Scientific Programming 19 (2011) 75–89
DOI 10.3233/SPR-2011-0319
IOS Press

Autonomic management of application workflows on hybrid computing infrastructure

Hyunjoo Kim [a], Yaakoub el-Khamra [b,c,*], Ivan Rodero [a], Shantenu Jha [a,d,e] and Manish Parashar [a]
[a] *NSF Center for Autonomic Computing, Department of Electrical and Computer Engineering, Rutgers University, Piscataway, NJ, USA*
[b] *Texas Advanced Computing Center, The University of Texas at Austin, Austin, TX, USA*
[c] *Craft and Hawkins Department of Petroleum Engineering, Louisiana State University, Baton Rouge, LA, USA*
[d] *Center for Computation and Technology, Louisiana State University, Baton Rouge, LA, USA*
[e] *Department of Computer Science, Louisiana State University, Baton Rouge, LA, USA*

Abstract. In this paper, we present a programming and runtime framework that enables the autonomic management of complex application workflows on hybrid computing infrastructures. The framework is designed to address system and application heterogeneity and dynamics to ensure that application objectives and constraints are satisfied. The need for such autonomic system and application management is becoming critical as computing infrastructures become increasingly heterogeneous, integrating different classes of resources from high-end HPC systems to commodity clusters and clouds. For example, the framework presented in this paper can be used to provision the appropriate mix of resources based on application requirements and constraints. The framework also monitors the system/application state and adapts the application and/or resources to respond to changing requirements or environment. To demonstrate the operation of the framework and to evaluate its ability, we employ a workflow used to characterize an oil reservoir executing on a hybrid infrastructure composed of TeraGrid nodes and Amazon EC2 instances of various types. Specifically, we show how different applications objectives such as acceleration, conservation and resilience can be effectively achieved while satisfying deadline and budget constraints, using an appropriate mix of dynamically provisioned resources. Our evaluations also demonstrate that public clouds can be used to complement and reinforce the scheduling and usage of traditional high performance computing infrastructure.

Keywords: Cloud computing, hybrid computing, distributed computing

1. Introduction

The computing infrastructure available to applications is becoming increasingly heterogeneous, integrating high-performance grids with computing clusters and private/public clouds [8,13,27,34]. While such a hybrid infrastructure can support new and potentially more effective usage modes and enable new application formulations, its inherent heterogeneity presents significant challenges. For example, the different resource classes available in such a hybrid infrastructure can vary significantly in their costs for usage, performance, availability and the guarantees for quality of service (QoS) they provide. High Performance Computing (HPC) grids such as TeraGrid provide high-end computing capabilities to authorized users through merit based allocations instead of direct costs, but the amount and duration of usage can be limited and jobs submitted to these resources can experience long queuing times. On the other hand, public clouds such as Amazon EC2 provide on-demand resource availability with a pay-as-you-go pricing model; however, its resources are not as powerful. Commodity clusters that are often locally available typically lie in between these two, both in cost and performance.

On the resource side we are faced with large variations in system capabilities, costs, configurations and availability. On the application side we are faced with dynamic requirements and constraints. Therefore, provisioning an appropriate mix of resources for applications is non-trivial. For example, an user may require results as soon as possible (or by a speci-

*Corresponding author. E-mail: yye00@tacc.utexas.edu.

fied deadline) irrespective of costs, or may provide constraints on resources used due to limited budgets or resource allocations. Furthermore, these requirements/constraints may change, due to, for example, a failure or an application event. The above challenges are further compounded due to these dynamic system behaviors, changing application requirements, and the need for dynamic system and/or application adaptations. In fact, manually monitoring these dynamic behaviors and requirements and enforcing adaptation can quickly become unfeasible, and autonomic management approaches become attractive.

In this paper, we present a programming and runtime framework that enables the autonomic management of complex applications workflows on hybrid computing infrastructures. The framework is designed to address system and application heterogeneity as well as dynamics. It is also designed to ensure that application objectives and constraints are satisfied. Specifically, the autonomic management framework can provision the appropriate mix of HPC grid and public/private cloud resources based on application requirements and constraints, monitor system/application state (e.g., workload, availability, delays) and adapt the application and/or the resources (e.g., change algorithms used or re-provision resources) to respond to changing applications requirements or system state.

Furthermore, to demonstrate the operation of the framework and to evaluate its ability, we employ a workflow used to characterize an oil reservoir executing on a hybrid infrastructure composed of TeraGrid nodes and Amazon EC2 instances of various types. The application workflow performs history matching using an ensemble of reservoir simulations and an ensemble Kalman filter application for analysis [12]. Specifically, we show how different application objectives can be effectively achieved while satisfying deadline and budget constraints, using an appropriate mix of the dynamically provisioned resources.

