CAP^3: A Cloud Auto-Provisioning Framework for Parallel Processing Using On-demand and Spot Instances

He Huang, Liqiang Wang
Department of Computer Science, University of Wyoming
Laramie, WY
{hhuang1, lwang7}@uwyo.edu

Byung Chul Tak, Long Wang, Chunqiang Tang
IBM T.J. Watson Research Center
Yorktown Heights, NY
{btak, wanglo, ctang}@us.ibm.com

***Abstract*—Cloud computing has drawn increasing attention from the scientific computing community due to its ease of use, elasticity, and relatively low cost. Because a high-performance computing (HPC) application is usually resource demanding, without careful planning, it can incur a high monetary expense even in Cloud. We design a tool called CAP^3 (Cloud Auto-Provisioning framework for Parallel Processing) to help a user minimize the expense of running an HPC application in Cloud, while meeting the user-specified job deadline. Given an HPC application, CAP^3 automatically profiles the application, builds a model to predict its performance, and infers a proper cluster size that can finish the job within its deadline while minimizing the total cost. To further reduce the cost, CAP^3 intelligently chooses the Cloud's reliable on-demand instances or low-cost spot instances, depending on whether the remaining time is tight in meeting the application's deadline. Experiments on Amazon EC2 show that the execution strategy given by CAP^3 is cost-effective, by choosing a proper cluster size and a proper instance type (on-demand or spot).**

***Index Terms*—Cloud computing; provisioning; virtual cluster; parallel scientific application; spot instance**

I. Introduction

Parallel scientific applications require massive computing resources. Traditionally, such applications run on dedicated high performance computing (HPC) clusters located at research institutes or government agencies. The recent advances of Cloud Computing has made computing resources widely available as a utility. Infrastructure-as-a-service (IaaS) Cloud such as Amazon EC2 [1] makes it possible to run HPC applications in a pay-as-you-go fashion.

Many HPC cluster users constantly feel the pain of waiting in a queue for dedicated computing resources to become available. Moreover, it is difficult to have a customized software environment because users have no administrator privilege. Virtual clusters in Cloud solve both problems. A user can easily construct a virtual cluster at any time by renting virtual machines (VMs) from the Cloud provider. The user has full control over the virtual cluster and can configure a customized software environment for his/her application.

In most cases, the performance of a virtual cluster in Cloud is still inferior to a dedicated HPC cluster, due to low network bandwidth and high virtualization overhead. Nevertheless, virtual clusters are sufficient and ideal for many loosely coupled HPC applications. For a university or enterprise department, a virtual cluster is a good alternative to a small physical cluster. For an HPC developer, the ease of use in obtaining a virtual cluster greatly improves productivity. Even for production HPC workloads, a virtual cluster can provide additional capacity when the local dedicated cluster runs out resources.

Cloud providers usually offer various types of VM instances at different prices, with varying compute, network, and storage capabilities. (We simply refer to a VM instance as instance.) Cloud providers also offer different pricing models. *E.g.*, Amazon EC2 provides on-demand, reserved, and spot instances. On-Demand instances charge users for compute capacity by hour without long-term commitment. Reserved instances provide the option of making a one-time discount payment for long-term use. Spot instances allow users to name their own price to bid on spare Amazon EC2 instances. Spot prices are often much cheaper than on-demand prices for the same EC2 instance types. However, the user runs the risk of not getting the compute resources his/she needs, if the bid price is too low. In this paper, we focus on on-demand instances and spot instances, which is preferred by users who do not want to make long-term commitment.

This paper studies how to run HPC applications in Cloud in a cost-effective manner. If customers want to use virtual clusters for the purpose of development and debugging, a small number of low-end instances is often sufficient. If customers want to use virtual cluster for production workloads, both performance and cost are important. However, it is hard for customers to determine the proper number of VMs that can meet the job deadline while minimizing the cost. We refer to this as sizing problem. On the one hand, using a smaller cluster may not finish the job in time. On the other hand, using an unnecessarily large cluster may incur a high cost without proportionally reducing the execution time, because an HPC application usually does not scale beyond a certain cluster size.

In this paper, we propose the CAP^3 Cloud auto-provisioning framework for HPC applications. Given a task specification, CAP^3 automatically determines the proper virtual cluster size

and the instance type (on-demand or spot). The task specification includes the application itself, small datasets for profiling runs, large datasets for production runs, and the user-specified upper bounds of cost and execution time.

We make the following contributions in this paper:

- CAP^3 infers a proper virtual cluster size (i.e., the number of VMs) that minimize the cost while meeting the deadline and budget constraints.
- CAP^3 automatically chooses on-demand instances or spot instances to further reduce cost, depending on whether the remaining time is tight in meeting the applications deadline.
- CAP^3 automates the entire procedure of running HPC applications in Cloud. A customer only needs to provide the task specification, while CAP^3 solves the sizing problem and automatically runs the job.
- We implemente CAP^3 on top of Amazon EC2 and our experiments show promising results.

