during the calibrations. We treat that constant component as lasting for a long period until we observe some significant changes in the network performance. By then, we need to perform re-calibrations and re-run the approach.

4.3 RPCA-based Approach

4.3.1 Approach Design

Recall that we have given an offline definition on finding the optimal constant component in Section 4.2. We develop an adaptive approach to incorporate RPCA to approximate the solution to our problem. We have the following two major considerations in our design.

First, we need to detect the significant changes in the network performance of the virtual cluster in an IaaS cloud. When we say "significant", we mean that the change is so large that it affects the constant component. However, an IaaS cloud user does not have the underlying workloads in the cloud, and thus cannot accurately determine the significant changes in the network performance. Furthermore, one may propose to periodically measure the pair-wise network performance of the virtual cluster. That approach can be prohibitively costly, resulting in high overhead. Thus, we use a simple and lightweight approach on comparing the expected performance of the network communication operation with the real performance. If the real performance is significantly different from the expected performance, we decide that significant changes in the network performance of the virtual cluster occur.

Second, during the period that the network performance does not change significantly (the network performance does change dynamically though), we can safely use one constant component in a part of the period to estimate the constant component of the entire period. Basically, we perform a few measurements on the link-wise network performance on the virtual cluster. Using our RPCA-based approach, we obtain the constant component in an offline manner. We then use that constant component for network performance aware optimizations until significant changes in the network performance of the virtual cluster occur.

Algorithm 5 shows the procedure of our RPCA-based approach on guiding the optimization for a network communication operation on a virtual cluster C . On the cloud, we first calibrate a series

of performance matrices forming a TP-matrix N_A . N_A is the input data matrix A for RPCA. Next, we run the RPCA approach by Ji et al. [48], and get N_D and N_E . We choose Ji et al's approach for its efficiency and effectiveness over other approaches [48]. First, N_D is the same as matrix D in RPCA with rank one. All the rows in N_D represent the same comparison among different pair-wise performance, which are used as inputs to many network performance aware optimizations. Second, N_E is the matrix E in RPCA. It represents the performance error. We can calculate the norm of N_E to determine the effectiveness of optimizations. The process of calculating RPCA is very efficient, and the overhead can be ignored in the experiments.

We describe the details of these two aspects in the following section.

Guiding performance optimization with N_D . With N_D , we can apply traditional network performance aware optimization to the network communication operation, for example, applying the FNF algorithm [31] to construct the binomial tree. Note, the network communication operation can run on a virtual cluster C' , where $C' \subseteq C$, and we can simply use the pair-wise performance belonging to C' . In most of the cases, our model can work well in a long term without any maintenance. But after a long period, we still need to recalibrate the matrix and rebuild the model. During this period, we use the same $P_{D_{t_i}}$ in N_D for many times until there is a significant change in the network performance (e.g., the virtual machine is migrated to another rack). We detect the changes on the network performance and re-calibrate the matrix (Lines 4–9). More details about approach maintenance will be described in the next sub-section.

Figure 4.1 illustrates an example of calculating $N_A = N_D + N_E$ with RPCA. Figure 4.1(a) shows a simplified topology of a virtual cluster of four machines. The number labeled on the edge between two machines represents the network performance of the link. We next perform five calibrations and form a TP-matrix (Figure 4.1(b)). Each row in the TP-matrix represents a performance matrix obtained from one calibration. Then, we run RPCA on the TP-matrix, and obtain a rank-one matrix N_D (Figure 4.1(c)). From N_D , we can obtain a performance matrix in Figure 4.1(d), which can be used for optimizations.

Figure 4.1: An example of calculating $N_A = N_D + N_E$ with RPCA

Algorithm 5 Overview of our RPCA-based approach

- 1: Given a virtual cluster C , calibrate the TP-matrix, and let the matrix be N_A ;
 - 2: Run the RPCA approach by Ji et al. [48], and get N_D and N_E .
 - 3: Given N_D , we apply some network performance aware optimization algorithms to the network communication operation $\mathcal{A}$ running on a virtual cluster C' , where $C' \subseteq C$;
 - 4: Measure the network performance of $\mathcal{A}$ and let it be t ;
 - 5: Let the expected performance of $\mathcal{A}$ be t' ;
 - 6: **if** $\frac{|t-t'|}{t'} \geq threshold$ **then**
 - 7: Go to Line 1; /* update maintenance*/
 - 8: **else**
 - 9: Go to Line 3; /* use the same N_D for later optimizations*/
-

Determining the effectiveness of optimizations. We aim at studying the relationship between performance error and the effectiveness of optimizations. When we view the measurement performance N_A as the optimal optimization solution from off-line, the performance error is the difference between the optimal solution and our performance aware optimizations solution which is based on N_D . Thus, the effectiveness of network performance aware optimizations on IaaS clouds is highly correlated with the performance error, N_E . Thus, we define the relative norm of error matrix N_E , $Norm(N_E) = \frac{\|N_E\|_0}{\|N_A\|_0} (0 \leq Norm(N_E) \leq 1)$ to measure the effectiveness of network performance aware optimizations.

4.3.2 Implementation Details

The previous sections have presented that we can use RPCA to capture the long-term performance of the network in virtual cluster. However, there are some implementation issues that are worth discussions.

Model calibration. We need to obtain the parameter $\alpha_{ij}(t)$ and $\beta_{ij}(t)$ in a given interval $[T_0, T_1]$ and use this data to predict the network performance in the future. In order to calibrate the network performance of an instance pair from Instance i to Instance j , we use the function Ping-pong_Send_Recv in a benchmark called SKaMPI [59] to send and receive messages and to measure the elapsed time. The latency $\alpha_{ij}(t)$ is the elapsed time of sending a one-byte message and the bandwidth $\beta_{ij}(t)$ is calculated from the elapsed time of sending 8MB data. In our experiment, we find that when the size of message is larger than 8 MB, the results are stable.

We have to calibrate each cell of the performance matrix. There have been some network coordinate algorithms (e.g., [60, 61]) to obtain the all-link network performance with a smaller number of cell measurements. Those approaches are not applicable to data center networks, because the triangle condition is not satisfied [20, 62].

The overhead of calibrating the performance one by one pair is too high if N is large. In order to reduce the overhead, at each step we choose $\frac{N}{2}$ instances to send messages and the other $\frac{N}{2}$ to receive.

In this way, we could obtain $N/2$ pairs at a time, and the time consumption is $2 \times N$. While it significantly reduces the calibration overhead, the concurrent message transfers may cause interference with each other. Fortunately, the data center is usually large enough in the scale of tens of thousands of servers, the interference of the virtual cluster (in the scale of hundreds of virtual machines) should be small. We study the impact of different scale in Section 4.4.

The number of rows in the TP-matrix is another key tuning parameter in RPCA. We call this parameter *time step*. If the time step is too large (i.e., the TP-matrix consists of too many rows), the results may be more accurate, but the overhead is too high. In contrast, if the time step is too small, the result obtained from RPCA may not fully reflect the long-term performance. We experimentally evaluate the impact of different time steps in Section 4.4.

Update maintenance. We use the real performance as feedbacks for update maintenance. As Lines 4–9 in Algorithm 5, we first monitor the performance of the network communication operation, and then compare the expected performance estimated from history. To support different data sizes in the performance estimation, we use α - β model [27] to calculate the network performance, with the input of N_D . If we find the difference is more than *threshold*, we could conjecture that the network performance has significant changes. Thus, we need to re-calibrate the TP-matrix and re-run the approach. The parameter *threshold* is a key parameter for update maintenance. We experimentally evaluate its impact in Section 4.4.

4.4 Evaluations

This section presents our experimental results on evaluating the proposed model and applications. Overall, there are four groups of experiments. First, we study the overhead of calibrating the temporal performance matrix (Section 4.4.2). Second, we study the performance impact of our RPCA-based approach in comparison with other baseline and heuristics approach on real cloud environments (Sec-

tion 4.4.4). Our experiments are running on Amazon EC2. Third, we study the impact of our approach in a large-scale cluster with simulations (Section 4.4.5).

4.4.1 Experimental Setup

We use two complementary evaluation approaches including simulations and real experiments. The real experiments were performed on Amazon EC2 in August 2013, with the focus on assessing the practical performance impact of our proposed approach. As for simulations, we use ns-2¹ to simulate a cluster. With the network simulations from ns-2, we are able to define different background traffics, and study the impact of network interference. Moreover, we can simulate different network topologies, and compare the topology aware optimizations with our approach.

Let us present more details about the experimental setup for each approach.

Experiments with Amazon EC2. On Amazon EC2, we consider different scales of virtual cluster in the real cloud environment. In particular, we consider two virtual clusters: one with 64 medium instances and the other with 196 medium instances. By default, we report the results for 196 medium instances.

For each virtual cluster size, the real experiment takes around one week, with one experimental run every 30 minutes. In each run, we run the following experiments one by one: calibration, MPI and topology mapping applications. For each application, we first run the baseline, immediately followed by Heuristics and our RPCA approach. The calibration is to generate the trace for network performance of virtual cluster. We replay the trace to validate our findings on Amazon EC2 for more details. The trace essentially forms a TP-matrix of our experimental period under study. Given the network performance measurements at a point of time, we use α - β model [27] to estimate the performance of one application. We use the trace to study the impact of optimizations and tunings.

Simulations. Recall that many data centers adopt the tree-structured network design [63, 64]. As a start, we utilize the topology with tree structure to interconnect servers. Machines are grouped into

¹<http://www.isi.edu/nsnam/ns/>

racks, and racks are connected with switches in high level. Specifically, we simulate a cluster of 1024 machines. There are totally 32 racks and each rack contains 32 servers. There are two level of switches. In the first level, there are 32 switches (one for each rack). In the second level, only one switch link with the 32 switches in the first level. The bandwidth in the same rack is 1Gb/s and the bandwidth between different racks is 10Gb/s. To simulate the share environment like IaaS clouds, we make some of the machines keep on sending messages to others. We call it background traffic. We first choose the links and then vary two parameters to control the background traffic: message size and the distribution of waiting time between sending the message. For each link, we assume the waiting time satisfies poisson distribution and the expected value is λ . We vary these two parameters to study the impact of errors in our RPCA-based approach (N_E). Applications are run in the simulator, independently with the background traffic. We measure their performance in the similar way as they run on the real cloud environment.

Applications. We apply the proposed RPCA-based approach to two kinds of basic applications – MPI collective operations and topology mapping. Both applications have network performance aware optimizations according to pair-wise network performance in the virtual cluster. In MPI, we study the basic collective operations including broadcast, reduce, scatter and gather. We obtain similar results of reduce and gather as broadcast and scatter, respectively. This is not surprising, because reduce and gather are the dual operations of broadcast and scatter, respectively. Thus, we present the results for broadcast and scatter only. We report the results for the following default setting, unless otherwise specified. The message size for broadcast and scatter is 8MB. The root process is randomly chosen from the virtual cluster. One MPI process runs on each instance. In topology mapping, we create the task graph by randomly generating the weight between 5MB to 10MB. Each task is mapped to one instance.

To further evaluate the impact of our network performance aware algorithms, we have implemented two real-world 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 [65] is a commonly used algorithm for the numerical solution of particular systems of linear equations. The conjugate gradient method 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 all-to-all communication with a gather followed by a broadcast, which is also used in MPICH2 [58]. This simple implementation is sufficient for us to investigate the impact of network performance aware optimizations on real-world distributed applications. During the execution period of both applications, we observed little changes in the network performance, and the temporal performance matrix is calibrated once for one execution of each application.

Comparisons. We assess the impact of the network performance optimizations by comparing the following four approaches.