We consider three types of objectives in this paper: (1) *acceleration* of application Time-To-Completion (TTC) using hybrid resources, (2) *conservation* of HPC resources by limiting the usage of CPU cores or CPU time when using public clouds, and (3) *resilience* to resource failure and unexpected delays. We consider two classes of constraints imposed by the user: (1) *deadlines* by which application tasks must be completed, and (2) *budgets* which must not be violated while executing the task. Note that these objectives and constraints can be combined resulting in complex management requirements for an application workflow. For example, an application may have strict budget limits and may request the maximum acceleration possible, as long this budget is not violated. Requirements may similarly combine acceleration and resilience or budgets and deadlines.

We have implemented and deployed the autonomic management framework on top of the CometCloud [21] autonomic cloud federation engine, which supports autonomic cloud bursts (on-demand scale-up and scale-out) and cloud bridging (on-the-fly integration of computational infrastructures). In the evaluation presented in this paper using this deployment, we demonstrate autonomic management behaviors for various objectives and constraints. We experimentally develop a model to estimate the runtime of application tasks on various resource classes, and use this model for initial resource provisioning. We also continuously update the model at runtime to ensure that user objectives are not violated. Finally, we also demonstrate that a public cloud (Amazon EC2) can be used to complement and reinforce more traditional high performance computing infrastructure (TeraGrid).

The rest of this paper is organized as follows. Section 2 describes the oil reservoir characterization application workflow used in this paper and introduces CometCloud. Section 3 describes the design of the autonomic management framework and its implementation on top of CometCloud. The target hybrid computing infrastructure, the different objectives and constraints addressed, and the operation of the autonomic management framework are presented in Section 3. Section 4 presents an evaluation of the framework. Related work is discussed in Section 5. Section 6 presents a conclusion.

2. Background

The software infrastructure in our experiments consist of two layers. The first layer is the science layer that is focused on the application. In our case the application is a reservoir characterization study that uses the ensemble Kalman filter. The second layer is the autonomic computing engine, which manages the workflow and provisions resources.

2.1. Application description

Reservoir characterization is an umbrella term for techniques that use direct and indirect information about a reservoir to construct accurate reservoir models. One of those techniques is history matching. History matching attempts to match actual reservoir

production with simulated reservoir production by modifying the models and therefore obtaining a more accurate set of reservoir models.

Among history matching techniques, the automatic history matching through Ensemble Kalman filters (EnKF) technique represents a promising approach that has gained a lot of popularity recently [15,16,18, 22]. In a typical EnKF study, an ensemble of models (of varying size and duration) is run through a reservoir simulator and their results are analyzed. The analysis step modifies the models and they are run through the reservoir simulator again. This process repeats in stages of reservoir simulation followed by analysis until no more reservoir data are available.

EnKF is a recursive filter that can be used to handle large, noisy data; the data in this case are the results and parameters from an ensemble of reservoir models that are sent through the filter to obtain the "true state" of the data. Since the model varies from one ensemble member to another, the run-time characteristics of the ensemble simulation are irregular and hard to predict. Furthermore, during simulations, when real historical data are available, all the data from the different ensemble members at that simulation time must be compared to the actual production data before the simulations are allowed to proceed. This translates into a global synchronization point for all ensemble members in any given stage.

The variation in computational requirements between individual tasks and stages can be large. As a result, the efficient execution of large scale complex reservoir characterization studies requires dynamic runtime management to ensure effective resource utilization and load-balancing. Furthermore, since the computational requirements are not known a priori, the application can potentially benefit from the elasticity of cloud resources. The components used in the characterization studies presented in this paper are:

- The Reservoir Simulator: The reservoir simulator [12] solves the equations for multiphase fluid flow through porous media, allowing us to simulate the movement of oil and gas in subsurface formations. It is based on the Portable Extensible Toolkit for Scientific Computing: PETSc [1].
- The Ensemble Kalman filter: The parallel EnKF [12] computes the Kalman gain matrix and updates the model parameters of the ensembles. The Kalman filter uses production data from the reservoir for it to update the reservoir models in real-time, and launch the subsequent long-term forecast, enhanced oil recovery and CO_2 sequestration studies.

2.2. CometCloud

CometCloud [21] is an autonomic computing engine that enables the dynamic and on-demand federation of clouds and grids as well as the deployment and execution of applications on these federated environments. It supports highly heterogeneous and dynamic cloud/grid infrastructures, enabling the integration of public/private clouds and autonomic cloud bursts, i.e., dynamic scale-out to clouds to address dynamic workloads, spikes in demands, and other extreme requirements. Conceptually, CometCloud is composed of a programming layer, service layer, and infrastructure layer. The infrastructure layer uses the Chord self-organizing overlay [32], and the Squid [29] information discovery and content-based routing substrate that is built on top of Chord. The routing engine supports flexible content-based routing and complex querying using partial keywords, wildcards, and ranges. It also guarantees that all peer nodes with data elements that match a query/message will be located.