II. Design of CAP^3

A. CAP^3 Overall Architecture

The end goal of CAP^3 tool is to facilitate the execution of scientific parallel applications in the Cloud by automating the entire process of cluster set-up and running in a cost-efficient way. In order to achieve this goal, we have designed CAP^3, which consists of two modules - *estimation module* and *scheduling module*. The *estimation module* is responsible for generating performance and cost estimates based on the given application specifications submitted by the user. The estimation process includes application profiling, model construction and plan generation. The output specifies the proper VM sizes suggested by the estimation module. Section II-B provides more details. The *scheduling module* takes the output from estimation module as input and plans scheduling based on the desired deadline and budget. In our CAP^3 framework, scheduling implies the decision of whether and how to use the on-demand or spot instances. Then, it carries out the task of actual cluster provisioning in the target Cloud and coordinates the execution of the scientific application. Details of the *scheduling module* is presented in Section II-C.

These overall process of CAP^3 is illustrated in Figure 1, which includes the following steps.

1) A customer first submits task specifications to CAP^3. The task specification contains configuration information for the target application including time limits, cost limit, Cloud storage ID and pointer to the customer's application and datasets.
2) The *estimation module* receives the task specification.
3) A *profiler component* within the *estimation module* prepares a small-sized virtual cluster and performs the sample runs using small training datasets for profiling. Application runtime features are extracted from this profiling run. Application runtime features vary for applications, which are described in more details in Section II-B2.
4) A *model construction component* takes these profiles to build performance models, which are used to predict the execution time of the application for a large dataset on a large-sized cluster.
5) The *planning component* combines Cloud-specific information and the prediction results, and then determines the most appropriate cluster size.
6) The *estimation module* outputs the desired size and estimated execution time as annotated task specification to the *scheduling module*.
7) Based on the results from the *estimation module* and deadline constraints, the *scheduling module* invokes appropriate *agents* who are responsible for interacting with the Cloud and running the application. For example, one agent may primarily focus on handling on-demand instances while the other agents handle spot instances.
8) Once an agent is selected, it creates actual VM instances, sets up the application and data, and runs the task.

B. Estimation Module

The estimation module has three components: *profiler*, *model construction* and *planning*.

1) Profiler Component: We use TAU (Tuning and Analysis Utilities) [2] as a profiling tool in CAP^3. TAU is a portable profiling and tracing toolkit in wide use for performance analysis of parallel scientific applications. It is capable of collecting performance-related information such as function invocation statistics, communication patterns and aggregated node/thread specific information via instrumentation.

The *profiler module* in CAP^3 utilizes TAU in the following way. Profiler first generates a script for application profiling based on the customer's task specification. According to the script, TAU automatically instruments the application code prior to the profiling run. Then, the profiling run is performed using small sample datasets on small size of virtual clusters. During profiling run, TAU-instrumented codes gather performance-related information, and the *profiler* uses the parser to analyze and extract mean time of each function and loop in the application code. Then, it classifies these information into communication time (*i.e.*, time spent on MPI function calls) and computation time (*i.e.*, time spent on code section excluding communication). Such information serves as inputs to the *model construction component*.

2) Model Construction Component: We apply linear regression (LR) technique to predict the execution time for the given application. We use two input data for the LR: CPU time spent for computation and network time from MPI communication. This is based on our assumption that the total execution time of scientific applications are the sequence of CPU time and delays from network communications.

CPU computation time is expressed as:

$$T_C = c_0 + \sum_{j=1}^{r}(c_j \cdot x_j) + c_{(r+1)} \cdot P \qquad (1)$$

where T_C is the execution time, c_j is the coefficients, P is the number of MPI processes, x_j is the application-specific feature

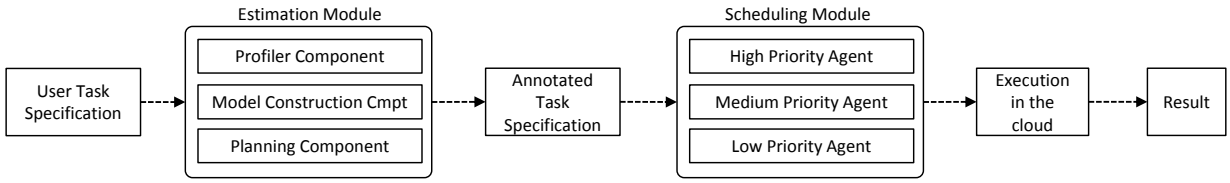

Fig. 1. The workflow of CAP^3.

