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

Yifan Gong
NEWRI
Interdisciplinary Graduate School
Nanyang Technological University, Singapore

Bingsheng He
School of Computer Engineering
Nanyang Technological University, Singapore

Amelie Chi Zhou
School of Computer Engineering
Nanyang Technological University, Singapore

ABSTRACT

In this paper, we propose monetary cost optimizations for MPI-based applications with deadline constraints on Amazon EC2. Particularly, we consider to utilize two kinds of Amazon EC2 instances (on-demand and spot instances). As a spot instance can fail at any time due to out-of-bid events, fault tolerant executions are necessary. Through detailed studies, we have found that two common fault tolerant mechanisms, i.e., checkpoints and replicated executions, are complementary for cost-effective MPI executions on spot instances. We formulate the optimization problem and propose a novel cost model to minimize the expected monetary cost. The experimental results with NPB benchmarks on Amazon EC2 demonstrate that 1) it is feasible to run MPI applications with performance constraints on spot instances, 2) our proposal achieves significant monetary cost reduction compared to the state-of-the-art algorithm and 3) it is necessary to adaptively choose checkpoint and replication techniques for cost-effective and reliable MPI executions on Amazon EC2.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance

Keywords

Cloud, Monetary Cost, Fault-tolerance, MPI, Spot Instance

1. INTRODUCTION

Recently, we have witnessed many emerging high performance computing (HPC) applications developed and hosted in the cloud [43, 5, 28, 33]. As those applications are usually long-running jobs and are costly in the cloud, monetary cost and performance become important optimization factors [8, 23]. Message Passing Interface (MPI) is the key programming paradigm for developing HPC applications [17, 16]. That motivates us to investigate whether and how we can reduce the monetary cost for MPI-based applications with performance constraints in the cloud.

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807612

In cloud computing environments, monetary cost optimizations have been a hot research topic. Researchers have addressed various problems in this field: minimizing the cost given performance requirements [25, 6], optimizing the performance in the given budgets [9] and scheduling optimizations with both cost and performance constraints [29]. However, most of previous studies [26, 42, 15, 34, 47, 48] mainly focus on job/task scheduling for workflows, rather than MPI applications. On the other hand, previous studies on MPI-based application optimizations [13, 39] 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 charging the users with a fixed rate for on-demand instances, Amazon EC2 provides spot instances, whose prices are mainly determined by the supply and demand in the market. We analyze the spot price history and have the following observations: a) The spot instances are usually much cheaper than 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 [40, 6, 9] compared to solutions with on-demand instances only. b) Different instance types have different variations on the spot price. c) The spot price is highly dynamic in both spatial and temporal dimensions. In the spatial dimension, different instance types and instances of the same type but 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. These observations are consistent with the previous studies [4, 28]. We present more details about the spot trace study in Section 2.

Many previous studies [9, 41, 46, 40, 30, 21, 24] have studied HPC applications on spot instances. Leveraging spot instances appropriately can greatly reduce the monetary cost. However, a spot instance is terminated whenever the spot price is higher than the bid price set by users. Such out-of-bid events pose significant challenges to the cost optimizations of MPI applications with performance requirements. In this paper, 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 [30]. 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.

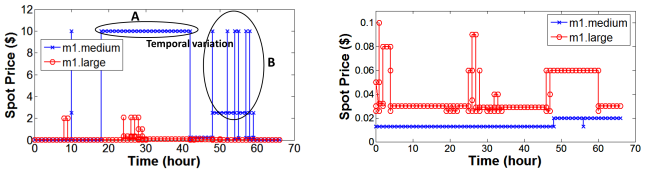
Combining checkpoints and replicated execution on Amazon EC2 requires making three key decisions: (D1) what instance types to choose; (D2) how much to bid for spot instances; (D3) when to checkpoint. First, Amazon EC2 provides a number of instance types, each with different prices and capabilities. Selecting 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 interval also matters. A short checkpoint interval leads to large overhead while a long interval 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 [30] has already studied the determination of bid price and checkpoint interval 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. Due to the large solution space, 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 are able to reduce the entire solution space from $N \times M$ to $N + M$. Second, we model the correlation between bid price and checkpoint interval to further reduce the dimension of the optimization 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.

We make the following contributions:

- We mathematically formulate the monetary cost optimization problem with performance guarantees for MPI applications in the Amazon EC2 spot market. We are able to find near optimal solutions for the problem with our proposed numerical methods.
- According to problem-specific features, we propose several dimensionality reduction techniques to reduce the optimization overhead while preserving the solution quality.
- We implement our optimization method on Amazon EC2 based on OpenMPI [14] using the BLCR library [22]. We evaluate our method on a number of applications, including LU, BT, SP, FT, IS, BTIO in NPB [2] benchmarks and a real-world application, LAMMPS [1]. Experimental results indicate that our proposed method can reduce 20% of total monetary cost on average (up to 40%) compared to the state-of-the-art algorithm [30].