The service layer provides a range of services to support autonomics at the programming and application level. This layer supports a Linda-like [4] tuple space coordination model, and provides a virtual shared-space abstraction as well as associative access primitives. The basic coordination primitives are the following; (1) *out*(ts, t): a nonblocking operation that inserts tuple t into space ts, (2) *in*(ts, t'): a blocking operation that removes a tuple t matching template t' from the space ts and returns it, (3) *rd*(ts, t'): a blocking operation that returns a tuple t matching template t' from the space ts without removing it from the space. Dynamically constructed transient spaces are also supported and enable applications to explicitly exploit context locality to improve system performance. Asynchronous (publish/subscribe) messaging and event services are also provided by this layer.

The programming layer provides the basic framework for application development and management. It supports a range of paradigms including the master/worker/BOT. Masters generate tasks which are described in XML and insert them into the Comet space using *out* operation. Workers pick up tasks from the space using *in* operation and consume them. Masters and workers can communicate via virtual shared space or using a direct connection. Scheduling and monitoring of tasks are supported by the application framework. The task consistency service handles lost/failed tasks. Other supported paradigms include workflow-based applications as well as MapReduce/ Hadoop [10,17].

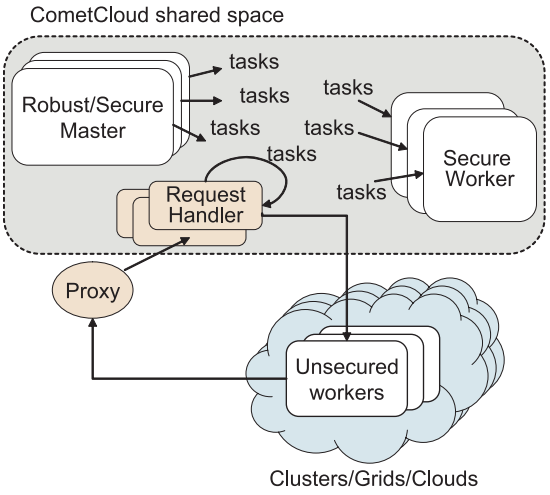


Fig. 1. A conceptual overview of CometCloud. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0319.)

The CometCloud master/worker/BOT layer supports the dynamic addition or removal of master and/or worker nodes from any of the federated environments (i.e., clouds, grids, local clusters, etc.) to enable on-demand scale up/down or out/in. It supports two classes of workers as shown in Fig. 1. Secure workers can access and possibly host part of the virtual shared space coordination infrastructure, while unsecured workers only provide computational cycles. CometCloud uses a *pull-based* task consumption model, i.e., workers pull tasks whenever they become idle. We can control the order in which the tasks are consumed by the workers by specifying the condition in the task tuples. For example, workers can pull a first come task first, a shorter deadline task first, or tasks satisfying a specific query condition, as well as any task regardless of values by range query. Also, we can support even push-based task consumption model by specifying worker to consume the task in its task tuple and then each worker can pull the tasks which are allocated to it. This model is well suited for cases where the capabilities of the workers and/or the computational requirements of the tasks are heterogeneous. The virtual shared space is used to host application tasks and possibly data associated with the tasks if it is small. Secure workers can connect to the space and *pull* task from the space. Unsecured works can only connect to the space through a proxy. CometCloud also provides autonomic management services and supports autonomic cloudburst driven by user-defined policies [19,20].

3. Autonomic workflow management

This section describes the autonomic management of the EnKF workflows using CometCloud. We first describe the target hybrid computing infrastructure. We then define the user objectives and underlying constraints that drive the autonomic management and the architecture of autonomic workflow manager. Finally, we describe the autonomic workflow management process including resource provisioning and resource adaptation.

3.1. Hybrid computing infrastructure

Computing infrastructures are getting increasingly hybrid, integrating different types of resource classes such as public/private clouds and grids from distributed locations [23,27,34]. In this work, we target such a federated hybrid computing infrastructure. As the infrastructure is dynamic and can contain a wide array of resource classes with different characteristics and capabilities, it is important to be able to dynamically provision the appropriate mix of resources based on applications objectives and requirements. Furthermore, application requirements and resource state may change, for example, due to workload surges, system failures or emergency system maintenance, and as a result, it is necessary to adapt the provisioning to match these changes in resource and application workload.