Benchmark	Feature Variables
CG	no. of rows, no of nonzeros no. of iterations, eigenvalue shift
EP	no of random-number pairs
FT	grid size: $x \times y \times z$, no. of iterations
IS	no. of keys, key max value
MG	grid size: $x \times y \times z$, no. of iterations
BT	write interval, Gbytes written
LU	grid size: $x \times y \times z$, no. of iterations, time step
SP	grid size: $x \times y \times z$, no. of iterations, time step

TABLE I
FEATURE VARIABLES FOR DIFFERENT APPLICATIONS. THE NUMBER OF PROCESSES IS A FEATURE COMMON TO ALL APPLICATIONS, AND HENCE OMITTED IN THIS TABLE.

variables and r is the number of feature variables. P and x_j are independent variables, and T_C is a dependent variable. The number of application-specific feature variables are different from application to application. Table I describes the features we have used for some benchmarks. Following [3], we take the logarithm of Equation 1 to increase the accuracy as follows.

$$\log T_C = c_0 + \sum_{j=1}^{r}(c_j \cdot \log x_j) + c_{(r+1)} \cdot \log P \qquad (2)$$

To construct the LR model for the network delay, we assume that the MPI collective communication cost is proportional to either $p \cdot log(p)$ or $log(p)$ as reported in [4]. They also specify which proportion should be used for each of the MPI function. We adopt their results. We substitute P in Equation 2 with term $p \cdot log(p)$ or $log(p)$ and yield Equation 3 or 4.

$$\log T_N = c_0 + \sum_{j=1}^{r}(c_j \cdot \log x_j) + c_{(r+1)} \cdot \log(\log P) \qquad (3)$$

$$\log T_N = c_0 + \sum_{j=1}^{r}(c_j \cdot \log x_j) + c_{(r+1)} \cdot \log(P \cdot \log P) \qquad (4)$$

where T_N denotes the network delay caused by the MPI function calls during the execution.

Using small dataset, the *profiler component* generates series of training data $(T'_C, P', x'_1, x'_2, x'_3, ...)$ for the CPU execution time and $(T'_N, P', x'_1, x'_2, x'_3, ...)$ for the network delay due to MPI function calls. They are plugged into the appropriate equations among Equation 2, 3 or 4 and the least square method is used to determine the model parameter c_j.

3) Planning Component: The planning component is responsible for computing approximate cost and assist to determine the number of instances used for real execution.

The cost of executing a task is determined by the number of instances, unit price of the instance type, and execution time, which is expressed in Equation 5.

$$cost_{task} = c_{unit} \times \lceil \frac{num\_proc}{ppn} \rceil \times \lceil T_C + T_N \rceil \qquad (5)$$

where c_{unit} is the unit price of the type of instance used for execution, num_proc is the number of MPI processes, ppn is the number of processes per instance and $\frac{num\_proc}{ppn}$ is the number of instances used for the task.

Given the deadline and budget, the planning component iterates over the predefined range of num_proc, and calls the *model construction component* and Equation 5 to find out the estimated time and cost. Then it chooses num_proc with the minimal time that satisfies both $time_{limit}$ and $cost_{limit}$. Finally the planning component annotates optimal num_proc and its corresponding estimated time into the task specification, which is used as input for the *scheduling module*.

C. Scheduling Module

The *scheduling module* takes an annotated task specification as input, and schedules the task to appropriate virtual clusters to run. The module tries to complete a task within the deadline under the budget. We first introduce the task priority rate (PR) as in Equation 6,

$$PR = \frac{T_{deadline} - T_{now} - T_{est} - T_{latency}}{T_{est}} \qquad (6)$$

where $T_{deadline}$ is the deadline specified by customer, T_{now} is the current wall clock time, T_{est} is the estimated time running on certain number of instances suggested by the estimation module, $T_{latency}$ is the latency to set up instances (*e.g.*, initiate instance, submit job to the queue, and tear-down instance). PR describes the priority of a task. If PR is negative, the task cannot meet the deadline. When PR tends to be a small positive number, the task can be finished before deadline, but the remaining time is very limited. So the task is considered to be in high priority. When PR tends to be a large positive number, the task is in lower priority, and the remaining time for the task is sufficient. The scheduling module first attempts to run the lower priority task on some low cost instances at the risk of failure. Note that PR is changing as the time elapses. A lower priority task can become a higher priority task if the task

Agent	Cluster	PR range	Explanation
framework agent AG	N/A	all	update and check task's PR, distribute tasks to appropriate cluster agent
ag_{hi}	high priority on-demand instance cluster	$PR < th_1$	run high priority task on on-demand instances
ag_{me}	medium priority spot instance cluster	$th_1 \leq PR \leq th_2$	run medium priority tasks on spot instances, bid at relatively high price
ag_{lo}	low priority spot instance cluster	$PR > th_2$	run tasks with low priority on spot instances, bid relatively low price

TABLE II
DIFFERENT AGENTS IN THE AUTO-SCHEDULING MODULE. th_1 AND th_2 ARE THRESHOLD VALUES, AND $th_1 < th_2$.

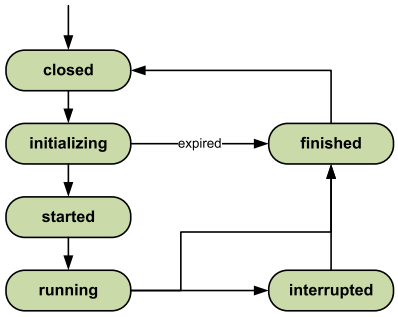

Fig. 2. Cluster state transition diagram.

does not make much progress in lower priority phase along time elapses.