The rest of paper is organized as follows. Section 2 gives the background on Amazon EC2 and the fault-tolerant techniques in MPI. Section 3 formulates the model to be an optimization problem, by aiming to compute the optimal bid price and checkpoint intervals. In Section 4, we present the optimization solution for this problem. We present our evaluation results in Section 5 and summarize the related work in Section 6. Section 7 concludes the paper.



(a) us-east-1a zone (b) us-east-1b zone

Figure 1: Spot price history of Amazon EC2 in three days

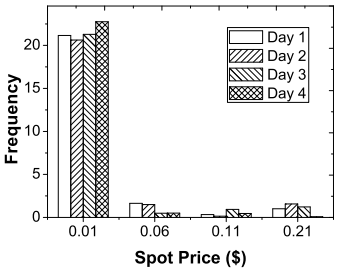


Figure 2: Histogram of m1.medium spot price in us-east-1a zone in four consecutive days

2. PRELIMINARY AND BACKGROUND

2.1 Spot Pricing Model

Amazon EC2 provides different types of virtual machines (instances), each with different computational capabilities and prices. There are multiple pricing models in the cloud, such as on-demand, spot and reservation. We focus on the on-demand and spot pricing models in this paper.

Different from the 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 need to bid the appropriate price they are willing to pay. The bid price is fixed once the instance is launched. If the 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 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.medium and m1.large instance types in two Amazon EC2 availability zones. Figure 1 illustrates the variation of the spot prices in three days.

Temporal variation. The spot price is not static, but changes along the time. The change of the spot price can be huge. For example, in Figure 1(a), the spot price of m1.medium instances in the us-east-1a zone 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 us-east-1a zone during 20 to 40 hours, highlighted with A in Figure 1(a)) and changing dramatically for some other time (e.g., spot price of m1.medium in us-east-1a zone during 50 to 60 hours, highlighted with B in Figure 1(a)). Thus, it is generally difficult or even impossible to predict the exact spot price, 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 1(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 1(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 feasible 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 shows the spot price histogram of m1.medium instances in the us-east-1a zone in four consecutive days. The spot price distributions in the four days are very close to each other. It is feasible and desirable to use the spot price history to estimate the probabilistic distribution of the spot price in a short time.

Implications to model design. Those observations have significant implications on our model design. First, the temporal and spatial price variations require special design of fault-tolerant mechanisms for reliability. This is particularly critical for MPI applications, where the failure of one MPI process usually cause the failure of the entire MPI application. We leverage the redundancies in different instance types and availability zones of Amazon EC2 to increase the probability of using spot instances to reduce the cost. Second, the dynamics in spot prices is a norm. It is impractical or unreliable to predict the exact next spot price. However, the probabilistic distribution of spot prices is predictable in a short time and we can use the spot price distribution to estimate the expected monetary cost.

2.2 Fault Tolerance in MPI

MPI is a de facto standard for distributed and parallel programs running on computer clusters and supercomputers. There are two types of commonly adopted techniques for fault tolerance, namely checkpoints and replicated execution. For checkpoint restart techniques, coordinated checkpointing and uncoordinated checkpointing are two major classes [11]. 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. There are two kinds of replicated execution techniques, namely primary-backup and active replication [20]. In our model, the client processes do not communicate with the replicas, and the active replication is more suitable. Due to the space limitation, we present the details in Appendix A of our technical report [18].

3. PROBLEM FORMULATION

We study the monetary cost optimization problem for MPI-based applications in the Amazon EC2 cloud, providing performance guarantees for the applications.We consider the problem in the spot market to benefit from the low prices of spot instances. 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.

3.1 Model Overview and Assumptions

3.1.1 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 spot and on-demand instances to reduce the monetary cost while meeting the performance requirements.

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 1: Notations in the model

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.

3.1.2 Assumptions

Without loss of generality, 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, $i = 1,2,\ldots,K$) 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 estimate the execution time of the application in different circle groups (denoted as T_i, $i = 1,2,\ldots,K$). There are a number of profiling tools [32] for this purpose. Note that the execution time refers to the productive time, excluding any checkpointing or recovery overhead. More implementation details can be found in Section 4.4.
- The spot prices in different availability zones are all independent. Our observation on history spot price traces confirms this assumption and the previous studies [30] also have similar findings. Based on this assumption, we estimate 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 (shown in Section 2). Based on this assumption, we can design the bid price and checkpoint interval with spot price history.