3.2. Autonomic management – objectives and constraints

Autonomic workflow management in the context of the hybrid and dynamic computing infrastructure described above can be decomposed into two aspects, *resource provisioning* and *resource adaptation*.In resource provisioning, the most appropriate mix of resource classes and the number of nodes of each resource class are estimated so as to match the requirements of the application and to ensure that the user objectives (e.g., throughput) and constraints (e.g., precision) are satisfied. Note that re-provisioning can be expensive in terms of time and other costs, and as a result, identifying the best possible initial provisioning is important. For example, if the initial estimate of required resources is not sufficient, additional nodes can be launched. However, this would involve additional delays due to, for example, time spent to create and configure new instances. In case of Amazon EC2 instances, our experience shows that this delay varies with image size and tends to be around 3 or 4 min. With different systems, for example FutureGrid [14], this

delay can be much higher. At runtime, delays can be caused by, for example, queue wait time variation, failures, premature job termination, performance fluctuation, performance degradation due to increasing user requests, etc. As a result, it is necessary to continuously monitor application execution and adapt resources to ensure that user objectives and constraints are satisfied.

The scheduling decision during resource provisioning and resource adaptation depends on the user objectives and the constraints. In this paper, we use the three objectives described below. However, additional objectives such as the energy-efficiency, minimized communication, etc, can also be used:

- *Acceleration*: This use case explores how clouds can be used as accelerators to reduce the application TTC by, for example, using cloud resources to exploit an additional level of parallelism by offloading appropriate tasks to cloud resources, given budget constraints.
- *Conservation*: This use case investigates how clouds can be used to conserve HPC grid allocations, within the appropriate runtime and budget constraints.
- *Resilience*: This use case investigates how clouds can be used to handle unexpected situations such as an unanticipated HPC Grid downtime, inadequate allocations, unanticipated queue delays or failures of working nodes. Additional cloud resources can be requested to alleviate the impact of the unexpected situations and meet user objectives.

To achieve the above user objectives, several constraints can be defined. In this paper, we consider two constraints:

- *Deadline*: The scheduling decision is to select the fastest resource class for each task and to decide the number of nodes per resource class based on the deadline. If the deadline can be achieved with a single node, then only one node will be allocated. When an application needs to be completed as soon as possible, regardless of cost and budget, the largest useful number of nodes is allocated.
- *Budget*: When a budget is enforced on the application, the number of allocatable nodes is restricted by the budget. If the budget is violated with the fastest resource class, then the next fastest and cheaper resource class is selected until the expected cost falls within the budget limit.

One or more constraints can be applied simultaneously. For example, we define *economical deadline* where resource class can be defined as the cheaper but slower resource class that can be allocated to save cost unless the deadline is violated. Also, multiple objectives can be combined as needed. An obvious example is combining an acceleration objective with a resilience objective.

The overall process for autonomic management consists of the following steps: (1) Initial benchmarks to estimate task runtime and projected TTC, (2) Initial resource scheduling and provisioning, (3) Resource monitoring and adaptation.

Initial benchmarks and estimation: The resource provisioning starts with small but representative benchmarks to estimate the runtime of all tasks on all resource classes. After we gather the benchmarks results from all resource classes, the best resource class for each task to achieve the user objective is selected based on the estimated runtime and the calculated costs for the estimated usage.

Scheduling and provisioning: Every task is mapped to the resource class most fitting to accommodate the user objectives. Should the user require the shortest possible TTC, i.e., acceleration, all tasks will be mapped to the resource class that provides the absolute best performance and the largest usable number of nodes will be allocated. Should acceleration be a primary objective within the confines of a budget (secondary objective), tasks will be mapped to the least costly resource class and the minimum required number of nodes required to meet the acceleration objective will be allocated. Tasks are grouped by the scheduled resource class and the number of nodes per resource class is decided. If we have enough resources so as to allocate a single node for a single task, then it can be one possible schedule to complete tasks most rapidly. However, this greedy schedule wastes resources especially when tasks are heterogeneous because the total runtime, or TTC, is decided by the longest task. For example, let us assume that three tasks (T_1, T_2, T_3) are mapped to EC2 and their estimated runtime are $3t$, $1t$ and $2t$, respectively. Then, it is a better schedule to allocate two nodes and map T_1 to one node and map T_2 and T_3 to another node instead of allocating three nodes mapping one task to a node. Total runtime is $3t$ for both schedules.

Monitor and adaptation: After the initial resource provisioning, the allocated resources and tasks are monitored. The autonomic framework continually updates the projected time of individual tasks and the TTC. If the user objective might be violated (for example, the updated TTC from results is larger than the initially estimated TTC or the updated cost is larger than