We maintain one framework agent AG and three cluster agents: high priority agent ag_{hi}, medium priority agent ag_{me} and low priority agent ag_{lo} that correspond to three different virtual clusters described in Table II.

- The framework AG takes all kinds of tasks as input. It does not manage a virtual cluster directly. Instead, it works as a coordinator for assigning tasks to appropriate agents and for receiving expired task notification from cluster agents.
- High priority agent ag_{hi} accepts high priority tasks and maintains an on-demand cluster. All the instances in ag_{hi}'s virtual cluster are from Amazon EC2 on-demand instances. ag_{hi} offers the highest price with the most reliable service. ag_{hi} is used for executing emergent tasks with high priority (PR is very small) at relatively higher cost. ag_{hi} only accepts emergent tasks with $PR < th_1$. Thresholds are set by administrator based on prior experience. In the experiment, we set $th_1 = 1.0$, which indicates that the task can only be run at most once with the current remaining time. When ag_{hi} accepts an emergent task, it applies certain number of on-demand instances according to num_proc in annotated task specification. The high priority task is scheduled to run immediately when on-demand instance cluster is ready.
- Medium priority agent ag_{me} is in charge of maintaining medium priority spot instance. ag_{me} bids relatively high compared with ag_{lo}. The bidding policy can use sophisticated strategies as discussed in Section IV. In the experiment, our bid price is set based on the current spot price, average historical spot price during the past 30 days and on-demand price. ag_{me} is intended to run medium priority task that is going to become high priority in the near future but still take advantage of spot instance's low price. ag_{me} accepts tasks with PR range from th_1 to th_2. We set $th_2 = 2.0$ in the experiment. A task with medium priority within thresholds 1.0 and 2.0 indicates that it can run twice before the deadline. On the one hand, bidding at higher price improves the reliability because it reduces the possibility of interruption by Amazon when spot price rises. On the other hand it enjoys low spot price most of the time. In case a task is interrupted, it is sent back to AG to redeliver.
- Low priority agent ag_{lo} maintains low priority spot instance cluster. ag_{lo} provides the lowest price but also with the lowest reliability. This agent is intended to run low priority tasks with sufficient remaining time at the lowest cost. ag_{lo} accepts tasks with $PR > th_2$ where $th_2 = 2.0$. It indicates that there are enough remaining time to run at least three times, implying that the task has multiple chances to try low price spot instances at the risk of interruption. To take advantage of spot price, ag_{lo} bids at a very low price. It is possible that a task may be interrupted during execution by Amazon, and sent back to AG. Since the remaining time of the task is still abundant, AG can either reschedule the task to run in ag_{lo} next time or deliver to other higher priority agents.

We next describe the behaviors of these agents. We refer *task* to working unit in the agent, and refer *job* to task submitted to cluster's queuing system, *e.g.*, PBS or Sun Grid Engine.

Framework agent AG stays in an infinite loop to accept incoming tasks. AG initially takes all tasks into its list, and updates each task's PR. Then it compares a task's PR with predefined threshold values, and delivers the task to an appropriate cluster agent.

Algorithm 1 describes the common behaviors of cluster agent. Figure 2 shows cluster's state transition when agent is working. When an agent receives an assigned task, the agent becomes active and its corresponding virtual cluster is set to be closed initially. The agent first checks spot history price and determines the bid price if the agent is a medium or low priority agent (line 3 to 6). Then the agent starts the cluster and sets its state to be initializing (line 7 to 8). For on-demand instances, the agent waits until cluster state becomes started. For spot instance, the agent has to wait until spot price drops below the bid price. The wait could be potentially very long, and if the current task's priority does not satisfy during the wait, the agent sends the task back to framework agent to be redelivered to higher priority agent. The above behaviors are

Algorithm 1 The algorithm executed by cluster agent ag. ag can be any one of $ag_{hi}, ag_{me}, and ag_{lo}$

```
1: a new task t is coming
2: set cluster state c_st = closed
3: if ag is ag_me or ag_lo then
4:     check spot price history
5:     determine bid price
6: end if
7: start a new cluster, setup master node
8: set c_st = initializing
9: while TRUE do
10:    if c_st == initializing then
11:        update t.PR
12:        if cluster is ready then set c_st = started
13:        else if t.PR does not satisfy current ag then   ▷ handle PR expired
14:            send task t to AG
15:            set c_st = finished
16:        else wait an interval
17:        end if
18:    else if c_st == started then
19:        add new instances to cluster as compute nodes
20:        submit t as a job to cluster queuing system
21:        set c_st = running
22:    else if c_st == running then
23:        if (ag is ag_me or ag_lo) and cluster is interrupted then
24:            set t.state = false
25:            set c_st = interrupted
26:        else if job is done then
27:            set t.state = true
28:            set c_st = finished
29:        else wait an interval
30:        end if
31:    else if c_st == interrupted then        ▷ handle interruption
32:        send task t to AG
33:        set c_st = finished
34:    else if c_st == finished then
35:        remove idle instances
36:        terminate cluster
37:        set c_st = closed
38:    else if c_st == closed then break
39:    end if
40: end while
```