- **Baseline.** This simulates the scenario of running directly in the cloud environment, essentially without network performance aware optimizations. In MPI, the binomial tree algorithm is used in MPI_Bcast and MPI_Scatter. We use the implementations from MPICH2. In topology mapping, we use the ring mapping algorithm, which maps each vertex in the task graph to a vertex in the machine graph one by one like a ring.
- **Topology.** In the ns-2 simulation, we use the topology information to optimize applications [27, 35, 29, 30]. This approach is denoted as “Topology-aware”. In the experiments on Amazon EC2, we do not include the comparison with this approach, because topology is not available in Amazon EC2.
- **Heuristics.** We capture the TP-matrix and use the average value of each column to optimize the applications. We denote this approach to be “Heuristics”. “Heuristics” represents the direct use of a few measurements of the network performance.

- **RPCA.** We denote the proposed approach to be “RPCA”. The network performance aware optimizations are guided by the long-term part captured by our RPCA approach. The traditional network performance optimizations (FNF for MPI and greedy heuristic algorithm for topology mapping) are used. We set time step =10 and *threshold* = 100% for calibration and update maintenance.

The choice on “Heuristics” is worth further discussions. First, we also use other approaches, for example, minimal value or exponential weighted average. For those approaches, we obtain similar results to the Heuristics approach. Overall, those heuristics based approaches only consider the link separately, whereas RPCA considers the relationship among all the links. Thus, RPCA is able to obtain more information from the same measurement. Second, one may consider performing network performance aware optimizations according to the distribution for each link. However, in order to get the meaningful distribution, excessive measurements are required and the overhead is unacceptably high in practice.

4.4.2 Calibrating Overhead

Figure 4.2 shows the overhead for different numbers of instances for calibrating a single temporal performance matrix when time step is ten. As the number of instances increases, the overhead of calibrating temporal performance matrix is almost linear to the number of instances. When the number is 64, the overhead is less than 4 minutes and when it reaches to 196 instances, the overhead is only about 10 minutes. Given this calibration overhead, our RPCA approach is more useful and acceptable.

4.4.3 Parameter Study

For each experiment, we only vary one parameter while keeping other parameters fixed to their default settings (time step =10 and *threshold* = 100% for calibration and update maintenance).

Figure 4.2: Overhead of calibrating temporal performance matrix

Time Step. We define a function to calculate the accuracy in different time step. With a specified time step, we can calculate the performance matrix P_D as the predicted long-term performance, which essentially is a row of the TC-matrix N_D . On the other hand, we use the whole TP-matrix to obtain the accurate value of the oracle long-term performance P'_D . We define the relative difference of long-term performance to be the accuracy of our prediction i.e., $Norm(P_D) = \frac{\|P_D - P'_D\|_0}{\|P'_D\|_0}$. When the difference is zero, it means the value is 100% accurate.

In our experiment, we find that as the time step increases, the relative difference becomes smaller. A larger time step means larger overhead. Thus, there is a trade-off between the accuracy and overhead. We select the maximum time step when the relative difference is within 10% among different time steps. In this experiment on broadcast, the suitable time step is ten. We obtain the similar results for other applications.

Update maintenance threshold. In this experiment, we find when the percentage is less than 20%, the maintenance is very frequent which leads to large overhead and the average elapsed time will be so huge. In contrast, when the percentage is more than about 150%, which means the current execution time is 1.5 times higher than the previous execution time, there will be no re-measurement

Figure 4.3: Overall performance comparison for the three applications on Amazon EC2 (all normalized to the average of Baseline)

of TP-matrix N_A and no change for TC-matrix N_D . When the percentage is about 100%, it almost achieves the best performance. The overhead is not so large and the performance improvement is comparable to the best.

4.4.4 Results with Amazon EC2

We present the following results related to Amazon EC2. First, we present the overall performance comparison on running real experiments on Amazon EC2. Second, we present some detailed studies with replaying the trace of network performance measurements from Amazon EC2.

4.4.4.1 Results on basic applications

Figure 4.3(a) shows the average performance comparison of the broadcast, scatter and topology mapping on 196 medium instances of Amazon EC2. The performance is normalized to Baseline. We repeat our experiments for more than 100 times and show the average results. Figure 4.3(b) shows the CDF for the execution time of broadcast in this experiment.

Overall, RPCA consistently outperforms other comparison approaches for all applications running on Amazon EC2. By carefully selecting the links with the best performance according to the constant

Figure 4.4: Overall performance comparison with baseline for the different number of instances on Amazon EC2

component, RPCA improves the effectiveness of network performance aware optimizations.

We have two major observations. First, both Heuristics and RPCA significantly outperform Baseline, with the performance improvement 32–40%. It indicates the importance of the network performance awareness in the cloud environment. With the knowledge of long-term performance captured by RPCA, those optimizations can select the suitable links for a better performance. Second, on Amazon EC2, we analyze the trace and find that the network performance error is relatively small ($N_E = 0.1$). Still, RPCA is 8–10% better than Heuristics. We also observe that as N_E becomes higher, RPCA can outperform Heuristics even more.

Figure 4.4 shows the performance improvement of RPCA over Baseline for different numbers of instances in Amazon EC2. The improvement on 196 instances is much higher than that on 64 instances. That is because, when the virtual cluster is large, its virtual machines may be more likely to be located in different racks in Amazon data center. We also observed that the improvement is relatively larger for larger message sizes. This is consistent with our previous observations that the contribution of the update maintenance becomes smaller to the overall execution time. We observe similar results when comparing RPCA with other comparisons.

Figure 4.5: Performance comparison of real-world applications on 196 medium instances: N-Body and CG.

4.4.4.2 Results on real-world applications

We study the breakdown of the execution time in the real-world application. We divide the entire application execution time into two parts: computation and communication. For our proposed algorithms, we also present the initialization cost including calibration and RPCA calculation (denoted as “Other Overheads”). We also note that RPCA calculation contributes to less than 2% of the total overhead.

Figure 4.5(a) shows the comparison studies for CG. We vary the size of vector from 1000 to 1024000. 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 calibrating and calculating overheads. As the vector size increases, more iterations are required for convergence, and the network performance aware optimization reduces the network communication time. The performance gain compensates for the overhead, with 31% and 14% performance improvement over baseline and Heuristics. Figures 4.5(b) and 4.5(c) show the performance comparison for N-body. We first fix the message size as 1M bytes and vary #Step from 10 to 2560. Then we fix #Step to be 2560 and vary the message size from 1K to 1M bytes. As the message size and #Step increase, the computation and

communication play a more important role and the overhead becomes insignificant. Our network performance aware algorithms reduce the network communication time by 36%, and the total execution time by 25% over the baseline approach. The performance improvement over Heuristics approach is around 10%.

4.4.4.3 Detailed Performance Comparison

The network environments of Amazon EC2 is dynamic. For repeatable experiments on studying different settings, we use the method of replaying the trace from the calibrations on network performance of a virtual cluster in Amazon EC2, and estimate the application performance given the pair-wise network performance measurement in the trace.

In the following, we first study the accuracy of our approach by comparing the performance distributions obtained from our trace-replay approach and from measurements. Next, we study the impact of N_E by introducing noises to the network performance and have a detailed study on a case when $Norm(N_E)=0.2$.

Accuracy of performance estimations. We compare the estimated performance distribution and the real measurements. With the performance estimation, our performance estimation from the trace-replay approach is close to the real measurements in the cloud. The average difference is only 18% and 9% for baseline and RPCA, respectively.

Impact of N_E . We study the impact of N_E of the virtual cluster. To study more scenarios for N_E , we randomly assign noises to the trace so that N_E is generated. For each time of adding noise, we change the network performance by 1% (increase or decrease, subject to the predefined N_E and current N_E). Then, we run our RPCA-based approach. If the updated N_E reaches the predefined value, we stop. Otherwise, we repeat the process.

Figure 4.6(a) shows the *expected* performance improvement of our RPCA approach in broadcast, scatter and topology mapping when $Norm(N_E)$ is varied. The $Norm(N_E)$ has significant impacted

Figure 4.6: The impact of N_E in the virtual cluster

the effectiveness of the network optimization on the three applications, compared with Baseline. As $Norm(N_E)$ increases, the performance improvement decreases. When it is less than 0.1, the improvement can reach to more than 40%. But when it is more than 0.2, the improvement is less than 20%. On real environment of Amazon EC2, the network is relatively stable ($Norm(N_E)$ is around 0.1), which is highlighted in Figure 4.6(a).

Figure 4.6(b) shows the performance improvement of comparing RPCA with Heuristics in broadcast when $Norm(N_E)$ is varied. Overall, both approaches can obtain good performance improvement. Our RPCA approach has better efficiency of optimization. When $Norm(N_E)$ is small, which means the network is stable, the network interference plays a less important role. When $Norm(N_E)$ is too large, the network is so dynamic that the network performance aware optimizations have little impact on the performance. When $Norm(N_E)$ is about 0.2, RPCA can obtain about 20% more improvement than Heuristics.

A detailed study. We have a detailed study on a case when $Norm(N_E)=0.2$, which is more dynamic than real Amazon EC2 environments. Figure 4.7(a) shows the average performance comparison, and Figure 4.7(b) shows the CDF for the execution time of broadcast in this experiment. The performance improvement of Heuristics and RPCA over Baseline on Amazon EC2 is consistent with

(a) Overall performance

(b) CDF

Figure 4.7: Overall performance comparison with $Norm(N_E)=0.2$

(a) Different expected values(λ) of Poisson Distribution

(b) Different message sizes

Figure 4.8: The impact of network interference on $Norm(N_E)$

our predictions in Figure 4.6(a) and 4.6(b). When $Norm(N_E)$ is about 0.2, the improvement decreases for both of the two approaches. However, our RPCA approach can obtain better performance improvement than Heuristics. In this case, RPCA outperforms Baseline by 20–28%, and outperforms Heuristics by 12–20%.

Figure 4.9: Performance Comparison in ns-2-based simulations

4.4.5 Simulations on Large-scale Cluster

We report our simulation results on the simulated cluster of 1024 machines. In Figure 4.8(a), we fix the message size in the background as 100MB and vary the expected value λ from 1 to 30 seconds. The expected value λ represents the frequency of the network interference. The figure shows that, as λ increases, $Norm(N_E)$ largely reduces and the network becomes more stable. In Figure 4.8(b), we fix λ as 5s and vary the message size from 10MB to 500 MB. The figure shows that there is an almost linear relationship between the message size in the background and $Norm(N_E)$. From these two experiments, we observe that $Norm(N_E)$ clearly has positive correlations with the background traffic in the simulated cluster. It can explain why we perform an experiment on Amazon EC2 for many times in order to obtain meaningful results. Also, the performance improvement results on Amazon EC2 are similar to our simulations when they have the same $Norm(N_E)$. It represents the interference of network performance satisfies some stable distribution.

Figure 4.9(a) shows the overall performance comparison of broadcast, scatter and topology mapping in the simulated cluster. We use the same execution setting for each application as those in Amazon EC2. Machines are randomly selected from the simulated cluster. We simulate the background traffic so that $Norm(N_E) = 0.1$. We obtain similar comparison results against Base-

line and Heuristics approaches, and focus on the comparison with the topology-aware algorithm. We find that the topology-aware and baseline performs very similar, which indicates that the topology-aware optimization is not effective in such a dynamic environment. RPCA is 25%-40% better than Baseline/Topology-aware approaches and 10%-15% better than Heuristics. Figure 4.9(b) shows the CDF of the elapsed time of broadcast. The results are similar to those on Amazon EC2.

We also vary different N_E to study its impact, and observe similar results as Amazon EC2. For example, as $Norm(N_E)$ increases, the performance improvement of RPCA over other approaches increase and the traditional optimizations relying on the topology and direct use of network measurements are no longer effective in such dynamic environments. When $Norm(N_E)$ is too high (e.g., higher than 0.5), the improvement of network performance aware optimizations becomes marginal. Compared with traditional optimizations, our RPCA-based approach accurately predict this trend with the estimation on $Norm(N_E)$.