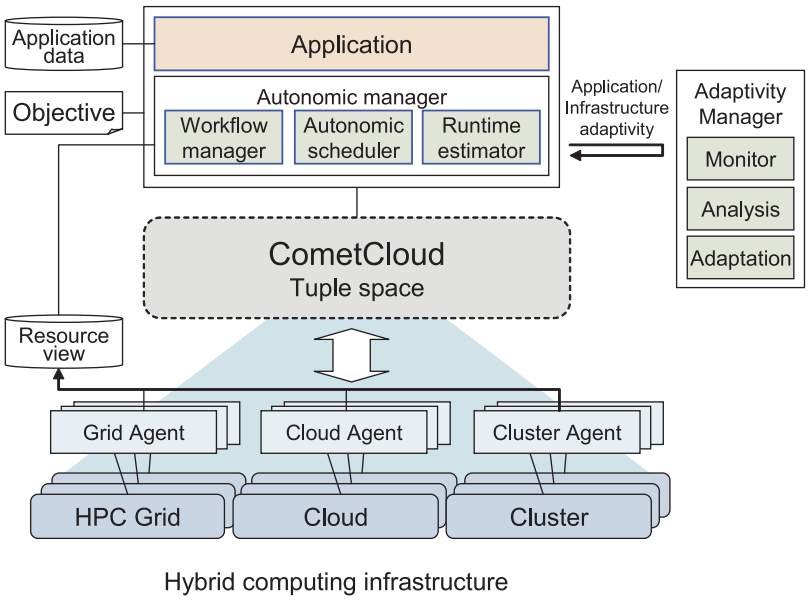


Fig. 2. The architecture of the autonomic management. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0319.)

the initially estimated cost), then additional, possibly different, resources will be provisioned and remaining tasks will be rescheduled.

Figure 2 shows the architecture of the autonomic manager. Each job has its own user objective that the framework seeks to honor. CometCloud supports master/worker, workflow and MapReduce/Hadoop application paradigms; however, in this paper we use the workflow based master/worker framework exclusively. The master generates tasks as XML tuples and inserts them into the CometCloud tuple space. Afterwards, the workers located in grids and public/private clouds pull tasks, execute them and send their results to the master. The agent of each grid and cloud helps pull tasks appropriately based on the job submission mechanism of resource classes. For example, TeraGrid has a batch queuing system where jobs should be inserted into the pilot job and workers pull tasks when they are ready by coming out from the queue. However, EC2 workers can start pulling tasks immediately once they are launched. The agent also manages its local resource view which includes node availability, failure, etc. The autonomic manager is responsible for managing workflows, estimating runtime and scheduling tasks at the beginning of every stage based on the resource view provided by the agents. In each stage, the adaptivity manger monitors tasks runtimes through results, handles the changes of application workloads and resource availability, and adapts resource provisioning if required. The detailed autonomic manager and adaptivity manager are described in the following subsections.

3.3. Resource provisioning

The autonomic manager is responsible for task scheduling, resource provisioning and application workflow management. It has the following four components for the resource provisioning.

Workflow manager: The workflow manager is responsible for coordinating the execution of the overall application workflow, based on the user objectives and the constraints.

Runtime estimator: The runtime estimator estimates the computational runtime and the cost of each task. This estimate can be obtained through a computational complexity model or through quick, representative benchmarks. Since performance is strongly affected by the underlying infrastructure (clouds or HPC grids) it is more effective to use benchmarks to obtain runtime and cost estimates.

Autonomic scheduler: The autonomic scheduler uses the information from the estimator modules and determines the optimal initial hybrid mix of HPC grids/clouds/clusters resources based on user/system-defined objectives, policies and constraints.

Grids/Clouds/Clusters agents: The grids and public/private clouds agents are responsible for provision-

ing the resources on their specific platforms, configuring *workers* as execution agents on these resources, and appropriately assigning tasks to these workers. CometCloud primarily supports a *pull-based* model for task allocation, where workers directly pull tasks from the CometCloud shared space. However, on typical HPC grid resources with a batch queue system, a combined *push–pull* model is used, where we insert "pilot-jobs" [33] containing workers into the batch queues and the workers then pull tasks from the space when they are scheduled to run.

3.4. Resource adaptation

The resource status can change during the run through application or resource performance degradation, increases in queue wait times, etc. Therefore, resource adaptation is necessary to prevent the violation of the user objective. The adaptivity manager is responsible for the resource adaptation and consists of the following components.

Monitor: The monitor observes the tasks runtimes which workers report to the master in the results. If the time difference between the estimated runtime and the task runtime on a resource is large, it could be that the runtime estimator under-estimated or over-estimated the task requirements, or the application/resource performance has changed. We can distinguish between those cases (estimation error versus performance change) if all of the tasks running on the resource in question show the same tendency. However, if the time difference between the estimated runtime and the task runtime on the resource increases after a certain point for most tasks running on the resource, then we can evaluate that the performance of the resource is becoming degraded. The monitor makes a function call to the analyzer to check if the changing resource status still meets the user objective when the time difference becomes larger than a certain threshold.

Analyzer: The analyzer is responsible for re-estimating the runtime of remaining tasks based on the historical data gathered from the results and evaluating the expected TTC of the application and total cost. If the analyzer observes the possibility for violating the user objective, then it calls for task rescheduling by the Adaptor.

Adaptor: When the adaptor receives a request for rescheduling, it calls the autonomic manager, in particular the autonomic scheduler, and retrieves a new schedule for the remaining tasks. The adaptor is responsible for launching more workers or terminating existing workers based on the new schedule.