described in line 10 to 17. Once cluster gets started (line 18 to 21), the agent adds new instances to cluster according to the number of MPI processes required by the task, and then submits the task as a job to the cluster's scheduler. Cluster state is set to running after these actions. When the job is running, the agent checks its state periodically (line 23 to 30). If the job is done, cluster state is set to finished. For spot instances, if out of bid happens during job running, cluster state is set to interrupted. When cluster is in interrupted state (line 31 to 33), the job's corresponding task is sent back to framework agent for rescheduling, and cluster is set to finished. Finally, if cluster is in finished state, the agent terminates the cluster and becomes idle again (line 34 to 37).

III. Evaluation

We evaluated CAP^3 on Amazon EC2. We chose StarCluster 0.93.3 [5] to configure and manage virtual cluster on Amazon EC2. CAP^3's application profiler invoked TAU 2.22 to extract performance information of applications. CAP^3 was programmed in Python 2.7 with NumPy 1.6.2 library. We used eight MPI C or MPI Fortran benchmarks from NAS Parallel Benchmarks (NPB) 3.3.1. These benchmarks represent a wide spectrum of parallel scientific applications. The eight benchmarks are:

- CG: conjugate gradient, irregular memory access and communication
- FT: discrete 3D FFTs, all-to-all communication
- MG: multi-grid on a sequence of meshes, long- and short-distance communication, memory intensive
- EP: embarrassingly parallel
- IS: integer sort, random memory access
- BT: block tri-diagonal solver
- SP: scalar penta-diagonal solver
- LU: lower-upper Gauss Seidel [6]

Instance Type Name	Memory	CU	PPN	On-demand Price	Price Per CU
small (m1.small)	1.7 GB	1	1	$0.06	$0.06
medium (m1.medium)	3.75 GB	2	1	$0.12	$0.06
large (m1.large)	7.5 GB	4	2	$0.24	$0.06
cluster (cc1.4xlarge)	23 GB	33.5	16	$1.30	$0.039

TABLE III
CONFIGURATIONS AND PRICES OF DIFFERENT AMAZON INSTANCE TYPES. CU: COMPUTE UNIT DEFINED BY AMAZON. PPN: THE NUMBER OF MPI PROCESSES RUNNING ON EACH INSTANCE.

Amazon EC2 provides dozens of optional instance types classified by different compute capacities or usage purposes. Table III lists attributes of four typical instance types: small, medium, and large instance from the 1st generation standard instance that are the most commonly used instance types, and quadruple extra large cluster compute instance that is designed for high performance computing with better IO and network. Compute Unit (CU) is Amazon defined metric to measure compute capability of VM. Larger CU indicates higher compute capability. The last column shows price per CU for each type. According to Table III, cluster instance is cheaper than other standard instances in terms of price per CU. Hence, we recommend to run parallel scientific applications using cluster instance type as it is the most efficient type.

We verified our recommendation by comparing the costs of different instance type runs with 25 or 32 [1]MPI processes for different tasks shown in Figure 3. Only IS showed benefit using medium instance. For the other seven tasks, using cluster is no worse than other standard types. If we run more processes, the advantage of using cluster instance is more obvious because of faster interconnection between cluster instances.

[1]Some applications require number of processes must be square number, some applications require power of two, and some applications accept arbitrary number.

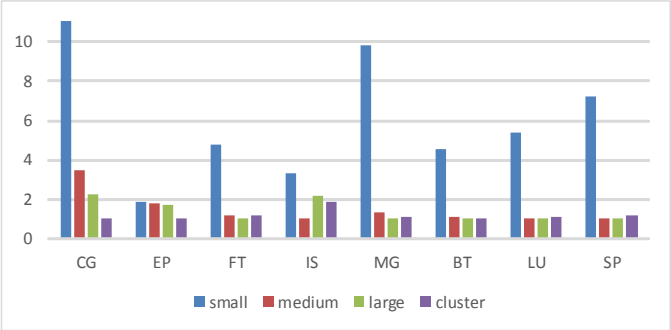

Fig. 3. Cost comparison using different instance types for different tasks. Each task runs 25 or 32 MPI processes. The y-axis shows the ratio of the real cost of a task using a specific instance type to the minimal cost of the task among all instance types.

App	CAP^3 #proc	customer #proc	priority	instance type
CG	16	64	medium	cluster
EP	96	48	high	cluster
FT	16	32	high	cluster
IS	16	32	high	medium
MG	16	32	low	cluster
BT	16	25	low	cluster
LU	128	64	low	cluster
SP	16	25	medium	cluster

TABLE IV
COMPARISON OF THE CONFIGURATION PROVIDED BY CAP^3 AND THAT PROVIDED BY A COST-IGNORANT CUSTOMER.