3.2 Problem Definition

We formulate the problem as a constrained optimization problem. Our optimization goal is to minimize the *expected* monetary cost of the MPI-based application while meeting the performance requirement. Table 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 ($i = 1,2,...K$) represent the bid price for circle group i. $P_i \in [0,\ H_i]$ ($i = 1,2,...K$), 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 ($i = 1,2,...K$) represent the checkpoint interval for circle group i. $F_i \in (0,\ T_i]$ ($i = 1,2,...K$), where T_i is the execution time of the application 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:

$$\begin{aligned} &minimize && E(Cost(\vec{P},\vec{F},d)) \\ &subject\ to && E(Time(\vec{P},\vec{F},d)) \leq Deadline \end{aligned} \tag{1}$$

3.2.1 Estimating Expected Monetary Cost

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

$$E(Cost(\vec{P},\vec{F},d)) = \sum_{\vec{t}} f(\vec{P},\vec{t}) \times Cost(\vec{P},\vec{F},\vec{t},d) \tag{2}$$

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 ($i = 1,2,...K$) 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. More details can be found in Section 4.4. 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:

$$f(\vec{P},\vec{t}) = \prod_{i=1}^{K} f_i(P_i,t_i) \tag{3}$$

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

$$Cost(\vec{P},\vec{F},\vec{t},d) = Cost^S(\vec{P},\vec{F},\vec{t}) + Cost^D(\vec{F},\vec{t},d) \tag{4}$$

where $Cost^S(\vec{P},\vec{F},\vec{t})$ (Formula 5) is the monetary cost for spot instances (or circle groups) and $Cost^D(\vec{F},\vec{t},d)$ (Formula 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, $t_i + O_i \times \lceil \frac{t_i}{F_i} \rceil$ 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.

$$Cost^S(\vec{P},\vec{F},\vec{t}) = \sum_{i=1}^{K} S_i \times (t_i + O_i \times \lceil \frac{t_i}{F_i} \rceil) \times M_i \tag{5}$$

To calculate the cost of on-demand instances $Cost^D(\vec{F},\vec{t},d)$, we first calculate the remaining execution time of the application, in the ratio of the entire execution time of the application (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 is $t_i - F_i$ and the remaining execution time is $T_i - (t_i - F_i) + R_i$, where R_i is the recovery overhead. The ratio

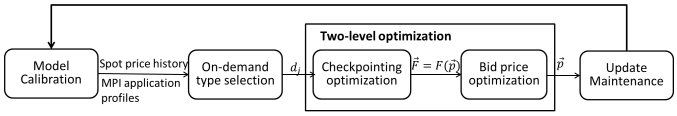


Figure 3: Overview of the optimization algorithm.

(as in Formula 7) is used to calculate the checkpoint which is the closest (the minimal value) to finish among all circle groups.

$$Cost^D(\vec{F},\vec{t},d) = \min_{i \in \{1,...,\ K\}} [Ratio(t_i,F_i)] \times T_d \times D_d \times M_d \tag{6}$$

$$Ratio(t_i,F_i) = \begin{cases} \frac{T_i - \max(t_i - F_i, 0) + R_i}{T_i} & if\ t_i < T_i \\ 0 & if\ t_i = T_i \end{cases} \tag{7}$$

3.2.2 Estimating Expected Execution Time

The expected execution time can be estimated as follows:

$$E(Time(\vec{P},\vec{F},d)) = \sum_{\vec{t}} f(\vec{P},\vec{t}) \times Time(\vec{F},\vec{t},d) \tag{8}$$

where $Time(\vec{F},\vec{t},d)$ is the total execution time for the application when circle group i ($i = 1,2,...K$) is terminated at time t_i ($i = 1,2,...K$). It consists of two parts as shown in Formula 9.

$$Time(\vec{F},\vec{t},d) = Time^S(\vec{F},\vec{t}) + Time^D(\vec{F},\vec{t},d) \tag{9}$$

where $Time^S(\vec{F},\vec{t})$ is the total execution time on spot instances and $Time^D(\vec{F},\vec{t},d)$ is the execution time on on-demand instances. We calculate the execution time on spot $Time^S(\vec{F},\vec{t})$ as shown in Formula 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 11. We exploit the checkpoint, which is the closest to finish (minimal value for Ratio) among all circle groups, to recover the application.

$$Time^S(\vec{F},\vec{t}) = \max_{i \in \{1,...,\ K\}} (t_i + O_i \times \lceil \frac{t_i}{F_i} \rceil) \tag{10}$$

$$Time^D(\vec{F},\vec{t},d) = \min_{i \in \{1,...,\ K\}} [Ratio(t_i,F_i)] \times T_d \tag{11}$$

4. MODEL OPTIMIZATION

In this section, we propose a numerical method to obtain the optimal solution for the problem (Formula 1). Figure 3 shows the 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.