4. Experiment

In this section, we describe the experimental environments and show the baseline results without applying the autonomic management to compare the performance of resource classes. Then we show the results with the autonomic management using the defined user objectives and the constraints.

4.1. Experimental environment

We implemented the autonomic and adaptivity manager in the CometCloud using Java. We ran with unsecured workers that do not share the CometCloud space based on the assumption that clouds and grids are not secure enough to manage private data. We used a local secure cluster (or a private cloud) to host the CometCloud management task, tuples and data. CometCloud worker tasks were run on the Ranger system as part of TeraGrid. We ran a proxy to redirect task requests from the unsecured workers to a request handler. The request handler picks up tasks from the CometCloud space and sends tasks to unsecured workers.

The request handler retrieve tasks from the space based on the enforced scheduling policy. The policy we employ is simple: TeraGrid workers are allowed to pull the largest tasks first, while EC2 workers pull the smallest tasks. As the number of tasks remaining in the space decreases, if there are TeraGrid resources still available, the autonomic scheduler may decide to throttle (i.e., lower the priority) EC2 workers to prevent them from becoming the bottleneck, since EC2 nodes are much slower than TeraGrid compute nodes. While this policy is not optimal, it was sufficient for our study.

We built a hybrid infrastructure with TeraGrid and 5 instance types of Amazon EC2. Each instance type is considered as a resource class. The number of cores on TeraGrid was set to 16. Detailed information about EC2 resource classes is described in Table 1.

The reservoir characterization workflow application launches ensembles of simulations in consecutive stages. In every stage, each ensemble member can have a unique size and duration. The output files from ensemble members of a stage are the input files of the next stage. Moving files across distributed resources is not a bottleneck at the moment; however, it might lead to future complications once data files become sufficiently large. We run a file server on Ranger to send input files to workers and collect their output. The en-

Table 1
EC2 instance types used in the experiments

Instance type	Mem (GB)	ECU	Virtual cores	Storage (GB)	Platform (bit)	IO	Linux cost ($/hour)
m1.small	1.7	1	1	160	32	Moderate	0.10
m1.large	7.5	4	2	850	64	High	0.34
m1.xlarge	15	8	4	1690	64	High	0.68
c1.medium	1.7	5	2	350	32	Moderate	0.17
c1.xlarge	7	20	8	1690	64	High	0.68

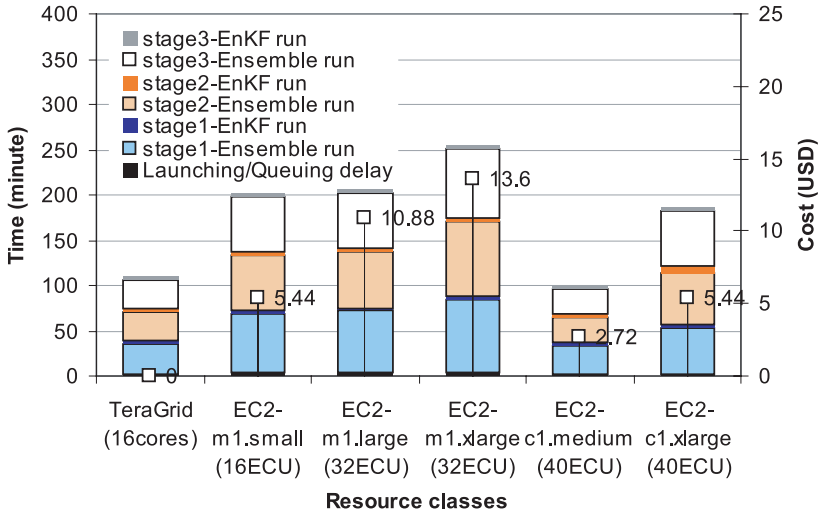


Fig. 3. Baseline results for TTC and the cost of using EC2 without using autonomic management. In each experiment, all ensemble tasks and Kalman filters were run on one type of resource class. The experiment run on EC2 c1.medium had the lowest time to completion (under 100 min) and the lowest cost among EC2 experiments. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2011-0319.)

semble Kalman filter application is also run on Ranger to reduce file transfer overhead.

In this paper, we fixed the number of stages to 3 and the number of ensemble members to 128 tasks of varying size and duration. The individual ensemble members and the EnKF application both use MPI on TeraGrid and EC2 instances. The types of EC2 instances used are: m1.large, m1.xlarge, c1.medium and c1.xlarge.