Chapter 5

Monetary Cost Optimizations for MPI-Based HPC Applications on Amazon Clouds: Checkpoints and Replicated Execution

5.1 Problem Overview

In cloud computing environments, monetary cost optimizations become a hot topic. Researchers proposed many problems in this field: minimizing the monetary cost in given performance requirements [66, 67], optimizing the performance in the given budgets [68] and scheduling optimizations with both performance and cost constraints [69]. However, most of previous studies [70, 71, 72, 73, 74, 75] mainly focus on job/task scheduling for workflows, rather than MPI applications. On the other hand, previous studies on MPI-based application optimizations [14, 76] have ignored many optimization opportunities provided by the cloud, e.g., considering only on-demand instances for optimizations.

Cloud has evolved into an economic market. Besides on-demand instances, which are charged with a fixed rate, Amazon EC2 provides spot instances, whose prices are mainly decided by the supply and demand in the market. We analyze the spot price history and observe that: a) The prices of spot instances are usually much cheaper than those of on-demand instances but can also be much more expensive in some cases. If spot instances are leveraged properly, they can reduce the monetary

cost [77, 67, 68] compared to solutions with on-demand instances only. b) The spot price is highly dynamic in both spatial and temporal dimensions. In the spatial dimension, different instance types in different Amazon EC2 availability zones can have very different spot prices. In the temporal dimension, even in the same cloud, spot prices can be unchanged for some time and changing dramatically for some other time.

Many previous studies [68, 78, 79, 77, 1, 44, 80] have studied HPC applications on spot instances. Leveraging spot instances appropriately can greatly reduce the monetary cost. However, a spot instance is terminated once the bid price set by the users is lower than the spot price. Such out-of-bid events pose significant challenges to the cost optimizations of MPI applications with performance requirements. In our work, we investigate the effectiveness of two common fault-tolerant mechanisms, including checkpoints and replicated execution, on reducing the monetary cost of MPI applications given user-defined performance requirements. We find that these two mechanisms are actually complementary in the spot market. Checkpoints is useful for reducing the cost of failures while replicated execution helps to reduce the risk of failures. Due to the temporal dynamics, failures can occur in MPI executions in an *undetermined* manner and checkpoints is necessary. Spatial redundancy, e.g., selecting instances from different Amazon EC2 zones for replicated execution, can significantly reduce the failure rate [1]. Combining the two mechanisms together brings great advantages for dealing with the dynamic spot prices. However, combining the mechanisms is very challenging due to the large optimization space.

Combining checkpoints and replicated execution on Amazon EC2 requires making three key decisions: (D1) what instance types to choose; (D2) how much to bid for spot instances; (D3) when to checkpoint. First, Amazon EC2 provides a number of instance types, each with different prices and capabilities. Selecting different instance types for MPI applications greatly affects the performance and monetary cost. Second, setting an appropriate bid price for a spot instance also has great impact on the performance and monetary cost of MPI-based applications. Low bid price leads to low

monetary cost but high probability of instance termination, and hinders the application performance. Third, a good checkpoint frequency also matters. The high checkpoint frequency leads to large overhead while the low frequency results in high recovery time. Making good decisions for all of the three factors is non-trivial and requires searching a large optimization space.

Although previous work [1] has already studied the determination of bid price and checkpoint frequency for replicated executions in multiple Amazon EC2 availability zones, they did not fully consider the above-mentioned optimization factors. In this paper, we aim to minimize the monetary cost for MPI-based applications in Amazon EC2 cloud with performance guarantees. We mathematically formulate this problem in the spot market and propose a numerical method to generate near optimal solutions. We propose three techniques to reduce the optimization overhead. First, we decouple the instance type selection (D1) from the other two decisions (D2 and D3). Assume the solution space for the instance type selection and the other decisions are N and M , respectively. With the divide-and-conquer technique, we can reduce the entire solution space from $N \times M$ to $N + M$. Second, we model the correlation between bid price and checkpoint frequency to further reduce the dimension of our problem. Third, based on the relationship between spot instance failure rate and bid price, we propose a logarithmic search method for bid price of spot instances. Our proposed techniques can greatly reduce optimization overhead, while preserving the optimality of searched results.

5.2 Problem Formulation

In this section, we first give an overview on the problem model and its assumptions. We then mathematically formulate the problem as a constrained optimization problem.

5.2.1 Model Overview and Assumptions

5.2.1.1 Model Overview

Consider an MPI-based application with N processes and the application is associated with a deadline *Deadline*. The number of processes N is fixed during the execution. We consider running the

applications with both on-demand and spot instances to reduce the monetary cost while satisfying the performance requirements.

Circle group. In our model, we define *circle groups* as independent groups of spot instances of the same type and the same availability zone. Each application independently runs on a circle group and multiple circle groups can be used as replications for a single application. In each circle group, we use the checkpointing mechanism for fault tolerance. We set the checkpoints in each circle group independently.

Hybrid execution. If an application is completed in any one of the circle groups, the application is completed and all the other circle groups are terminated. If all the circle groups are terminated by the out-of-bid events before the application is completed, we select the checkpoint which is the nearest to completion and utilize on-demand instances to recover the application. Assume there are K_D types of on-demand instances and we choose the one with the lowest expected monetary cost to recover the application.

5.2.1.2 Assumptions

We have the following assumptions for formulating the problem.

- One MPI process is attached to one core. Based on this assumption, the number of instances in each circle group (denoted as $M_i, i = 1, 2, \dots, K$) can be calculated as follows: assume the instances in circle group j have k cores each, we have $M_j = \frac{N}{k}$.
- When the number of instances is calculated, users can calculate the execution time in different circle groups (denoted as $T_i, i = 1, 2, \dots, K$). There are a number of profiling tools [81] for this purpose. Note that the execution time is the productive time, excluding any checkpointing or recovery overhead.
- The spot prices in different availability zones are all independent. Our observation on history spot price traces confirms this assumption and the previous studies [1] also have similar findings. Based

Name	Description
$Deadline$	Application deadline defined by users
K	Number of circle groups
P_i	Bid price for circle group i
F_i	Checkpoint interval for circle group i
t_i	Failure time for circle group i
M_i	Number of instances for circle group i
T_i	The time to complete MPI execution for circle group i
S_i	Expected spot price for circle group i
O_i	Checkpointing overhead for circle group i
R_i	Recovery overhead for circle group i
K_D	Number of types for on-demand instances
d	The d_{th} type of on-demand instance
M_d	Number of machine for type d of on-demand instance
T_d	The time to complete the MPI execution for type d of on-demand instance
D_d	On-demand price for type d of on-demand instance

Table 5.1: Notations in the model

on this assumption, we calculate the probability of failure for all the circle groups as the product of the failure probability in each circle group.

- The spot price distribution is stable in a short period of time. Based on this assumption, we can design the bid price and checkpoint interval with spot price history.

5.2.2 Problem Definition

We can formulate our problem as a constrained optimization problem. Our goal is to minimize the *expected* monetary cost of the MPI-based application while meeting the performance requirement. Table 5.1 summarizes the key notations in our model. We have three parameters to optimize: the bid price ($\vec{P}$) for each circle group, the checkpoint interval ($\vec{F}$) for each circle group and the type of on-demand instance (d) for the application. $\vec{P}$ is a K dimension vector, where P_i ($i = 1, 2, \dots, K$) represent the bid price for circle group i . $P_i \in [0, H_i]$ ($i = 1, 2, \dots, K$), where H_i denotes the highest spot

price in the history of circle group i . If $P_i = 0$ ($\exists i \in [1, K]$), we do not use circle group i for replicated execution. If $P_i = H_i$ ($\exists i \in [1, K]$), circle group i can be terminated in extremely low probability, which we can ignore. $\vec{F}$ is a K dimension vector as well and F_i ($i = 1, 2, \dots, K$) represent the checkpoint interval for circle group i . $F_i \in (0, T_i]$ ($i = 1, 2, \dots, K$), where T_i is the execution time in circle groups i . If $F_i = T_i$ ($\exists i \in [1, K]$), we do not use checkpoints for this circle group. d denotes the selected on-demand instance type for recovering the application from circle group failures, which is an integer value between 1 and K_D . The problem can be formulated as:

$$\begin{aligned} & \text{minimize} && E(\text{Cost}(\vec{P}, \vec{F}, d)) \\ & \text{subject to} && E(\text{Time}(\vec{P}, \vec{F}, d)) \leq \text{Deadline} \end{aligned} \quad (5.1)$$

5.2.2.1 Estimating Expected Monetary Cost

The expected monetary cost $E(\text{Cost}(\vec{P}, \vec{F}, d))$ is calculated as below:

$$E(\text{Cost}(\vec{P}, \vec{F}, d)) = \sum_{\vec{t}} f(\vec{P}, \vec{t}) \times \text{Cost}(\vec{P}, \vec{F}, \vec{t}, d) \quad (5.2)$$

where the failure rate function $f(\vec{P}, \vec{t})$ denotes the possibility of failures for all circle groups at time $\vec{t}$ when the bid price is set to $\vec{P}$. $\vec{t}$ is also a K dimension vector. t_i ($i = 1, 2, \dots, K$) is the failure time for circle group i . To simplify the problem, we discrete t_i to integers using the floor function. $t_i = k$ means circle group i is terminated between $[k, k + 1]$. When $t_i = T_i$, it means the application is completed on circle group i . The failure rate function is determined by the spot price history. As the failures are independent in different circle groups, we define the failure rate function for each circle group i as $f_i(P_i, t_i)$, and the failure rate function $f(\vec{P}, \vec{t})$ can be represented as:

$$f(\vec{P}, \vec{t}) = \prod_{i=1}^K f_i(P_i, t_i) \quad (5.3)$$

In Formula 5.2, $Cost(\vec{P}, \vec{F}, \vec{t}, d)$ denotes the total monetary cost of all circle groups when circle group i ($i = 1, 2, \dots, K$) is terminated at time t_i ($i = 1, 2, \dots, K$). This cost can be further divided into two parts:

$$Cost(\vec{P}, \vec{F}, \vec{t}, d) = Cost^S(\vec{P}, \vec{F}, \vec{t}) + Cost^D(\vec{F}, \vec{t}, d) \quad (5.4)$$

where $Cost^S(\vec{P}, \vec{F}, \vec{t})$ (Formula 5.5) is the monetary cost for spot instances (or circle groups) and $Cost^D(\vec{F}, \vec{t}, d)$ (Formula 5.6) is the monetary cost for on-demand instances.