4.1 On-demand Type Selection

On observing Formula 4 and 6, we can easily find that the monetary cost of on-demand instances is independent from the cost of spot instances. Formally, we have that if $\exists\ \vec{P'}$ and $\vec{F'}$ such that $E(Cost(\vec{P'},\vec{F'},d_1)) > E(Cost(\vec{P'},\vec{F'},d_2))$, then for $\forall\ \vec{P}$ and $\vec{F}$, $E(Cost(\vec{P},\vec{F},d_1)) > E(Cost(\vec{P},\vec{F},d_2))$. This means 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 on-demand instance type.

When optimizing the on-demand instance type, we have two principles in mind. First, the monetary cost induced by the on-demand instance should be minimized (Formula 12). Second, the on-demand instance should be able to meet the deadline requirement (Formula 13). Since the price of on-demand instances is fixed, the monetary cost of on-demand instances of type d can be calculated with the unit price of the instances D_d, the execution time of the application on the instances T_d and the number of utilized instances M_d (see Formula 12).

Formula 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 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 $Slack = \frac{Deadline - Deadline'}{Deadline}$. $Deadline'$ is the execution time allowed for on-demand instances. The difference between $Deadline$ and $Deadline'$ is the time reserved for checkpointing and recovery.

$$minimize \qquad Cost^{OD}(d) = T_d \times D_d \times M_d \tag{12}$$

$$subject\ to \qquad T_d \leq Deadline \times (1 - Slack) \tag{13}$$

Solving the above optimization problem is straightforward (given $Slack$ is determined in Section 5). After finding the optimal solution d^* to the above problem, we can simplify the problem model in Formula 1 as below.

$$minimize \qquad E(Cost(\vec{P},\vec{F})) = E(Cost(\vec{P},\vec{F},d^*)) \tag{14}$$

$$subject\ to \qquad E(Time(\vec{P},\vec{F})) = E(Time(\vec{P},\vec{F},d^*)) \leq Deadline$$

We denote the execution time of the application 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 6 and 11 can be simplified as below.

$$Cost^D(\vec{F},\vec{t}) = \min_{i\in\{1,\ldots,K\}} [Ratio(t_i,F_i)] \times T \times D \times M \tag{15}$$

$$Time^D(\vec{F},\vec{t}) = \min_{i\in\{1,\ldots,K\}} [Ratio(t_i,F_i)] \times T \tag{16}$$

4.2 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 propose 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.

4.2.1 Difficulty Analysis

One straight-forward idea for solving the problem in Formula 14 is to utilize convex optimization method [7]. In order to use the convex optimization method, it is necessary 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 $S_i(P_i)$ $(i = 1,2,...K)$ (see Formula 2, 4 and 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 in Section 2.

Second, for parameter $\vec{F}$, $E(Cost(\vec{P},\vec{F}))$ is not continuous when $F_i \leq t_i$ $(i = 1,2,\ldots,K$ and $t_i = 1,2,\ldots,T_i)$, because of the $\lceil \frac{t_i}{F_i} \rceil$ component in the calculation (see Formula 2, 4 and 5). Moreover, Formula 7 is not derivable when $F_i = t_i$ $(i = 1,2,\ldots,K$ and $t_i = 1,2,\ldots,T_i)$.

As the convex optimization method is not viable, we consider exploiting numerical methods to solve this problem. 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 propose two techniques to reduce the searching space in the following subsections.

4.2.2 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 4.2.1, 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 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 14.

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

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

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

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

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

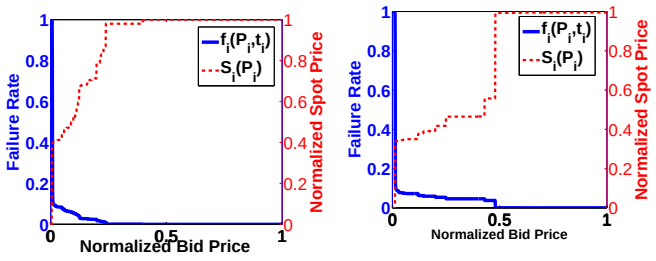
$$\forall \vec{F}, Cost(\vec{P}^*,\vec{\phi}(\vec{P}^*)) \leqslant Cost(\vec{P}^*,\vec{F})$$

Thus,

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

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 $(i = 1,2,\ldots,K)$, such that $F_i = \phi_i(P_i)$.