	CG	EP	FT	IS	MG	BT	LU	SP	sum
CAP^3 dl	✓	✓	✓	✓	✓	✓	✓	✓	8
Cus dl	✓		✓	✓	✓	✓	✓	✓	7
CAP^3 bd	✓		✓	✓	✓	✓		✓	6
Cus bd		✓	✓	✓	✓		✓		5

TABLE V
DEADLINE AND BUDGET SATISFACTION. A CHECK MARK MEANS THE TASK MEETS THE DEADLINE OR BUDGET. CAP^3 DL: CAP^3 DEADLINE. CUS DL: CUSTOMER DEADLINE. CAP^3 BD: CAP^3 BUDGET. CUS BD: CUSTOMER BUDGET. LAST COLUMN INDICATES TOTAL NUMBER OF TASKS THAT MEET DEADLINE OR BUDGET FOR CAP^3 OR CUSTOMER.

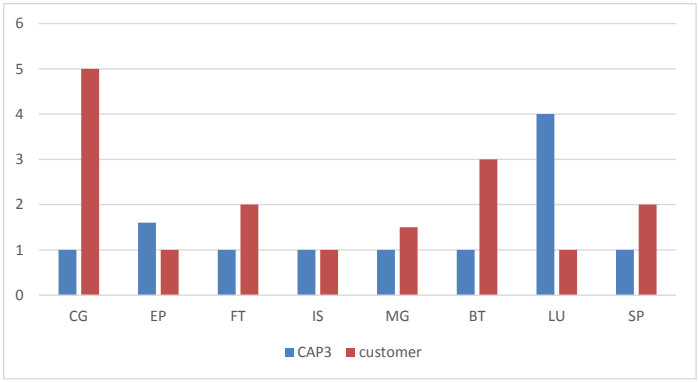

Fig. 4. Cost comparison between CAP^3-provided cluster size and customer-provided cluster size. The y-axis is the ratio of the actual cost to the minimal cost of CAP^3 and customer .

A. Constraints Satisfaction and Cost Comparison of On-demand Instance

CAP^3 first ran the estimation module to give appropriate cluster size used for executing large dataset. The estimation module ran small training dataset for each task. The course of training each task is within one hour. Note that the training only needs to be run once. After the first run, the training dataset was saved and can be reused for future usage. In this experiment, each task specification provided 4 or 5 small training dataset. The estimation module profiled each dataset from 2 to up to 32 processes. Then the estimation module estimated the proper size of each task listed in the second column of Table IV. The third column is the size used by cost-ignorant customer (referred as customer for short) as comparison. From the table, the estimation module tended to give a smaller size (16 processes) for 6 out of 8 tasks, while the customer tended to use a reasonable large size based on his/her prior experiences on HPC clusters. Many of parallel scientific applications require intensive communication. Smaller size provided by CAP^3 indicated lower communications between instances made applications' scalability different from HPC cluster.

Next, CAP^3's scheduling module applied for certain number of instances according to the size provided by the estimation module in the second column of Table IV, and ran tasks with large dataset at virtual cluster. We first checked if CAP^3 can meet deadline and budget and compared it with a cost-ignorant customer in Table V. CAP^3 met all deadline, while the customer missed one deadline. Six out of eight tasks using CAP^3 were within budget, while five were within budget for customer. CAP^3 can meet deadline and budget in most cases, and is better than the customer.

Then we compared the actual cost using the sizes provided by CAP^3 and customer in Figure 4. To simplify cost comparison, we calculate cost based on on-demand instance price. Five (CG, FT, MG, BT, SP) of tasks using CAP^3 costed less than the customers solution, one (IS) task costed the same, and two (EP, LU) tasks costed more. On average, CAP^3's solution costs less than the customer's solution.

We further investigated LU, which used a fairly large size provided by the tool and costed three times more expensive than using customer provided size. The reason behind the false size was that, the tool only ran up to 32 processes for profiling in the estimation module. LU was very scalable from 2 to 32 processes. So the tool gave a fairly large size to run large dataset. However, by doing expanded experiments, we found LU is not scalable at larger number of processes. We can solve this problem by running training dataset with larger number of processes at the expense of higher cost and longer time. The tool should learn that LU is not scalable after certain number of processes and therefore give a reasonable smaller size.

B. Cost Comparison of Spot Instance

One of the features of CAP^3 is that it can intelligently choose between on-demand instance and spot instance for

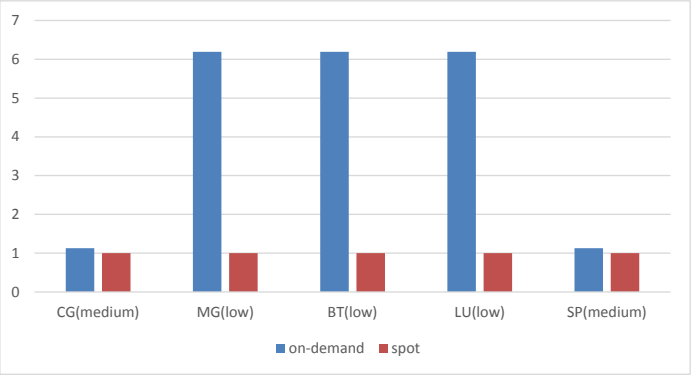

Fig. 5. Cost comparison between on-demand instance and spot instance. The symbols (medium) and (low) represent different cluster agents. The y-axis is the ratio of the actual cost to the minimal cost of on-demand and spot.