To calculate the cost of the spot instances $Cost^S(\vec{P}, \vec{F}, \vec{t})$, we first calculate the expected spot price (denoted as S_i). We find the spot prices lower than the bid price P_i from the spot price history, and use their mean value as the expected spot price S_i for calculation. In Formula 5.5, $t_i + O_i \times \lceil \frac{t_i}{F_i} \rceil$ denotes the total execution time for circle group i . $\lceil \frac{t_i}{F_i} \rceil$ is the number of checkpoint operations and O_i is the average overhead for each checkpoint.

$$Cost^S(\vec{P}, \vec{F}, \vec{t}) = \sum_{i=1}^K S_i \times (t_i + O_i \times \lceil \frac{t_i}{F_i} \rceil) \times M_i \quad (5.5)$$

To calculate the cost of on-demand instances $Cost^D(\vec{F}, \vec{t}, d)$, we first calculate the remaining execution time, in the ratio of the entire execution time (denoted as $Ratio(t_i, F_i)$). When $t_i = T_i$, the remaining execution time is 0 and $Ratio$ is equal to 0. When t_i is less than the checkpoint interval F_i , no checkpoint is taken (the first checkpoint is taken at time F_i) and the remaining execution time is $T_i + F_i$. When t_i is larger than F_i , the execution time becomes $t_i - F_i$ while the remaining execution time is $T_i - (t_i - F_i) + R_i$, where R_i is the recovery overhead. The ratio (as in Formula 5.7) is used to calculate the checkpoint which is the closest (the minimal value) to finish among all circle groups.

$$Cost^D(\vec{F}, \vec{t}, d) = \min_{i \in \{1, \dots, K\}} [Ratio(t_i, F_i)] \times T_d \times D_d \times M_d \quad (5.6)$$

$$Ratio(t_i, F_i) = \begin{cases} \frac{T_i - \max(t_i - F_i, 0) + R_i}{T_i} & \text{if } t_i < T_i \\ 0 & \text{if } t_i = T_i \end{cases} \quad (5.7)$$

5.2.2.2 Estimating Expected Execution Time

The expected execution time can be estimated as follows:

$$E(\text{Time}(\vec{P}, \vec{F}, d)) = \sum_{\vec{i}} f(\vec{P}, \vec{i}) \times \text{Time}(\vec{F}, \vec{i}, d) \quad (5.8)$$

where $\text{Time}(\vec{F}, \vec{i}, d)$ is the total execution time for the application when circle group i ($i = 1, 2, \dots, K$) is terminated at time t_i ($i = 1, 2, \dots, K$). It consists of two parts as shown in Formula 5.9.

$$\text{Time}(\vec{F}, \vec{i}, d) = \text{Time}^S(\vec{F}, \vec{i}) + \text{Time}^D(\vec{F}, \vec{i}, d) \quad (5.9)$$

where $\text{Time}^S(\vec{F}, \vec{i})$ is the total execution time on spot instances and $\text{Time}^D(\vec{F}, \vec{i}, d)$ is the execution time on on-demand instances. We calculate the execution time on spot $\text{Time}^S(\vec{F}, \vec{i})$ as shown in Formula 5.10. It is equal to the longest execution time for all the circle groups. The execution time on on-demand instances is shown in Formula 5.11. We exploit the checkpoint, which is the closest to finish (minimal value for Ratio) among all circle groups, to recover the application.

$$\text{Time}^S(\vec{F}, \vec{i}) = \max_{i \in \{1, \dots, K\}} (t_i + O_i \times \lceil \frac{t_i}{F_i} \rceil) \quad (5.10)$$

$$\text{Time}^D(\vec{F}, \vec{i}, d) = \min_{i \in \{1, \dots, K\}} [\text{Ratio}(t_i, F_i)] \times T_d \quad (5.11)$$

5.3 Model Optimization

We propose a numerical method to obtain the optimal solution for the problem (Formula 5.1). Figure 5.1 gives an overview of the proposed optimization method. The algorithm takes the profiling results of the MPI-based application and spot price history as input. With the application profiles, we first select the on-demand instance type with the minimum expected cost. The selected on-demand instance type is then passed to the two-level optimization algorithm for deciding the optimal bid price and checkpointing interval. The two-level optimization method first reduces the problem dimension by representing the checkpointing interval $\vec{F}$ with the bid price $\vec{P}$ and then utilizes a logarithmic searching method to efficiently search for the optimal bid price.

Figure 5.1: Overview of the optimization algorithm.

5.3.1 On-demand Type Selection

On observing Formula 5.4 and 5.6, we can easily find that the cost of on-demand instances is independent from that of spot instances. Formally, we have that if $\exists \vec{P}'$ and $\vec{F}'$ such that $E(\text{Cost}(\vec{P}', \vec{F}', d_1)) > E(\text{Cost}(\vec{P}', \vec{F}', d_2))$, then for $\forall \vec{P}$ and $\vec{F}$, $E(\text{Cost}(\vec{P}, \vec{F}, d_1)) > E(\text{Cost}(\vec{P}, \vec{F}, d_2))$. In this case, we can separate the decision of on-demand instance type from the other parameters in this problem. In this subsection, we introduce the selection of the most cost-efficient type of on-demand instance.

When optimizing the type of on-demand instance, we have two principles in mind. First, the monetary cost induced by the on-demand instance should be minimized (Formula 5.12). Second, the on-demand instance should be able to meet the deadline requirement (Formula 5.13). Since the price of on-demand instances is fixed, the cost of on-demand instances of type d can be calculated with the unit price of the instances D_d , the execution time on the instances T_d and the number of utilized instances M_d (see Formula 5.12).

Formula 5.12 shows the monetary cost of on-demand instances of type d . The on-demand instance should be able to meet the deadline requirement (Formula 5.13). When evaluating whether the deadline can be met by an on-demand instance type, we should take the checkpointing and recovery overhead into consideration. We define a parameter called *Slack*, where $\text{Slack} = \frac{\text{Deadline} - \text{Deadline}'}{\text{Deadline}}$. $\text{Deadline}'$ is the execution time allowed for on-demand instances. The difference between Deadline and $\text{Deadline}'$ is the time reserved for checkpointing and recovery.

$$\text{minimize} \quad \text{Cost}^{OD}(d) = T_d \times D_d \times M_d \quad (5.12)$$

$$\text{subject to} \quad T_d \leq \text{Deadline} \times (1 - \text{Slack}) \quad (5.13)$$

Solving the above optimization problem is straightforward (given *Slack* is determined in Section 5.4). After finding the optimal solution d^* to the above problem, we can simplify the problem model in Formula 5.1 as below.

$$\begin{aligned} \text{minimize} \quad & E(\text{Cost}(\vec{P}, \vec{F})) = E(\text{Cost}(\vec{P}, \vec{F}, d^*)) \\ \text{subject to} \quad & E(\text{Time}(\vec{P}, \vec{F})) = E(\text{Time}(\vec{P}, \vec{F}, d^*)) \leq \text{Deadline} \end{aligned} \quad (5.14)$$

We denote the execution time on on-demand instances of type d^* as T , the price of d^* instances as D and the number of utilized instances as M . With those notations, Formula 5.6 and 5.11 can be simplified as below.

$$\text{Cost}^D(\vec{F}, \vec{t}) = \min_{i \in \{1, \dots, K\}} [\text{Ratio}(t_i, F_i)] \times T \times D \times M \quad (5.15)$$

$$\text{Time}^D(\vec{F}, \vec{t}) = \min_{i \in \{1, \dots, K\}} [\text{Ratio}(t_i, F_i)] \times T \quad (5.16)$$

5.3.2 Two-Level Optimization

In the subsection, we propose a two-level optimization algorithm to optimize the bid price and checkpoint interval parameters. We first analyze the difficulty of the problem and find that the solution space is too large to find an optimal solution in reasonable time. We utilize two techniques to reduce the optimization space. First, we reduce the problem dimension by modeling the correlation between bid price and checkpoint interval. Second, we develop a logarithmic search method for bid price to further reduce the optimization space.

5.3.2.1 Difficulty Analysis

One straight-forward idea for solving the problem in Formula 5.14 is to utilize convex optimization method [82]. To utilize the convex optimization method, we need to prove that the functions

$E(Cost(\vec{P}, \vec{F}))$ and $E(Time(\vec{P}, \vec{F}))$ are convex on parameter $\vec{P}$ and $\vec{F}$. $E(Cost(\vec{P}, \vec{F}))$ and $E(Time(\vec{P}, \vec{F}))$ are *not* always convex because the following two necessary conditions cannot be met.

First, for parameter $\vec{P}$ (fixed $\vec{F}$), whether $E(Cost(\vec{P}, \vec{F}))$ is convex depends on the function $f(\vec{P}, \vec{t})$ and $S_i(P_i)$ ($i = 1, 2, \dots, K$) (see Formula 5.2, 5.4 and 5.5). Those two functions are extracted from the spot price history and it is not reasonable to have such strong assumptions for the two functions based on our observations.

Second, for parameter $\vec{F}$, $E(Cost(\vec{P}, \vec{F}))$ is not continuous when $F_i \leq t_i$ ($i = 1, 2, \dots, K$ and $t_i = 1, 2, \dots, T_i$), because of the $\lceil \frac{t_i}{F_i} \rceil$ component in the calculation (see Formula 5.2, 5.4 and 5.5). Moreover, Formula 5.7 is not derivable when $F_i = t_i$ ($i = 1, 2, \dots, K$ and $t_i = 1, 2, \dots, T_i$).

As the convex optimization method is not viable, we consider exploiting numerical methods. We first estimate the optimization space that we need to search. Assume the solution space of the checkpoint interval is T , the solution space of the bid price is P and the number of available circle groups is K , then the problem complexity is $O((P \times T)^K)$. The optimization space grows exponentially with the increase of K . Due to the large optimization space, the overhead of finding an optimal solution is high for a practical MPI solution. We utilize two techniques to reduce the searching space in the following subsections.

5.3.2.2 Algorithm Details

We propose a two-level optimization algorithm to reduce the solution space of the optimization problem. In the first level, we perform dimension reduction by finding the correlation of $\vec{P}$ and $\vec{F}$ and modeling $\vec{F}$ with a function of $\vec{P}$. In the second level, we search for the optimal bid price for parameter $\vec{P}$ with a logarithmic search method.

Prior to presenting the algorithm details, we give an example to intuitively show how much we can reduce the optimization space. Assume the number of possible bid prices in the solution space is 100. That is, for each circle group, we consider 100 bid prices equally distributed from 0 to the

highest spot price in the history. We further assume the solution space for the checkpoint interval is 10 and there are 4 types of circle groups for users to choose. According to our analysis in Section 5.3.2, the overall optimization space for the problem is as large as $(100 \times 10)^4 = 10^{12}$. With the proposed dimension reduction technique, we reduce the optimization space to $100^4 = 10^8$. With the proposed logarithmic search method, we further reduce the optimization space to $(\log_2 100)^4 \approx 2000$. Thus, with the two optimizations, our proposal becomes a practical solution for making timely decisions for MPI applications.

Reducing problem dimension. According to the following Theorem 1, we can find the correlation between the bid price parameter $\vec{P}$ and checkpoint interval $\vec{F}$ and model $\vec{F}$ with a vector function of $\vec{P}$, i.e., $\vec{F} = \vec{\phi}(\vec{P})$. Then, we can simplify the optimization problem described by Formula 5.14 to an optimization problem with a single variable $\vec{P}$. The optimal solution $(\vec{P}^*, \vec{\phi}(\vec{P}^*))$ to the simplified problem is also the optimal solution for the original problem in Formula 5.14.

Theorem 1. *If we can find a vector function $\vec{\phi}(\vec{P})$ such that, given $\vec{P}$, we have $\forall \vec{F}, \text{Cost}(\vec{P}, \vec{\phi}(\vec{P})) \leq \text{Cost}(\vec{P}, \vec{F})$. Then the optimal solution $(\vec{P}^*, \vec{F}^*)$ to function $\text{Cost}(\vec{P}, \vec{F})$ satisfies the equation*

$$\text{Cost}(\vec{P}^*, \vec{\phi}(\vec{P}^*)) = \text{Cost}(\vec{P}^*, \vec{F}^*)$$

Proof. As $(\vec{P}^*, \vec{F}^*)$ is the optimal solution, we have

$$\text{Cost}(\vec{P}^*, \vec{F}^*) \leq \text{Cost}(\vec{P}^*, \vec{\phi}(\vec{P}^*))$$

On the other hand, for the given value $\vec{P}^*$, we have

$$\forall \vec{F}, \text{Cost}(\vec{P}^*, \vec{\phi}(\vec{P}^*)) \leq \text{Cost}(\vec{P}^*, \vec{F})$$

Thus,

$$\text{Cost}(\vec{P}^*, \vec{\phi}(\vec{P}^*)) = \text{Cost}(\vec{P}^*, \vec{F}^*) \quad \square$$

We first introduce how to find the vector function $\vec{\phi}$, such that $\vec{F} = \vec{\phi}(\vec{P})$. Note that, we independently perform checkpointing in different circle groups. Since the optimal checkpoint interval is uniquely decided by the bid price in that circle group, we optimize the checkpoint interval for each circle group independently. Formally, we try to find the function ϕ_i for circle group i ($i = 1, 2, \dots, K$), such that $F_i = \phi_i(P_i)$.