Given a bid price P_i', the monetary cost optimization problem in Formula 14 is equivalent to an optimization problem on the execution time of the MPI application in the circle group. Thus,



(a) m1.small spot instances (b) c3.xlarge spot instances

Figure 4: 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.

we can formulate the checkpoint interval optimization problem as the problem to minimize the execution time of a MPI application in a circle group, given a failure rate function $f_i(P_i',t_i)$. Di et al. [10] have proposed a mathematical method to solve this problem. In this paper, we adopt their approach to solve this problem. We briefly describe the solution here, and more details can be found in their paper [10]. The optimal checkpoint interval can be estimated as:

$$\phi_i(P_i') = \sqrt{\frac{2O_i \cdot T_i}{E_i(P_i')}} \tag{17}$$

where O_i is the checkpoint overhead in circle group i, T_i is the application execution time and $E_i(P_i')$ denotes the expected number of failure events occurring 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.

$$E_i(P_i') = \sum_{t_i=1}^{T_i} f_i(P_i',t_i) \tag{18}$$

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 17, we now simplify the problem in Formula 14 to an optimization problem with a single variable $\vec{P}$ as follow:

$$\begin{aligned} &minimize && E(Cost(\vec{P},\vec{\phi}(\vec{F}))) \\ &subject\ to && E(Time(\vec{P},\vec{\phi}(\vec{F})) \le Deadline \end{aligned} \tag{19}$$

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. In order to reduce the optimization space, we propose a logarithmic searching method.

Our optimization goal is depending on the optimization of two functions, i.e., the failure rate function $f_i(P_i,t_i)$ (see Formula 2) and the expected spot price $S_i(P_i)$ (see Formula 5) for each circle group i ($i = 1,2,\ldots,K$). Figure 4 shows the changing trends of the two functions with the increase of bid price, for spot instances of m1.small and c3.xlarge type in the us-east-1a zone. Both of the two functions are sensitive to the bid price, but not in a uniform way. For example, when the bid price of m1.small spot instances 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 equally divide the space into I ($I = 2^L$) intervals. We only search the j^{th} point in the range, where $j = 2^l$ ($l = 1,2,..,L$). For each searched point $\vec{P_j}$ ($\vec{P_j} = \{P_{1j},P_{2j},\ldots,P_{Kj}\}$), we evaluate it with $Cost(\vec{P},\vec{\phi}(\vec{P}))$ and $Time(\vec{P},\vec{\phi}(\vec{P}))$ in Formula 19. The bid price vector with the minimal expected monetary cost while satisfying the deadline constraint is returned as the optimal solution. With the logarithmic searching method, we reduce the searching space to $O((log_2 H)^K)$.

4.3 Update Maintenance

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

```
1: Select the on-demand instance type d (d ∈ {1,2,...,K_D}) by solving Formula 12;
2: Define the optimization window size T_m;
3: Initialize the execution time T_e ← 0, T_o ← T_d, T ← T_m, D ← D_d, M ← M_d and
   T_i ← T_i × T_m/T_o (i = 1,2,...K);
4: Set the original start point Start (The beginning of the application);
5: while T_o>0 do
6:   if T_o< 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:      Deadline* ← (Deadline − T_e) × T_m/T_o;
14:    else
15:      /*Complete the rest of the application*/
16:      Deadline* ← Deadline − T_e, T ← T_o and T_i ← T_i × T_o/T_m (i = 1,2,...K);
17:    Calculate Distribution function f_i(P_i,t_i) and S_i(P_i) based on Trace i (i =
       1,2,...K);
18:    Compute (P⃗,F⃗) based on two-level optimization algorithm when the
       deadline is Deadline* (Solve Formula 14);
19:    Hybrid run the application based on the configuration from start point
       Start until the time interval T is completed;
20:    T_o ← 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,...K);
23: return
```

Our algorithm needs to adapt to the variances of spot prices, as observed in Section 2. Another reason is that the evaluation of each searched solution has a time complexity of $O(\bar{T}^K)$ ($\bar{T}$ 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 spot price trace with the spot price history 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 1 shows the optimization process of the adaptive method.

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 of the application (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 1). 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 (usually less than 1% of the total execution time).

4.4 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.

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 ($i = 1, 2, ...K$). 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 from a random point in the spot price history of 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 to the counter *count*. We repeat the same process for G times (G is sufficiently large) and finally use $\frac{count}{G}$ as the failing probability 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 on different instance types using TAU (Tuning and Analysis Utilities) [36] with the following profile: $< \#Isr$, $Data_{send}$, $Data_{rev}$, IO_{seq}, $IO_{rnd} >$. $\#Isr$ 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 Open-MPI [14] 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. If the checkpoint is stored in local disk, the data may be lost at any time when the spot instance is terminated. We choose to use Amazon Simple Storage Service (Amazon S3) to store the checkpoint for two reasons. First, Amazon S3 gives any developer access to the same highly reliable and fast data storage infrastructure. 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. EVALUATIONS

This section presents our experimental results on evaluating the proposed approach and model. Overall, there are three groups of experiments. First, we study the impact of different parameters in our model (Section 5.2). Second, we study the monetary cost optimization results for different applications in comparison with the state-of-the-art algorithm [30] and other heuristic approaches of utilizing on-demand and spot instances on Amazon EC2 (Section 5.3). Third, we study the model accuracy and the detailed comparison with the approaches designed by individual fault-tolerance method in simulations (Section 5.4).