4.2. *Baseline without Autonomics*

To compare TTC and cost between resource classes, we ran the application with 16 cores of a single resource class without applying autonomics. As shown in Table 1, an EC2 core is a virtual core and the number of EC2 compute Units (ECU) forming a virtual core is different from instance types. We ran 16, 8, 4, 8, 2 nodes for m1.small, m1.large, m1.xlarge, c1.medium and c1.xlarge, respectively. Figure 3 shows the baseline result. It shows the whole three stages including ensemble runs and EnKF runs including EC2 launching delay or TeraGrid waiting queue delay. The m1.small and m1.large show similar TTC even though the number ECU of m1.large is twice as large as m1.small. This is due to the fact that the number of nodes for m1.large is half less than m1.small.

On the other hand, m1.xlarge shows larger TTC than m1.large even though they have the same number of ECUs. This is due to the fact that the number of nodes for m1.xlarge is less than m1.large. Interestingly, c1 instance types show better performance than m1 instance types because c1 instances are high CPU instances. Of particular interest is c1.medium which exhibits the least TTC among EC2 instances while spending the least cost.

The types c1.medium and c1.xlarge have the same number of ECUs but c1.xlarge exhibits a larger TTC. The number of nodes for c1.xlarge is 2 while the number of nodes for c1.medium is 8. The larger the num-

ber of simultaneous instances the more distributed the application is.

4.3. Autonomics with Objective 1: Acceleration

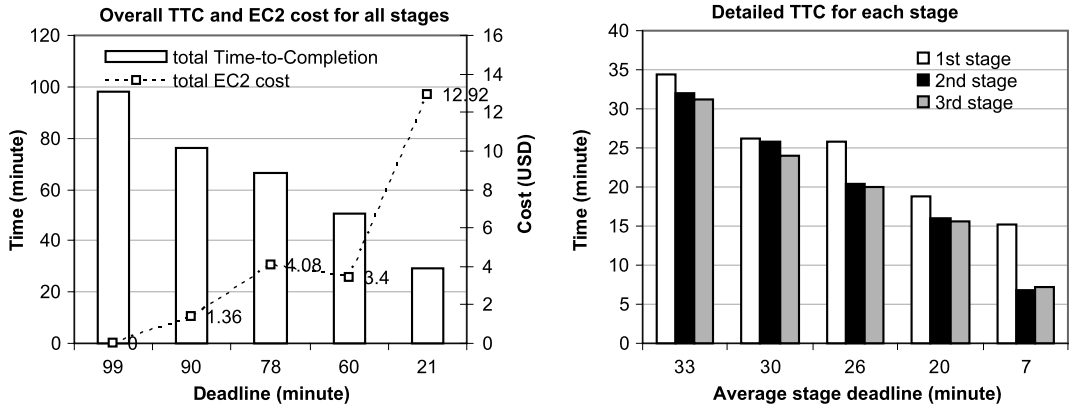
To accelerate the HPC application runtime, some tasks need to be offloaded to cloud resources regarding the accelerated runtime. We define three constraints to achieve acceleration: (1) *The Greedy Deadline*, (2) *The Economical Deadline* and (3) *The Budget Limit*. In greedy deadline, the offloaded tasks are scheduled to the fastest resource class regardless of the cost. This can guarantee the application deadline more safely. Economical deadline can be used if the user would like to spend budget economically avoiding choosing the expensive resource class for the fastest runtime but allowing to select less expensive resource class if it can achieve the application deadline closely. This can save budget but it might violate the application deadline slightly. If budget constraints are in effect, the scheduler allocates cloud nodes within the budget and the acceleration is achieved as much as the budget allows. Nodes are allocated at startup and terminated right before the one hour mark. This is due to the fact that EC2 instances are billed on an hourly basis. If there are remaining tasks after the budget is consumed, then they run on the HPC resource class.

Figure 4 shows the results of acceleration; using (a) greedy deadline, (b) economical deadline and (c) budget limit. A deadline of 99 min means there is no acceleration using EC2 (which means all members run on TeraGrid). Based on the TTC of ensemble members without acceleration, we change the deadline to 90, 78 60 and 21 min corresponding to 10, 20, 40 and 80% acceleration, respectively. The figures on the left show the overall TTC and EC2 cost for all stages and the right figures show the detailed runtime of each stage. Greedy deadline shows faster overall TTC than economical deadline but at a much higher cost. This is because the resource class of c1.xlarge is selected for greedy deadline where the estimated runtime of each task is the fastest but c1.medium is selected for economical deadline since it has the least cost while still meeting the deadline. In this configuration of economical deadline, it is more likely that we violate the performance deadline than in the greedy deadline case because it schedules tasks tightly close to the application deadline. The detailed schedule per stage is described in Table 2.