Given a bid price P_i' , the monetary cost optimization problem in Formula 5.14 is equivalent to an optimization problem on the execution time in the circle group. Thus, we can formulate the

checkpoint interval optimization problem as the problem to minimize the execution time in a circle group, given a failure rate function $f_i(P'_i, t_i)$. Di et al. [83] have used a mathematical method for this problem. In our work, we adopt their approach as well. We only give a brief introduction here, and more details can be found in their paper [83]. The optimal checkpoint interval can be estimated as:

$$\phi_i(P'_i) = \sqrt{\frac{2O_i \cdot T_i}{E_i(P'_i)}} \quad (5.17)$$

where O_i is the checkpoint overhead in circle group i , T_i is the application execution time and $E_i(P'_i)$ is the expected number of failure events during the execution. Note that we assume no failure occurs on the on-demand instances. $E_i(P'_i)$ is calculated with the given failure rate function as follows.

$$E_i(P'_i) = \sum_{t_i=1}^{T_i} f_i(P'_i, t_i) \quad (5.18)$$

Based on the technique, we reduce the complexity of searching the optimal checkpoint interval from $O(T)$ to $O(1)$.

Optimizing bid price: a logarithmic approach. With Formula 5.17, we now simplify the problem in Formula 5.14 to an optimization problem with a single variable $\vec{P}$ as follow:

$$\begin{aligned} \text{minimize} \quad & E(\text{Cost}(\vec{P}, \vec{\phi}(\vec{F}))) \\ \text{subject to} \quad & E(\text{Time}(\vec{P}, \vec{\phi}(\vec{F}))) \leq \text{Deadline} \end{aligned} \quad (5.19)$$

A naive approach is to search the range from 0 to the highest spot price H in the spot price history trace. That makes an $O(H^K)$ complexity for the searching problem. To reduce the optimization space, we utilize a logarithmic searching method.

Our optimization goal depends on the optimization of two functions, i.e., the failure rate function $f_i(P_i, t_i)$ (see Formula 5.2) and the expected spot price $S_i(P_i)$ (see Formula 5.5) for each circle group i ($i = 1, 2, \dots, K$). Figure 5.2 shows the trends of changing for the two functions with the increase of bid

Figure 5.2: Changing trends of the failure rate function $f_i(P_i, t_i)$ and the expected spot price $S_i(P_i)$ in us-east-1a zone.

price, for spot instances of m1.small and c3.xlarge type in the us-east-1a zone. Both of the functions are sensitive to the bid price, but not in a uniform way. For instance, when bid price of m1.small increases from 0 to 0.2, $f_i(P_i, t_i)$ decreases abruptly by almost 90%. When the bid price increases from 0.6 to 1, $f_i(P_i, t_i)$ varies less than 5%. We have similar observation for other spot instance types. Those observations show that, the bid prices are not of equal importance to the optimization goal and it is inefficient and ineffective to search the entire solution space of bid price with the same granularity.

We propose a new logarithmic searching method to take advantage of the observations. The basic idea is that we do not search the entire solution space with the same granularity. Instead, as the bid price increases, the interval between searched points is increased.

For a given solution space ranging from 0 to H , we divide the space equally into I ($I = 2^L$) intervals. We only search the j^{th} point in the range, where $j = 2^l$ ($l = 1, 2, \dots, L$). For each searched point $\vec{P}_j$ ($\vec{P}_j = \{P_{1j}, P_{2j}, \dots, P_{Kj}\}$), we evaluate it with $Cost(\vec{P}, \vec{\phi}(\vec{P}))$ and $Time(\vec{P}, \vec{\phi}(\vec{P}))$ in Formula 5.19. The bid price vector with the minimal expected monetary cost while meeting the deadline constraint is returned as the optimal solution. With the logarithmic searching method, we reduce the searching space to $O((\log_2 H)^K)$.

5.3.3 Update Maintenance

Algorithm 6 Adaptive Optimization Algorithm

Input: $K, T_i, M_i, O_i, R_i, K_D, T_d, M_d, D_d$ and *Deadline* as defined in Table 5.1

Trace i: Spot price history of the spot instances in circle group i ($i = 1, 2, \dots, K$)

- 1: Select the on-demand instance type d ($d \in \{1, 2, \dots, K_D\}$) by solving Formula 5.12;
 - 2: Define the optimization window size T_m ;
 - 3: Initialize the execution time $T_e \leftarrow 0$, $T_o \leftarrow T_d$, $T \leftarrow T_m$, $D \leftarrow D_d$, $M \leftarrow M_d$ and $T_i \leftarrow T_i \times \frac{T_m}{T_o}$ ($i = 1, 2, \dots, K$);
 - 4: Set the original start point *Start* (The beginning of the application);
 - 5: **while** $T_o > 0$ **do**
 - 6: **if** $T_o < \text{Deadline} - T_e$ **then**
 - 7: /*Deadline could not be satisfied*/
 - 8: Run the application on on-demand instance type d from start point *Start* until the application is completed;
 - 9: **return**
 - 10: **else**
 - 11: **if** $T_o > T_m$ **then**
 - 12: /*Complete the application for the next interval*/
 - 13: $\text{Deadline}^* \leftarrow (\text{Deadline} - T_e) \times \frac{T_m}{T_o}$;
 - 14: **else**
 - 15: /*Complete the rest of the application*/
 - 16: $\text{Deadline}^* \leftarrow \text{Deadline} - T_e$, $T \leftarrow T_o$ and $T_i \leftarrow T_i \times \frac{T_o}{T_m}$ ($i = 1, 2, \dots, K$);
 - 17: Calculate Distribution function $f_i(P_i, t_i)$ and $S_i(P_i)$ based on *Trace i* ($i = 1, 2, \dots, K$);
 - 18: Compute $(\vec{P}, \vec{F})$ based on two-level optimization algorithm when the deadline is Deadline^* (Solve Formula 5.14);
 - 19: Hybrid run the application based on the configuration from start point *Start* until the time interval T is completed;
 - 20: $T_o \leftarrow T_o - T_m$;
 - 21: Compute the real execution time T_e ;
 - 22: Checkpointing the final state of the application as the next start point *Start* and update the trace information *Trace i* ($i = 1, 2, \dots, K$);
 - 23: **return**
-

Our algorithm needs to adapt to the variances of spot prices. Another reason is that the time complexity for the evaluation of each searched solution is $O(\bar{T}^K)$ ($\bar{T}$ is the average execution time), and the optimization overhead tends to be high when $\bar{T}$ is large.

In the adaptation, we define an optimization window with size T_m . Every T_m time, we update the

trace of spot price with the history of spot price from the previous window and generate the optimal solution for the optimization problem using our offline optimization method with the latest spot price history. Algorithm 6 shows our adaptive method of the optimization process.

Given an optimization window size T_m , we first identify the residual execution time of the application, which is denoted as T_o . In each optimization window, we exploit our two-level optimization algorithm to find the optimal configuration $(\vec{P}, \vec{F})$ and execute the application. At the end of an optimization window, we update the spot price trace with the spot price history in this window and update the failure rate function for later optimizations. One problem in the adaptive method is how to allocate the deadline for each optimization window. In our algorithm, we calculate the leftover time in a dynamic manner. Particularly, we first calculate the leftover time to the deadline (i.e., $Deadline - T_e$) and the residual execution time (i.e., T_o). Then we allocate the deadline to each optimization window according to the ratio of the two parts (see Line 13 of Algorithm 6). If the deadline could not be satisfied, we utilize on-demand instances to execute the rest of the application. With the adaptive method, the evaluation overhead of each searched solution is reduced to $O(\frac{\bar{T}}{T_m} \cdot T_m^K)$. And the total overhead for the algorithm is $O(\frac{\bar{T}}{T_m} \cdot T_m^K \cdot (\log_2 H)^K)$. In the experiments, we explicitly study how T_m affects the quality of our adaptation. With the suitable parameter settings, our proposed methods can significantly reduce the optimization overhead.

5.3.4 Implementation Details

There are some implementation issues that are worth discussion.

Optimizing selection of circle groups. Evaluating each searched bid price solution with $Cost(\vec{P}, \vec{\phi}(\vec{P}))$ and $Time(\vec{P}, \vec{\phi}(\vec{P}))$ induces a time complexity of $O(\frac{\bar{T}}{T_m} \cdot T_m^K)$, where K is the maximum number of circle groups for consideration. However, in real implementation, due to the redundancy of spot prices in different circle groups, we are able to successfully complete an MPI application with only κ ($\kappa < K$) circle groups. Based on this observation, we introduce a parameter named κ ($\kappa < K$). It means that

only κ out of K circle groups are selected for executing the MPI applications. We traverse all of possible cases each with a specific combination of κ circle groups and compute optimal $(\vec{P}, \vec{F})$ for each case. The case with minimal expected monetary cost serves as the final optimal solution. Based on this method in implementation, we can reduce the time complexity to $O(C_K^\kappa \cdot \frac{\bar{T}}{T_m} \cdot T_m^\kappa \cdot (\log_2 H)^\kappa)$. We experimentally study the impact of κ in Section 5.4.

Obtaining Failure Rate Function. We define failure rate function $f_i(P_i, t_i)$ to be the probability of the spot instances being failed due to the out-of-bid event at time t_i when the bid price is set to P_i for circle group i ($i = 1, 2, \dots, K$). When t_i is equal to T_i , it means that the application is completed on circle group i . Otherwise, the spot instances fail before the completion of the application in circle group i .

The possibility of failure $f_i(P_i, t_i)$ for circle group i can be calculated using the spot price history in histogram-based way. For each given bid price P'_i and execution time t'_i , the algorithm starts at a random point in the history of spot price in the previous two days. We check whether the spot price firstly becomes larger than P'_i at time t'_i . If so, we add one value to the counter *count*. We utilize the same process for G times, where G is large enough, and finally use $\frac{\text{count}}{G}$ as the probability of failing at (P'_i, t'_i) .

In the experiments, we explicitly study how the accuracy of failure rate function affects the quality of our decision.

Profiling. We estimate the execution time of MPI applications for different types of instance using TAU (Tuning and Analysis Utilities) [84] with the following profile: $\langle \#Isr, Data_{send}, Data_{rev}, IO_{seq}, IO_{rnd} \rangle$. $\#Isr$ represents the total number of instructions to be executed. $Data_{send}$ and $Data_{rev}$ are the amount of data that need to send and receive respectively (called by MPI function). IO_{seq} and IO_{rnd} are the amount of I/O data for sequential and random accesses respectively to local disk.

We estimate the execution time as the summation of its CPU, networking and I/O time. In the estimation, the CPU time is determined by the $\#Isr$ of the application as well as the CPU frequency of

the instance that the application is executed on. Similarly, the networking and I/O time is determined by networking and I/O data size divided by the network and I/O bandwidth, respectively. The total execution time is the sum of the three parts.

Checkpointing. We implement our system based on OpenMPI [21] with BLCR [22]. This checkpoint/restart tool uses a system-level coordinated non-locking checkpointing strategy for implementation. The advantage is that BLCR does not require source code-level modifications and it does not significantly increase the length of runs in which no checkpoints are taken. Another problem in the implementation is where to store the checkpoint. The checkpoint could not be stored in local disk, because the data may be lost at any time when the spot instance is terminated. We choose to use Amazon S3 (Simple Storage Service) to store the checkpoint for two reasons. First, Amazon S3 gives access to the same highly reliable and fast data storage infrastructure for any developer. Second, the storage resource is inexpensive. The monetary cost is about \$0.03/GB per month. Compared with the cost for MPI executions, the cost for storage is ignorable in our experiments (less than 0.1%).