5.1 Experimental Setup

Applications. We apply our model to NAS Parallel Benchmarks (NPB [2]) kernels version 2.4. In order to measure the monetary cost of MPI-based applications on spot instances, we choose three types of applications, 1) *computation-intensive applications*, including BT (Block Tri-diagonal solver), SP (Scalar Penta-diagonal solver) and LU (Lower-upper Gauss-Seidel 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) [1]. 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 5.3.1), (2) comparison with simple utilization of on-demand and spot instances (Section 5.3.2) and (3) comparison with separate fault-tolerance techniques (Section 5.4.2).

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.

5.2 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 same 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. Due to the space limitation, we summarize our findings here, and present more details on parameter study in Appendix B of our technical report [18].

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 κ=4 as our default setting.

$\mathbf{T_m}$. 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.

5.3 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 [30]. Second, we evaluate the monetary cost in comparison with the heuristic algorithms of utilizing on-demand and spot 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.

5.3.1 Comparison with state-of-the-art algorithm

In comparison with the state-of-the-art algorithm [30], 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. [30] 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. To the best of our knowledge, 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. The results can be found in our technical report [18]. 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

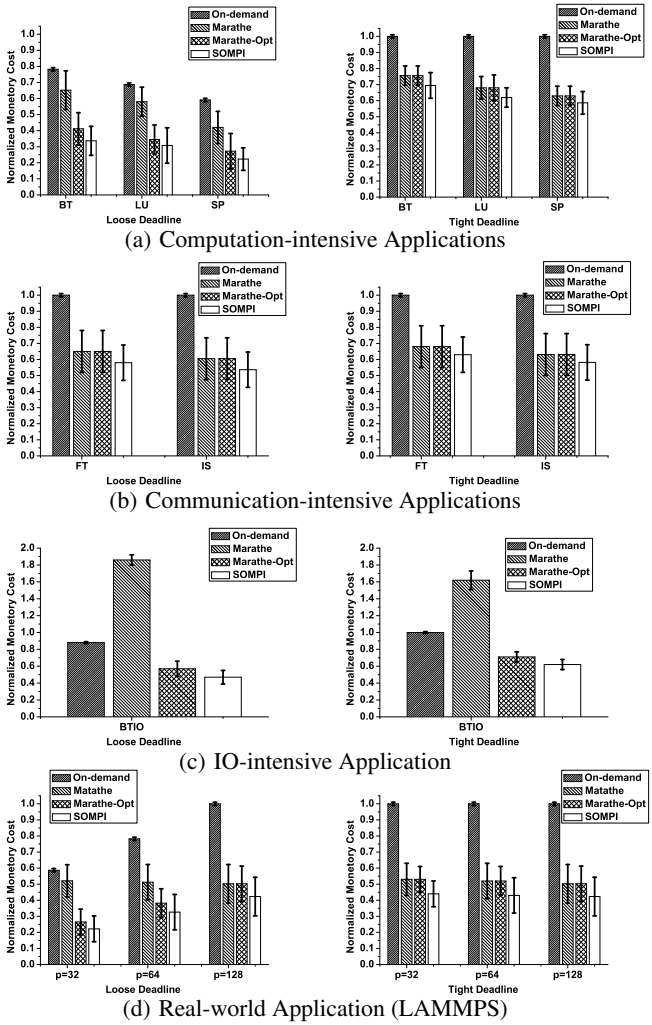


Figure 5: Monetary cost comparison in different deadline

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 proportion 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.

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 2: Normalized Execution time comparison for Marathe-Opt and SOMPI

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

5.3.2 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 5.3.1.
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 6 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.

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.

5.3.3 Detailed study for SOMPI

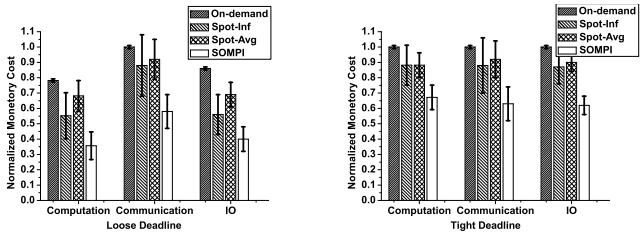


Figure 6: Monetary cost comparison with heuristic algorithms

In this section, we study the relationship between the deadline requirement and the monetary cost. Figure 7 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% off. 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 7(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% off 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% off, compared with Baseline Cost. As the deadline increases to about 10% larger than Baseline Time (denoted by the arrow in Figure 7(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 Results with Simulation

5.4.1 Failure Rate Function and Model Accuracy

Due to the space limitation, we summarize our findings here, and present the details on the accuracy of failure rate function and our model in Appendix B of our technical report [18].