each task according to remaining time. In the 4th column of Table IV, CG and SP used medium priority agent to bid at higher price for spot instance, and MG, BT, LU used low priority agent to bid at lower price for spot instances. The other three tasks (EP, FT and IS) used on-demand instances. The on-demand price for cluster instance is $1.30. During the experiment, the tool bided at $1.15 for the medium agent, and bided at $0.21 for low priority agent. Figure 5 compared cost between on-demand and spot instances for tasks that used medium or low priority agent. The cost was computed by assuming that spot instances were charged at bid price. In fact, spot instances were charged at real-time spot price, which is lower than bid price if spot instances are fulfilled. So the actual cost was equal or lower than the cost in the figure. Thus, by automatically choosing spot instance, the cost was further reduced significantly.

The risk of using spot instance is the potential interruption during execution. We did not apply any recovering mechanism to handle the runtime failure. We simply deliver the failed task to framework agent, and framework agent will reassign the failed task to appropriate cluster agent. The low priority agent faces the higher risk of interruption because of low bid price. *E.g.*, during the first run in experiment, BT was interrupted after running for one hour twenty minutes. BT was redelivered by freamwork agent to low priority agent to rerun the task. At the second time, BT was lucky and ran for four hours until finished. According to Amazon's spot instance policy, it does not charge the last partial hour interrupted due to rising spot price. So for this task, Amazon total charged for five hours. The total cost was $1.05 ($0.21 \times 5$), but was still much lower than on demand cost which was $5.2 ($1.30 \times 4$). In most cases, task in low priority agent has enough time to run multiple times. Such task usually bids less than 30% of on-demand price. So it is still less expensive than using on-demand price if interruption happens. In some rare cases, using spot instances could be more expensive than on-demand instances. Either because spot price keeps in high price or there is not too much remaining time. An interrupted task may be delayed till it has to be delivered to high priority agent using on-demand price. Statistically, CAP^3 is less expensive than solely using on-demand instances.

Compared with using pure on-demand instance, CAP^3 can save significant cost by using spot instances. Compared with using pure spot instance, CAP^3 is more reliable, because some high priority tasks (EP, FT and IS) using spot instance could potentially miss the deadline if interruption happens.

IV. Related Work

In HPC community, significant effort has been put in developing approaches for performance modeling and prediction of application running on traditional clusters. Approaches of performance modeling or prediction fall into three categories: analytics based, simulation based and regression based.

Analytical models such as LogP [7], LogGP [8], PLogP [9], and LoPC [10] use mathematical methods to characterize behaviors of application during execution. Analytical performance models can capture high level of performance or scalability trend, but are intrinsically difficult to predict the application running time accurately.

Simulation based modeling relies on either profiling tools such as TAU [2] or tracing tools such as Vampir [11] and Paraver [12]. [13] gives a comprehensive survey of HPC performance modelling and prediction tools. Dimemas [14] performs instruction based simulation by replaying of trace obtained by Paraver during runtime. Estimation of different machine sizes requires rerunning the program to generate new traces. PSINS [15] collects event traces during application execution (PSINS Tracer), and simulates event traces (PSINS Simulator) on target HPC systems. WARPP [16] performs post-execution analysis on trace-based profiles that combines running capture of call-graph with computation timings.

Regression based approaches correlate parallel application execution time with various input parameters. In [3], several program executions were used on a small subset of the processors to predict execution time on larger number of processors. An improved focused regression approach is proposed in [17] to study time-constrained scaling of scientific applications.

Since spot instance is the first Cloud service based on supply and demand in the market, it draws significant research interest during the last three years. From customer's perspective, a good bidding strategy is necessary in order to save cost and satisfy application requirement. Sophisticated bidding strategies has been proposed by various researchers. Ben-Yehuda et al. [18] construct a model that generates consistent price with spot price by analyzing spot price history of Amazon. Tang et al. [19] model bidding problem as Constrained Markov Decision Process (CMDP). Their "AMAZING" tool can intelligently adapt the bid from detected state pattern. Zafer et al. [20] study dynamic bidding policy for spot instances in the context of deadline constrained jobs. Mattess et al. [21] investigate the strategies of extending local cluster using spot instances in an economic way when peak load coming.

Spot instances are suitable for MapReduce because of its massive scalability and fault tolerance. In [22], spot instances are used as accelerator to reduce runtime of MapReduce

jobs. New technique is devised to handle the adverse effect of interruption. Kambatla et al. [23] improve MapReduce provisioning by taking resource consumption statistics of jobs into account.