5.4 Evaluations

5.4.1 Experimental Setup

Applications. We apply our model to NPB (NAS Parallel Benchmarks) [23]) kernels version 2.4. To measure the cost of MPI-based applications on spot instances, we choose three types of applications, 1) *computation-intensive applications*, including LU (Lower-upper Gauss-Seidel solver), SP (Scalar Penta-diagonal solver) and BT (Block Tri-diagonal solver); 2) *communication-intensive applications*, including FT (Fast Fourier Transform) and IS (Integer Sort); 3) *IO-intensive application*, BTIO (Block Tri-diagonal solver with IO subtypes). The number of processes is set to 128. The default problem size is CLASS D. We run each of the applications multiple times (100 to 200 times) to extend to large scale computing. We further apply our algorithm to one real-world application, LAMMPS (Large-

scale Atomic/Molecular Massively Parallel Simulator) [24]. It aims at simulating the movement, position and other attributes of atoms or molecules with interaction forces exerted on one another.

Comparisons. We execute the applications on on-demand instance with the best performance (minimal execution time) and denote the method as *Baseline*. In the experiments, the monetary cost and execution time are normalized to Baseline. We denote the monetary cost of Baseline as *Baseline Cost* and the execution time of Baseline as *Baseline Time*. We set *tight deadline* as 5% larger than Baseline Time and *loose deadline* as 50% larger than Baseline Time.

We denote our proposed optimization algorithm as *SOMPI* (short for Spot and On-demand MPI). We evaluate SOMPI in the following aspects, including (1) comparison with the state-of-the-art algorithm (Section 5.4.3.1), (2) comparison with simple utilization of on-demand and spot instances (Section 5.4.3.2) and (3) comparison with separate fault-tolerance techniques (Section 5.4.4.2).

We use two complementary evaluation approaches including real experiments and simulations.

Experiments on Amazon EC2. The real-world experiments were performed on Amazon EC2 in August 2014, with the focus on assessing the practical performance impact of our proposed approach. With the real experiments, we present the overall monetary cost comparison on different approaches. On Amazon EC2, we consider different instance types, including m1.small, m1.medium, c3.xlarge and cc2.8xlarge, in us-east-1a, us-east-1b and us-east-1c zones, as the candidates of circle groups. We choose m1.small and m1.medium for their low price and select c3.xlarge and cc2.8xlarge (cluster compute instance) for their high computational power. In our experiments, one process is attached to one core. For each instance type, the number of instances is determined by the number of cores in one instance. For example, we utilize 128 m1.small instances to execute the applications in NPB, because the total number of processes is 128 and there is only one core in one m1.small instance. We run each of the applications on Amazon EC2 for more than 100 times and calculate the expected monetary costs.

Simulation. As for simulations, we use the real trace of spot price in us-east-1a, us-east-1b and us-east-1c zones in 2014. With the simulations, we are able to verify the accuracy and efficiency of our model in a fully controlled manner.

The spot environment of Amazon EC2 is dynamic. For repeatable experiments on studying different settings and traces, we use the method of replaying the trace from the spot market, and calculate the monetary cost given the spot price in the trace. We randomly choose a start point in the trace and compare our bid price with the spot price along the time. If our bid price is lower than the spot price at that point, we treat the application as terminated and plus an overhead of recovery when it is restarted. Otherwise, we keep on calculating the running time of the application. We repeat the simulation for one million times and calculate the expected cost.

5.4.2 Parameter Study

We evaluate different parameters and compare the monetary cost and execution time, as well as the optimization overhead. We use simulation to control the experimental environment. For each experiment, we vary one parameter while keeping other parameters fixed to their default settings ($slack = 20\%$, $\kappa = 4$ and $T_m = 15$). In the following experiments, we use BT as an example and we obtain similar results for other applications.

Slack. In order to compare the impact of different *slack*, we fix the *deadline'* (the deadline for the on-demand execution) as Baseline Time. We compare the monetary cost and execution time in different *slack*.

Based on the comparison, we observe that: (1) When the *slack* is smaller than 20%, the monetary cost reduces and the execution time increases as the slack increases. (2) When the *slack* is larger than 20%, the monetary cost does not further reduce when the *slack* increases and the longest execution time keeps at 1.16 times of the Baseline Time. Based on the observations, we select the *slack* as 20% in our experiments.

κ . In this experiment, we find that when κ is larger than 4, the monetary cost reduction is very small but the total optimization overhead for calibration and searching becomes very large. When κ is 10, the total overhead is 2 times larger than Baseline Time. However, when κ is equal to 4, the total optimization overhead is smaller than 1% of Baseline Time. Thus, we choose $\kappa=4$ as our default setting.

T_m . We vary optimization window size T_m and compare the monetary cost. When T_m is around 15 (hours), the monetary cost reduction is the lowest. When T_m is too small (less than 10 hours), the frequent checkpointing and recovery after each interval T_m lead to extra monetary cost. Furthermore, the large overhead results to tight deadline to optimize the solution. When T_m is too large (larger than 20 hours), the dynamics of spot price leads to the change of the optimal solution.

5.4.3 Results on Amazon EC2

We present the following results related to Amazon EC2. First, we present the monetary cost and performance comparison with the state-of-the-art algorithm [1]. Second, we evaluate the monetary cost in comparison with the heuristic algorithms of utilizing spot and on-demand instances. Third, we study the impact of deadline requirements to the monetary cost optimization in SOMPI. For a fair comparison, all the results include the optimization overhead. The optimization overhead is generally smaller than 1% of the total execution time.

5.4.3.1 Comparison with state-of-the-art algorithm

In comparison with the state-of-the-art algorithm [1], we evaluate the following approaches.

On-demand. We select the type of on-demand instance with the smallest expected monetary cost, which satisfies the deadline requirement at the same time. This simulates the monetary cost optimization with only on-demand instances.

Marathe. Marathe et al. [1] propose to exploit spot instance in different zones as redundancy to reduce the monetary cost. In their algorithm, they utilize CC2 instances (cc2.8xlarge) as default setting. This approach is the state-of-the-art monetary cost optimization with spot instances for MPI-based applications.

Marathe-Opt. We optimize Marathe by selecting the suitable instance type. We compare different types of spot instance based on their algorithm and show the results with the minimal monetary cost.

In the experiments, we first run On-demand, immediately followed by Marathe, Marathe-Opt and SOMPI. Each experiment was ran for 100 times. Figure 5 shows the monetary cost comparison for three different kinds of applications, including computation-intensive, communication-intensive and IO-intensive applications, and a real-world application. Overall, SOMPI outperforms other comparisons for all applications in both loose and tight deadline. On average, compared with On-demand, Marathe and Marathe-Opt, SOMPI reduces the monetary cost by 70%, 48% and 20%, respectively. We also find that Marathe, Marathe-Opt and SOMPI are all sensitive to the accuracy of estimated execution time. The variation of I/O and network performance can affect the accuracy of the prediction, and further affect the optimization results. We have studied the impact of inaccurate profiling results and find that our proposed method can still outperform other algorithms when the estimated execution time is inaccurate. We propose our major observations for each type of applications.

Computation-Intensive Applications. We have two major observations from the monetary cost comparison under different deadlines. (1) Under loose deadline, the monetary cost of Marathe is 36% larger than Marathe-Opt. The reason is that cc2.8xlarge is the most powerful instance with the shortest execution time but the highest monetary cost for computation-intensive applications. Marathe-Opt can select other types of spot instance to reduce the monetary cost. For tight deadline requirement, Marathe and Marathe-Opt have equal monetary cost, because both Marathe and Marathe-Opt select the most powerful instance type (cc2.8xlarge) to meet the deadline requirement. (2) SOMPI can reduce 8%-25% of monetary cost for Marathe-Opt in different deadline requirements. There are two

reasons. First, SOMPI can select different instance types to run the applications, but Marathe-Opt can only select one instance type for replicated execution. From more candidates, SOMPI can select the suitable instance type with lower spot price for different cases. Second, SOMPI sets the bid price and checkpoint intervals by solving an optimization problem, which can further reduce the monetary cost.

Communication-Intensive Applications. We have the following observations. (1) The best instance type to execute communication-intensive applications is cc2.8xlarge. The reason is that for cc2.8xlarge instance, the network is much better than the network of m1.small, m1.medium and c3.xlarge. On another hand, many processes in cc2.8xlarge are running in the same instance and they utilize shared memory instead of exchanging message through the network. (2) The monetary cost of all the approaches is less than Baseline Cost in both loose and tight deadline requirement, but the cost reduction is less than the reduction for running computation-intensive applications. The reason is that for communication-intensive applications, cc2.8xlarge instances lead to the minimal monetary cost with the shortest execution time. But for computation-intensive applications, the selection of less powerful instance type (e.g. m1.small or m1.medium) can further reduce the monetary cost. In this case, we can make a trade-off between performance and monetary cost. (3) Monetary cost reduction for SOMPI between tight and loose deadline requirement is 5% (from 35% to 40%), which is much smaller than the span for executing computation-intensive applications (30%, from 40% to 70%). It indicates that the monetary cost optimization for communication-intensive applications is not sensitive to the deadline requirements. (4) Marathe and Marathe-Opt have equal monetary costs in both tight and loose deadline requirements. The reason is that for communication-intensive applications, both of the two the approaches select the most powerful instance type (cc2.8xlarge) to meet the deadline requirement.

IO-Intensive Application. We have two observations for IO-intensive application. (1) Compared with cc2.8xlarge, m1.small and m1.medium have lower costs and higher performance for IO-intensive applications. The reason is that when running the same application for different types of instance,

the number of cc2.8xlarge instances is much smaller than the number of m1.small and m1.medium instances. In this case, the degree of IO parallelism is much smaller than the other three types. (2) The monetary cost of Marathe is higher than Baseline Cost and On-demand, because Marathe only selects cc2.8xlarge instance, which is not efficient for IO-intensive applications.

Real-world Application. In executing the real-world application, LAMMPS, we fix the problem size and vary the number of processes. We have three observations. (1) When the number of processes is 32, the monetary cost reduction is sensitive to the deadline requirements. Compared with Baseline Cost, SOMPI reduces 75% and 45% of total costs in loose and tight deadline requirements, respectively. (2) When the number of processes is 128, the monetary cost reduction is similar in both loose and tight deadline requirements. Compared with Baseline Cost, the span of improvement for SOMPI between tight and loose deadline requirements is 3% (from 57% to 60%). (3) In loose deadline, as the number of processes increases, the reduction of monetary cost for On-demand, Marathe-Opt and SOMPI is decreasing. But for Marathe, the reduction of monetary cost is similar in different number of processes.

The reason for these observations is that as the total number of atoms/molecules is fixed, when the number of processes is small, the number of atoms/molecules for each process is large, which leads to high computation requirement. The application is computation-intensive and On-demand, Marathe-Opt and SOMPI select powerless instance (e.g. m1.small, m1.medium) to reduce monetary cost. As the number of processes increases, the communication portion is increasing. When the number of processes is large, the number of atoms/molecules for each process is small and the number of atoms/molecules in different processes, which need to exchange information (e.g. speed, position), is large. The application becomes communication-intensive. On-demand, Marathe-Opt and SOMPI select more powerful instance (cc2.8xlarge) and the reduction of monetary cost is less. Marathe only selects cc2.8xlarge to execute the application and the reduction of monetary cost is similar.