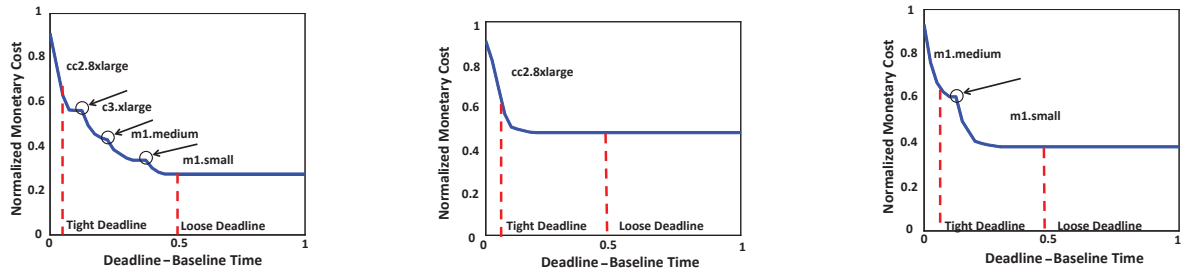
Accuracy of Failure Rate Function. We verify the accuracy of the failure rate function $f(\vec{P},\vec{t})$. In the first step, we randomly choose the history of the spot price in the threshold for four days from the trace. We treat the data of the first 3 days as the training data and calculate the 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 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 5.1. We use the Monte Carlo method to calculate the expected cost and compare with the calculation by Formula 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%.

5.4.2 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 8 shows the evaluation results in loose and tight deadline requirements. We have the following observations. (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 (BT) (b) Communication-intensive (FT) (c) IO-intensive (BTIO)

Figure 7: Monetary cost in different Deadline: BT, FT and BTIO

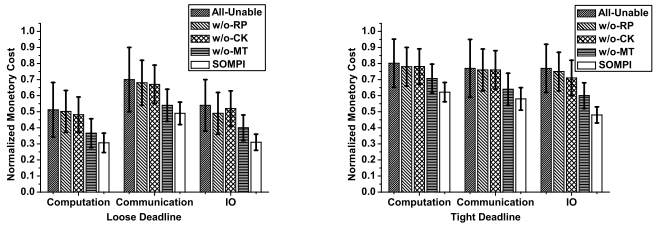


Figure 8: Monetary cost comparison with fault-tolerance mechanisms

6. RELATED WORK

Cost-aware optimizations have generated fruitful research results for decades in various areas such as grid computing [12, 27], databeases [19] and Internet [3]. The pay-as-you-go scheme of cloud computing attracts many interests in optimizing the monetary cost for various applications in the cloud. The resource provisioning problem has been tackled with many different methods such as control theory [49], machine learning [45] and models [35]. Zhan et al. [44] propose a cost-aware cooperative resource provisioning solution for heterogeneous workloads in data centers. For scheduling problem, workflow scheduling with deadline and budget constraints [26, 42, 15, 34] has been widely studied. Yu et al. [42] proposed a cost-based workflow scheduling algorithm that minimizes execution cost while meeting the deadline for delivering results.

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. [37] proposed a bidding strategy to maximize the revenue of cloud brokers on utilizing spot instances for computation. Mazzucco et al. [31] 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. [21] discussed bidding policy of spot instance for highly available services. He et al. [24] proposed to cut the cost of hosting online Services with spot instance. But none of them focuses on MPI-based applications. Taifi et al. [38] introduced a formal model to guide the design of checkpoint for MPI-based applications. But they only considered checkpointing technique. Marathe et al. [30] exploit replicated executions for cost-effective, time-constrained execution of HPC applications on Amazon EC2. It is the most related work with ours. Our work has three major differences from theirs. First, they propose to deploy redundancy in independent availability zones with the same type. In our work, we can exploit different instance types to further reduce monetary cost. Second, we carefully study the setting of the bid price and checkpoint intervals and propose a cost model to show how they affect the monetary cost of execution MPI applications. Third, we study the distribution of spot price and consider how to adapt the replicated execution and checkpoint to the changed distribution on spot prices.