Unlike MapReduce applications, parallel scientific applications usually require lots of inter-task communication. Also, it is difficult to scale up and down during execution as MapReduce. In addition, using fault tolerance in such applications is costly. Thus, it is more challenging to apply spot instances to parallel scientific applications. Voorsluys et al. [24] propose a resources allocation policy for running deadline constrained compute-intensive jobs on spot instances based on job runtime estimation. Their later research employs fault tolerance such as checkpointing, task duplication, and migration [25]. SpotMPI [26] facilitates execution of MPI applications on auction-based Cloud platforms based on adjusted optimal checkpoint-restart (CPR) intervals. Their tookit can automate checkpointing at bidding price and restart application after interruption.

V. Conclusion and Future Work

We proposed CAP^3, an auto-provisioning framework for HPC applications in Cloud. CAP^3 helps a Cloud user to apply for instances and run applications without the users interference. It reduces the monetary expense by automatically choosing the proper number of instances and the right instance type (on-demand or spot). Experiments showed that CAP^3 was effective on Amazon EC2.

For future work, CAP^3 can be improved along multiple directions. It can benefit from a more sophisticated and more accurate performance estimation module, as the current model is relatively simple. It can also intergarte sophisticated bidding strategies to save cost, and recovering mechnisum to handle runtime interruption. Another limitation is that the current scheduling module only supports one task per customer. We plan to extend it to handle multiple tasks for multiple customers in the future.

Acknowledgment

He Huang and Liqiang Wang were supported in part by NSF under Grants 0941735 and CAREER-1054834. Liqiang Wang did this work when he was a visiting scientist at the IBM T.J. Watson Research Center.

References

[1] "Amazon EC2," Feb. 2013, http://aws.amazon.com/ec2.

[2] S. Shende and A. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[3] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 368–377.

[4] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[5] "Starcluster," Feb. 2013, http://star.mit.edu/cluster.

[6] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas, "Performance evaluation of Amazon EC2 for NASA HPC applications," *Proceedings of the 3rd workshop on Scientific Cloud Computing Date - ScienceCloud '12*, p. 41, 2012.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, *LogP: Towards a realistic model of parallel computation*. ACM, 1993, vol. 28, no. 7.

[8] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, "Loggp: incorporating long messages into the logp model-one step closer towards a realistic model for parallel computation," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. ACM, 1995, pp. 95–105.

[9] T. Kielmann, H. Bal, and K. Verstoep, "Fast measurement of logp parameters for message passing platforms," *Parallel and Distributed Processing*, pp. 1176–1183, 2000.

[10] M. Frank, A. Agarwal, and M. Vernon, *LoPC: modeling contention in parallel algorithms*. ACM, 1997, vol. 32, no. 7.

[11] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel, "The vampir performance analysis tool-set," *Tools for High Performance Computing*, pp. 139–155, 2008.

[12] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *WoTUG-18*, pp. 17–31, 1995.

[13] R. Allan and A. Mills, *Survey of HPC Performance Modelling and Prediction Tools*. Science and Technology Facilities Council, 2010.

[14] S. Girona, J. Labarta, and R. Badia, "Validation of dimemas communication model for mpi collective operations," *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 39–46, 2000.

[15] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely, "PSINS : An Open Source Event Tracer and Execution Simulator for MPI Applications," pp. 135–148, 2009.

[16] S. D. Hammond, G. R. Mudalige, J. Smith, S. A. Jarvis, J. Herdman, and A. Vadgama, "Warpp: a toolkit for simulating high-performance parallel scientific codes," in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009, p. 19.

[17] B. Barnes, J. Garren, D. K. Lowenthal, J. Reeves, B. R. de Supinski, M. Schulz, and B. Rountree, "Using focused regression for accurate time-constrained scaling of scientific applications," *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, 2010.

[18] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 Spot Instance Pricing," *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pp. 304–311, Nov. 2011.

[19] S. Tang, J. Yuan, and X.-Y. Li, "Towards Optimal Bidding Strategy for Amazon EC2 Cloud Spot Instance," *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 91–98, Jun. 2012.

[20] M. Zafer, Y. Song, and K.-W. Lee, "Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs," *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 75–82, Jun. 2012.

[21] M. Mattess, C. Vecchiola, and R. Buyya, "Managing Peak Loads by Leasing Cloud Infrastructure Services from a Spot Market," *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pp. 180–188, Sep. 2010.

[22] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz, "See spot run: using spot instances for mapreduce workflows," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2010, pp. 7–7.

[23] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *Proc. of the First Workshop on Hot Topics in Cloud Computing*, 2009.

[24] W. Voorsluys, S. Garg, and R. Buyya, "Provisioning spot market cloud resources to create cost-effective virtual clusters," *Algorithms and Architectures for Parallel Processing*, pp. 395–408, 2011.

[25] W. Voorsluys and R. Buyya, "Reliable provisioning of spot instances for compute-intensive applications," in *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*. IEEE, 2012, pp. 542–549.

[26] M. Taifi, J. Shi, and A. Khreishah, "Spotmpi: a framework for auction-based hpc computing using amazon spot instances," *Algorithms and Architectures for Parallel Processing*, pp. 109–120, 2011.