Overall Performance Comparison. In comparison with the state-of-the-art algorithm, we further study the performance in different deadline requirements. Table 5.2 shows the normalized execution

time comparison for Marathe-Opt and SOMPI. We have the following observations. (1) Marathe-Opt and SOMPI have similar execution time in both loose and tight deadline requirements. (2) In loose deadline, the execution time of all the applications is much less than the deadline. The performance decrease of communication-intensive applications is less than computation-intensive and IO-intensive applications. (3) In tight deadline, the execution time of all the applications is very near to the deadline.

5.4.3.2 Comparison with heuristic algorithms

In comparison with simple utilization of on-demand and spot instances, we evaluate the following approaches.

On-demand. We defined this approach in Section 5.4.3.1.

Spot-Inf. We choose a simple strategy to use spot instances to run our MPI applications. In this approach, we first fix the bid price as infinite (in our experiment we use \$999 dollars) so that the spot instance can be terminated in extremely low probability, which we can ignore. Then we choose the type of spot instances with minimal monetary cost to run the application.

Spot-Avg. In this approach, for each type of spot instances, we fix the bid price as the average price in the price history.

Figure 5.4 shows the evaluation results in loose and tight deadline requirements. In this experiment, we evaluate three types of applications, including computation-intensive application (denoted as Computation), communication-intensive application (denoted as Communication) and IO-intensive application (denoted as IO). We show the *average* evaluation results for each category.

Deadline	App Alg	Comp.Intensive			Comm.Intensive		IO Intensive
		BT	LU	SP	FT	IS	BTIO
Loose	Marathe-Opt	1.39	1.40	1.34	1.17	1.23	1.25
	SOMPI	1.42	1.38	1.35	1.18	1.22	1.27
Tight	Marathe-Opt	1.05	1.04	1.05	1.03	1.04	1.03
	SOMPI	1.04	1.05	1.03	1.03	1.04	1.04

Table 5.2: Normalized Execution time comparison for Marathe-Opt and SOMPI

We have the following observations. (1) Both Spot-Inf and Spot-Avg outperforms On-demand for all applications in different deadline requirements. It shows the importance of utilizing spot instance and indicates that spot instance can reduce the monetary cost even without any complex fault-tolerance techniques. (2) Compared with Spot-Inf and Spot-Avg, SOMPI reduces 28%, 38% of total costs in loose deadline requirement and 20%, 22% of total costs in tight deadline requirement. It indicates that combining the two fault-tolerance mechanisms, checkpointing and replication, together brings great advantages for dealing with the dynamic spot prices. (3) The variance for Spot-Inf is much larger than SOMPI. The reason is that the spot price is dynamic. When the price becomes much larger than on-demand instance, the infinite bidding strategy could not save the money. For SOMPI, the suitable setting of bid price with checkpointing and replicated execution techniques can avoid the worst case and reduce the variance.

5.4.3.3 Detailed study for SOMPI

We study the relationship between the deadline requirement and the monetary cost. Figure 5.5 shows the monetary cost when we vary the deadline requirements. The value of x-axis denotes how much larger than Baseline Time (Deadline subtracts Baseline Time). The loose and tight deadline requirements in the previous experiment are 0.5 and 0.05, respectively. We select BT, FT and BTIO as examples and we obtain similar results for other applications. We propose our major observations for each application.

BT. As the deadline increases, the monetary cost can reduce to about 70%. When the deadline is equal to Baseline Time, the monetary cost can reduce to about 10%. When the deadline is small, we can only choose the type of on-demand instance with the smallest execution time and highest monetary cost (cc2.8xlarge). As the deadline increases, we can select the other types with longer execution time and lower monetary cost. The points for selecting the new type of on-demand instance is denoted by the arrows in Figure 5.5(a). In our experiment, we first utilize the type of cc8.xlarge

on-demand instance to guarantee the completion of the application. As the deadline becomes larger, we can orderly select c3.xlarge, m1.medium and m1.small.

FT. The monetary cost can reduce to about 50% at most. For communication-intensive applications, cc2.8xlarge is the most suitable type with smallest monetary cost and minimum execution time. When the deadline requirement extends to about 10% larger than Baseline Time, the monetary cost reduction reaches to maximum value.

BTIO. The cost can reduce to more than 60%, compared with Baseline Cost. As the deadline increases to about 10% larger than Baseline Time (denoted by the arrow in Figure 5.5(c)), we select m1.small instead of m1.medium in order to further reduce the monetary cost. When the deadline requirement extends to about 20% larger than Baseline Time, the monetary cost reduction reaches to maximum value.

5.4.4 Results with Simulation

5.4.4.1 Failure Rate Function and Model Accuracy

Accuracy of Failure Rate Function. We verify the accuracy of the failure rate function $f(\vec{P}, \vec{t})$. In the first step, we randomly choose the history of the spot price in the threshold for four days from the trace. We treat the data of the first 3 days as the training data and calculate the failure rate function based on it. Then we use the data of the last day as the testing data and recalculate the failure rate function. After that, we compare the difference between the two failure rate functions. In the final step, we repeat the previous steps for several times and compute the average difference. We define the real value of the failure rate function as A and the value estimated from the training data as A' . The relative difference is defined as $\frac{|A-A'|}{|A|}$. In our experiments, we observe that the failure rate function is sufficiently accurate. About 90% of the relative difference is less than 3% and 98% is less than 5%.

Accuracy of Model. We use the simulation to verify whether Formula 5.1 can represent the expected monetary cost in our model. In this experiment, we design the simulation based on the trace.

More details have been introduced in section 5.4.1. We use the Monte Carlo method to calculate the expected cost and compare with the calculation by Formula 5.1. We still calculate the relative difference in the simulation. In our experiments, we observe that 20% of the relative difference is less than 5% and 40% is between 5% and 10%. The largest relative difference is only 15%.

5.4.4.2 Comparison with individual fault-tolerance mechanisms

In the experiments, we compare the approaches of executing the applications without any fault-tolerance techniques (denoted as *All-Unable*), without replicated executions (denoted as *w/o-RP*), without checkpointing (denoted as *w/o-CK*), without update maintenance technique (denoted as *w/o-MT*) and SOMPI.

Figure 5.6 shows the evaluation results in loose and tight deadline requirements. We observe that: (1) Compared with All-Unable, w/o-RP and w/o-CK outperform less than 5% in different deadline requirements. It indicates that single fault-tolerance mechanism does not efficiently reduce the monetary cost. (2) Compared with w/o-RP and w/o-CK, SOMPI outperforms more than 25% on average. It indicates the necessary of combining the two fault-tolerance mechanisms. The reason is that the optimizing solution space of SOMPI is much larger than the solution spaces of All-Unable, w/o-RP and w/o-CK. (3) The monetary cost of SOMPI is about 15% less than the monetary cost of w/o-MT. Furthermore, the variance of SOMPI is much less than w/o-MT. It indicates that SOMPI can reduce both the monetary cost and the variance efficiently. The reason is that our algorithm consider the dynamic of spot price and update our design when the distribution is changed.

(a) Computation-intensive Applications

(b) Communication-intensive Applications

(c) IO-intensive Application

(d) Real-world Application (LAMMPS)

Figure 5.3: Monetary cost comparison in different deadline

Figure 5.4: Monetary cost comparison with heuristic algorithms

(a) Computation-intensive (BT) (b) Communication-intensive (FT) (c) IO-intensive (BTIO)

Figure 5.5: Monetary cost in different Deadline: BT, FT and BTIO

Figure 5.6: Monetary cost comparison with fault-tolerance mechanisms

Chapter 6

Conclusion and future work

6.1 Conclusion

In the first work, 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 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.

In our second work, we revisit network performance aware optimizations of distributed applications running on virtual clusters in IaaS cloud. We find that the important assumptions of existing network performance aware optimizations are no longer valid, because the network topology information is not available, and direct use of a few network performance measurements cannot capture the long-term performance. We propose a novel approach based on RPCA (a well-known problem in computer vision) to find the constant component from dynamic network performance while minimizing the difference between the constant component and network performance. Moreover, the RPCA-

based approach adapts to significant changes in the network performance. Based on our approach, we are able to determine the effectiveness of network optimizations on the applications running in the virtual cluster. In the experiments, we find that Amazon EC2 has relatively stable network performance ($N_E = 0.1$), and network performance aware optimizations are still important on Amazon EC2. Moreover, our RPCA-based approach is more robust than other approaches for different degree of network dynamics, and always outperforms the other comparison approaches. On Amazon EC2, our RPCA-based approach achieves 20–40% and 8–20% over the baseline approach and the approach based on direct use of the ad-hoc measurements, respectively. In the simulation on ns-2, we compare with the topology-aware algorithm and find that our approach obtains 25–40% performance improvement.

In the third work, we formulate a monetary cost optimization problem for MPI applications and leverage both on-demand and spot instances to guarantee the deadline constraint. We developed a cost model guided approach to combine checkpoint and replication in an adaptive manner. We implement our model on Amazon EC2 and evaluate its efficiency with NPB benchmarks and one real-world application, LAMMPS. Our experimental results show that 1) due to the dynamics of spot price, it is necessary to adaptively choosing checkpoint and replication techniques and 2) our approach can reduce the monetary cost than the state-of-the-art algorithm [1] by 20% on average (up to 40%) while satisfying the deadline requirement.

6.2 Future Work

We discuss two directions which we are interested to extend our work in the future.

6.2.1 Optimization on Exascale Cloud Environments

We plan to extend our optimization techniques to exascale cloud environments in the future. However, in our previous research, the all-link performance measurement leads to huge calibration time as the

number of instances increases. It leads to low efficiency for utilizing network performance aware optimizations, which largely limit the scalability of our optimization techniques. In order to extend our work, we plan to consider both effectiveness and efficiency, where effectiveness refers to the performance improvement for the network performance aware optimization and efficiency refers to the overhead of enabling the network performance aware optimization.

6.2.2 Optimization in Geo-distributed Cloud

Many HPC applications are utilizing the scalable and flexible cloud resources. Currently, most cloud providers offer geo-distributed cloud services to serve users that need to run their applications in multiple geographic locations. For example, Amazon EC2 currently has 11 service regions geographically distributed around the world, Windows Azure operates from 20 regions and Google Compute Engine (GCE) provides services from four geo-distributed regions. Our second direction is the performance and monetary cost optimization problem in geo-distributed cloud environment.

The geo-distributed optimization problem is challenging due to several reasons. First, the network bandwidth in the geo-distributed cloud environment is much more heterogeneous than the network bandwidth inside one cloud. For example, the network bandwidth within a single geographic region is usually much faster than the bandwidth across regions. Furthermore, the upload and download bandwidths of a single data center can also be highly heterogeneous due to widely different link capacities and network traffic. Those features are very different to the single cloud environments. Second, regulations and laws such as data privacy regulations add another dimension of constraints on the problem.

6.2.3 Optimization for Scientific Workflows in IaaS Cloud

We consider to extend our optimization techniques to another programming paradigm, scientific workflows. The workflow model is widely utilized by many researchers to analyze the large-scale

scientific applications. Due to the pay-as-you-go pricing scheme, performance and cost optimizations of scientific workflows in Infrastructure-as-a-service (IaaS) clouds becomes an important and complicated problem. The complexities come from severe cloud performance and price dynamics and various user requirements on performance and cost.

In study of the workflow optimization problems, we find that the existing studies of the workflows are not aware of the cloud dynamics, and assume static workflow execution time in the cloud. We plan to propose new workflow optimization approaches, which is based on the fully consideration of the dynamic cloud environment, including the network heterogeneity, network interference and so on.