7. CONCLUSION

Amazon EC2 spot instances give us a chance to reduce monetary costs of HPC applications compared with on-demand instances only. In our work, we formulate a monetary cost optimization problem for MPI applications and leverage both spot and on-demand 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 [30] by 20% on average (up to 40%) while satisfying the deadline requirement.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. We acknowledge the support from the Singapore National Research Foundation under its Environmental & Water Technologies Strategic Research Programme and administered by the Environment & Water Industry Programme Office (EWI) of the PUB, under project 1002-IRIS-09. Yifan and Amelie are supported by the scholarship from NEWRI, IGS(Interdisciplinary Graduate School). Bingsheng is in part supported by a MoE AcRF Tier 1 (2014-T1-001-145) of Singapore.

9. REFERENCES

[1] Lammps molecular dynamics simulator. Technical report, http://lammps.sandia.gov/.

[2] The nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[3] M. Adler, J.-Y. Cai, J. K. Shapiro, and D. Towsley. Estimation of congestion price using probabilistic packet marking. In *INFOCOM'03*, 2003.

[4] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM TEAC*, 2013.

[5] Amazon Case Studies. *http://aws.amazon.com/hpc-applications/*.

[6] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under sla constraints. In *MASCOTS'10*, 2010.

[7] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2009.

[8] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC'12*, 2012.

[9] 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.

[10] 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.

[11] 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.

[12] C. Ernemann, V. Hamscher, and R. Yahyapour. Economic scheduling in grid computing. In *JSSPP'02*, 2002.

[13] C. Evangelinos and C. Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on Amazons EC2. In *CCA'08*, 2008.

[14] E. Gabriel and et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *PVM/MPI'04*, 2004.

[15] 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.

[16] Y. Gong, B. He, and D. Li. Finding constant from change: Revisiting network performance aware optimizations on iaas clouds. In *SC'14*, 2014.

[17] Y. Gong, B. He, and J. Zhong. Network performance aware MPI collective communication operations in the cloud. *IEEE TPDS*, 2013.

[18] Y. Gong, A. C. Zhou, and B. He. Monetary cost optimizations for mpi-based hpc applications on amazon clouds: Checkpoints and replicated execution. Technical Report 2015-TR-222, Nanyang Technological University, Singapore, http://www3.ntu.edu.sg/home/bshe/2015-TR-222-SOMPI.pdf.

[19] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM Sigmod Record*, 1997.

[20] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies Ada-Europe'96*, 1996.

[21] 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.

[22] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *JPCS'06*, 2006.

[23] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SoCC'10*, 2010.

[24] X. He, P. Shenoy, R. Sitaraman, and D. Irwin. Cutting the cost of hosting online services using cloud spot markets. In *HPDC'15*, 2015.

[25] 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.

[26] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis. Schedule optimization for data processing flows on the cloud. In *SIGMOD'11*, 2011.

[27] Y. C. Lee, A. Y. Zomaya, and M. Yousif. Reliable workflow execution in distributed systems for cost efficiency. In *GRID'10*, 2010.

[28] W. Lu, J. Jackson, and R. Barga. Azureblast: a case study of developing science applications on the cloud. In *HPDC'10*, 2010.

[29] M. Malawski and et al. Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In *SC'12*, 2012.

[30] 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.

[31] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *HPCC'11*, 2011.

[32] S. Moore, D. Cronk, K. S. London, and J. Dongarra. Review of Performance Analysis Tools for MPI Parallel Programs. In *PVM/MPI'01*, 2001.

[33] S. Núnez, 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'10*, 2010.

[34] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos. Scheduling workflows with budget constraints. In *Integrated Research in GRID Computing'07*. 2007.

[35] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *ICDCS'11*, 2011.

[36] S. S. Shende and A. D. Malony. The tau parallel performance system. *SAGE IJHPCA*, 2006.

[37] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *INFOCOM'12*, 2012.

[38] M. Taifi, J. Y. Shi, and A. Khreishah. Spotmpi: A framework for auction-based HPC computing using Amazon spot instances. In *ICA3PP'11*, 2011.

[39] J. Wu, J. Liu, P. Wyckoff, and D. Panda. Impact of on-demand connection management in mpi over via. In *CLUSTER'02*, 2002.

[40] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE TSC*, 2012.

[41] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *CLOUD'10*, 2010.

[42] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science'05*, 2005.

[43] 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'11*, 2011.

[44] J. Zhan, L. Wang, X. Li, W. Shi, C. Weng, W. Zhang, and X. Zang. Cost-aware cooperative resource provisioning for heterogeneous workloads in data centers. *IEEE TC*, 2013.

[45] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC'07*, 2007.

[46] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *UCC'11*, 2011.

[47] A. C. Zhou and B. He. Transformation-based monetary cost optimizations for workflows in the cloud. *IEEE TCC*, 2014.

[48] 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.

[49] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *HPDC'10*, 2010.