Bibliography

- [1] A. Marathe, R. Harris, D. Lowenthal, B. R. de Supinski, B. Rountree, and M. Schulz, “Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on Amazon EC2,” in *HPDC’14*, 2014.
- [2] W. Lu, J. Jackson, and R. Barga, “Azureblast: a case study of developing science applications on the cloud,” in *1st Workshop on Scientific Cloud Computing*, 2010.
- [3] A. Nagavaram, G. Agrawal, M. A. Freitas, K. H. Telu, G. Mehta, R. G. Mayani, and E. Deelman, “A cloud-based dynamic workflow for mass spectrometry data analysis,” in *ESCIENCE*, 2011.
- [4] S. Nunez, B. Bethwaite, J. Brenes, G. Barrantes, J. Castro, E. Malavassi, and D. Abramson, “Ngtaphra: A massively parallel, nimrod/g-enabled volcanic simulation in the grid and the cloud,” in *ESCIENCE*, 2010.
- [5] K. A. C. S. Ocana, D. de Oliveira, J. Dias, E. Ogasawara, and M. Mattoso, “Optimizing phylogenetic analysis using scihmm cloud-based scientific workflow,” in *ESCIENCE*, 2011.
- [6] J. C. Mudge, P. Chandrasekhar, G. S. Heinson, and S. Thiel, “Evolving inversion methods in geophysics with cloud computing - a case study of an escience collaboration,” in *ESCIENCE*, 2011.
- [7] J.-S. Vöckler, G. Juve, E. Deelman, M. Rynge, and B. Berriman, “Experiences using cloud computing for a scientific workflow application,” in *ScienceCloud*, 2011.
- [8] Amazon Case Studies, <http://aws.amazon.com/hpc-applications/>.
- [9] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, “Cloud versus in-house cluster: evaluating amazon cluster compute instances for running mpi applications,” in *SC*, 2011.

- [10] W. Lu, J. Jackson, and R. Barga, “Azureblast: a case study of developing science applications on the cloud,” in *HPDC*, 2010.
- [11] S. Núñez, B. Bethwaite, J. Brenes, G. Barrantes, J. Castro, E. Malavassi, and D. Abramson, “Ng-tephra: A massively parallel, nimrod/g-enabled volcanic simulation in the grid and the cloud,” in *e-Science*, 2010.
- [12] Y. Gong, B. He, and J. Zhong, “Network performance aware MPI collective communication operations in the cloud,” *IEEE TPDS*, 2013.
- [13] Y. Gong, B. He, and D. Li, “Finding constant from change: Revisiting network performance aware optimizations on iaas clouds,” in *SC*, 2014.
- [14] C. Evangelinos and C. Hill, “Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2,” *Ratio*, 2008.
- [15] J. C. Mouriño, M. J. Martín, R. Doallo, D. E. Singh, F. F. Rivera, and J. D. Bruguera, “The stem-ii air quality model on a distributed memory system,” in *Parallel Processing Workshops, 2001. International Conference on*, 2001.
- [16] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “Dcell: a scalable and fault-tolerant network structure for data centers,” in *SIGCOMM*, 2008.
- [17] M. Chowdhury and I. Stoica, “Coflow: An application layer abstraction for cluster networking,” tech. rep., EECS Department, University of California, Berkeley, 2012.
- [18] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, “Managing data transfers in computer clusters with orchestra,” in *SIGCOMM ’11*.
- [19] S. Srikantaiah, A. Kansal, and F. Zhao, “Energy aware consolidation for cloud computing,” in *HotPower*, 2008.
- [20] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *IMC*, 2010.
- [21] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kam-badur, B. Barrett, A. Lumsdaine, *et al.*, “Open mpi: Goals, concept, and design of a next generation mpi implementation,” in *PVM/MPI’04*, 2004.

- [22] P. H. Hargrove and J. C. Duell, “Berkeley lab checkpoint/restart (blcr) for linux clusters,” in *JPCS’06*, 2006.
- [23] “The nas parallel benchmarks,” (<http://www.nas.nasa.gov/publications/npb.html>).
- [24] “Lammps molecular dynamics simulator,” tech. rep., <http://lammps.sandia.gov/>.
- [25] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, “Towards predictable datacenter networks,” in *SIGCOMM*, 2011.
- [26] T. Zou, R. Le Bras, M. V. Salles, A. Demers, and J. Gehrke, “Cloudia: a deployment advisor for public clouds,” in *PVLDB*, 2012.
- [27] R. Thakur and R. Rabenseifner, “Optimization of collective communication operations in mpich,” *International Journal of High Performance Computing Applications*, 2005.
- [28] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaatz, and R. A. F. Bhoedjang, “Magpie: Mpi’s collective communication operations for clustered wide area systems,” in *PPoPP*, 1999.
- [29] 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.
- [30] 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.
- [31] M. Banikazemi, V. Moorthy, and D. K. Panda, “Efficient collective communication on heterogeneous networks of workstations,” in *ICPP ’98*.
- [32] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, “Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters,” in *ICS’06*, 2006.
- [33] J. Zhang, J. Zhai, W. Chen, and W. Zheng, “Process mapping for mpi collective communications,” in *European Conference on Parallel Processing*, Springer, 2009.
- [34] Y. Gong, B. He, and J. Zhong, “Network performance aware MPI collective communication operations in the cloud,” *IEEE TPDS*, 2013.

- [35] T. Hoefler and M. Snir, “Generic topology mapping strategies for large-scale parallel architectures,” in *ICS*, 2011.
- [36] T. Lei and S. Kumar, “A two-step genetic algorithm for mapping task graphs to a network on chip architecture,” in *Proceedings of the Euromicro Symposium on Digital System Design*, 2003.
- [37] J. Hu and R. Marculescu, “Energy-and performance-aware mapping for regular noc architectures,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2005.
- [38] C. Ernemann, V. Hamscher, and R. Yahyapour, “Economic scheduling in grid computing,” in *JSSPP’02*, 2002.
- [39] Y. C. Lee, A. Y. Zomaya, and M. Yousif, “Reliable workflow execution in distributed systems for cost efficiency,” in *GRID’10*, 2010.
- [40] J. Gray and G. Graefe, “The five-minute rule ten years later, and other computer storage rules of thumb,” *ACM Sigmod Record*, 1997.
- [41] M. Adler, J.-Y. Cai, J. K. Shapiro, and D. Towsley, “Estimation of congestion price using probabilistic packet marking,” in *INFOCOM’03*, 2003.
- [42] Y. Song, M. Zafer, and K.-W. Lee, “Optimal bidding in spot instance market,” in *INFOCOM’12*, 2012.
- [43] M. Mazzucco and M. Dumas, “Achieving performance and availability guarantees with spot instances,” in *HPCC’11*, 2011.
- [44] W. Guo, K. Chen, Y. Wu, and W. Zheng, “Bidding for highly available services with low price in spot instance market,” in *HPDC’15*, 2015.
- [45] M. Taifi, J. Y. Shi, and A. Khreishah, “Spotmpi: A framework for auction-based HPC computing using Amazon spot instances,” in *ICA3PP’11*, 2011.
- [46] E. J. Candès, X. Li, Y. Ma, and J. Wright, “Robust principal component analysis?,” *JACM*, 2011.
- [47] J.-F. Cai, E. J. Candès, and Z. Shen, “A singular value thresholding algorithm for matrix completion,” *SIAM J. on Optimization*, 2010.
- [48] S. Ji and J. Ye, “An accelerated gradient method for trace norm minimization,” in *ICML ’09*.

- [49] RPCA Accelerated Proximal Gradient(APG) Method , http://perception.csl.illinois.edu/matrix-rank/sample_code.html.
- [50] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, “Uncoordinated checkpointing without domino effect for send-deterministic mpi applications,” in *IPDPS’11*, 2011.
- [51] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, “A survey of rollback-recovery protocols in message-passing systems,” *ACM CSUR*, 2002.
- [52] R. Guerraoui and A. Schiper, “Fault-tolerance by replication in distributed systems,” in *Reliable Software Technologies Ada-Europe’96*, 1996.
- [53] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, “MPI-FT: Portable fault tolerance scheme for MPI,” *Parallel Processing Letters*, 2000.
- [54] S. Genaud and C. Rattanapoka, “P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids,” *Journal of Grid Computing*, 2007.
- [55] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok, “Volpexmpi: an mpi library for execution of parallel applications on volatile nodes,” in *PVM/MPI*, 2009.
- [56] B. Lowekamp, D. O’Hallaron, and T. Gross, “Topology discovery for large ethernet networks,” *SIGCOMM*, 2001.
- [57] B. Donnet, P. Raoult, T. Friedman, and M. Crovella, “Efficient algorithms for large-scale topology discovery,” *SIGMETRICS Perform. Eval. Rev.*, 2005.
- [58] MPICH2, <http://www.mcs.anl.gov/research/projects/mpich2/>.
- [59] R. Reussner, P. S. L. Prechelt, and M. Muller, “Skampi: A detailed, accurate mpi benchmark,” in *Recent advances in Parallel Virtual Machine and Message Passing Interface*, 1998.
- [60] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, “Vivaldi: a decentralized network coordinate system,” *SIGCOMM ’04*.
- [61] T. S. E. Ng and H. Zhang, “Towards global network positioning,” in *IMW ’01*.
- [62] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, “The nature of data center traffic: measurements & analysis,” in *SIGCOMM ’09*.

- [63] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, “The nature of data center traffic: measurements & analysis,” in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pp. 202–208, ACM, 2009.
- [64] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, ACM, 2010.
- [65] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*. 1952.
- [66] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang, “Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances,” in *CLOUD’13*, 2013.
- [67] A. Andrzejak, D. Kondo, and S. Yi, “Decision model for cloud computing under sla constraints,” in *MASCOTS’10*, 2010.
- [68] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, “See spot run: using spot instances for mapreduce workflows,” in *HotCloud’10*, 2010.
- [69] M. Malawski and et al., “Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds,” in *SC’12*, 2012.
- [70] H. Killapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis, “Schedule optimization for data processing flows on the cloud,” in *SIGMOD’11*, 2011.
- [71] J. Yu, R. Buyya, and C. K. Tham, “Cost-based scheduling of scientific workflow applications on utility grids,” in *e-Science’05*, 2005.
- [72] S. K. Garg, R. Buyya, and H. J. Siegel, “Time and cost trade-off management for scheduling parallel applications on utility grids,” *Elsevier FGCS*, 2010.
- [73] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, “Scheduling workflows with budget constraints,” in *Integrated Research in GRID Computing’07*, 2007.
- [74] A. C. Zhou and B. He, “Transformation-based monetary cost optimizations for workflows in the cloud,” *IEEE TCC*, 2014.
- [75] A. C. Zhou, B. He, X. Cheng, and C. T. Lau, “A declarative optimization engine for resource provisioning of scientific workflows in iaas clouds,” in *HPDC’15*, 2015.

- [76] J. Wu, J. Liu, P. Wyckoff, and D. Panda, "Impact of on-demand connection management in mpi over via," in *CLUSTER'02*, 2002.
- [77] S. Yi, A. Andrzejak, and D. Kondo, "Monetary cost-aware checkpointing and migration on amazon cloud spot instances," *IEEE TSC*, 2012.
- [78] S. Yi, D. Kondo, and A. Andrzejak, "Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud," in *CLOUD'10*, 2010.
- [79] Q. Zhang, Q. Zhu, and R. Boutaba, "Dynamic resource allocation for spot markets in cloud computing environments," in *UCC'11*, 2011.
- [80] X. He, P. Shenoy, R. Sitaraman, and D. Irwin, "Cutting the cost of hosting online services using cloud spot markets," in *HPDC'15*, 2015.
- [81] S. Moore, D. Cronk, K. S. London, and J. Dongarra, "Review of Performance Analysis Tools for MPI Parallel Programs," in *PVM/MPI'01*, 2001.
- [82] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2009.
- [83] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-L. Wang, and F. Cappello, "Optimization of cloud task processing with checkpoint-restart mechanism," in *SC'13*, 2013.
- [84] S. S. Shende and A. D. Malony, "The tau parallel performance system," *SAGE IJHPCA*, 2006.