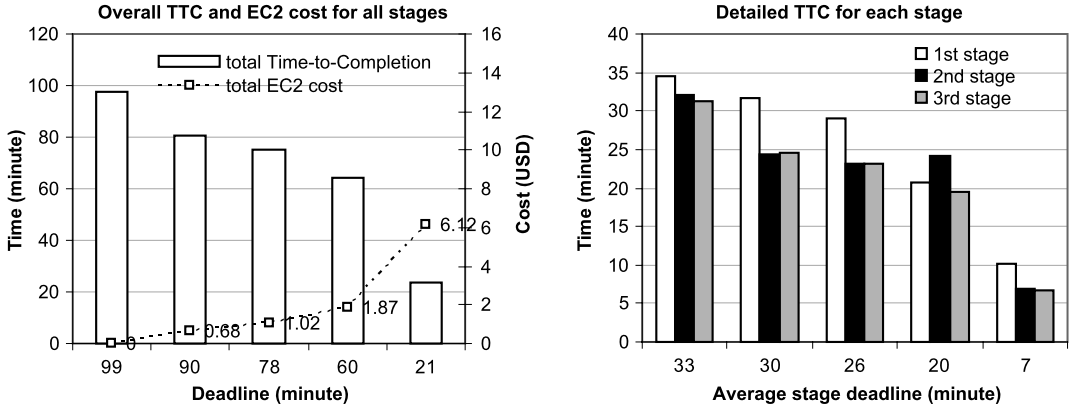
After the start of each stage, the time to completion of each task is monitored. If the gap between the required task time to completion (to meet the deadline) and the estimated task time to completion remains large for a certain number of tasks, the adaptivity manager adjusts the schedule. In these experiments, we adjust the schedule only when the number of EC2 workers needs to be increased since EC2 bills hourly once instances start. The number of added nodes is decided by the remaining time up to the target deadline and the remaining number of tasks. In Fig. 4(a), the greedy deadline of 21 min is slightly violated and its TTC is even larger than that of economical deadline. This is because the stage 1 takes pretty long time (around 15 min) compared with other stages as shown in right figure and this increases the whole application TTC. At 21 min deadline in stage 1, we start with 14 nodes of c1.xlarge for the greedy deadline case and 17 of c1.medium for the economical deadline case. After we observed some tasks with a runtime that was higher than the estimated runtime, the adaptivity manager decided to increase the number of nodes with an additional 5 and 19 nodes for greedy deadline and economical deadline respectively, placing the total at the end of stage 1 to 19 and 36. Since the rescheduling decision of greedy deadline was made slightly later than that of economical deadline and the launching delay of c1.xlarge was larger than that of c1.medium, the time when resource adaptation is applied and all new nodes were ready to work was 8.40 min for greedy deadline and 5.98 min for economical deadline. The deadline was set for each stage (7 min per stage), hence applying resource adaptation for greedy deadline happened a little late. Figure 4(c) shows the results when the budget limit is applied. With a larger budget, the TTC for all stages decreases. The runtime of stage 3 at the $1 budget limit is larger than other stages because all EC2 instances are terminated during stage 3 after spending one hour. The detailed schedule for the budget limit is also included in Table 2.

4.4. Autonomics with Objective 2: Conservation

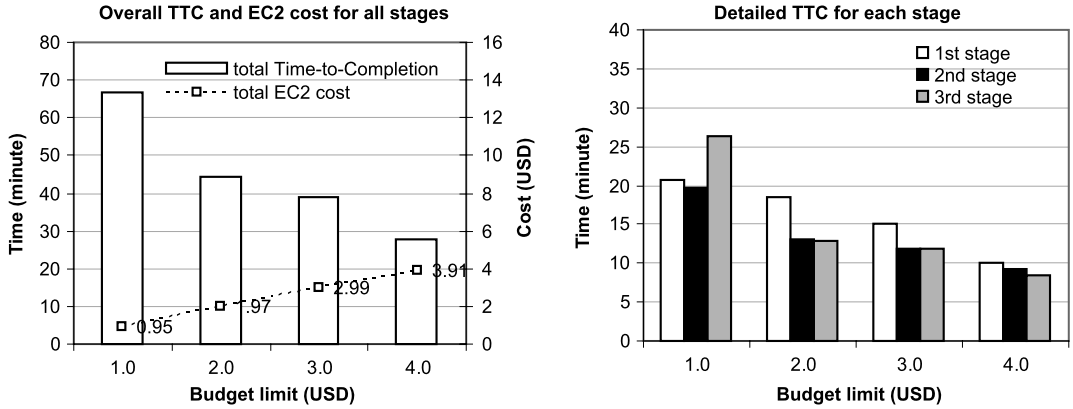
In this scenario, we assume that the CPU time usage of TeraGrid is limited. The application deadline is set to 33 min which is the application TTC when all tasks run on TeraGrid. Our goal in this scenario is to accomplish the same deadline even when we have the limited TeraGrid CPU usage. Figure 5 shows the overall TTC and EC2 cost as well as the schedule for stage 1 varying the time of TeraGrid CPU usage. EC2 c1.medium is selected as a cloud resource class because we use economical deadline to save budget usage



(a) Greedy deadline



(b) Economical deadline

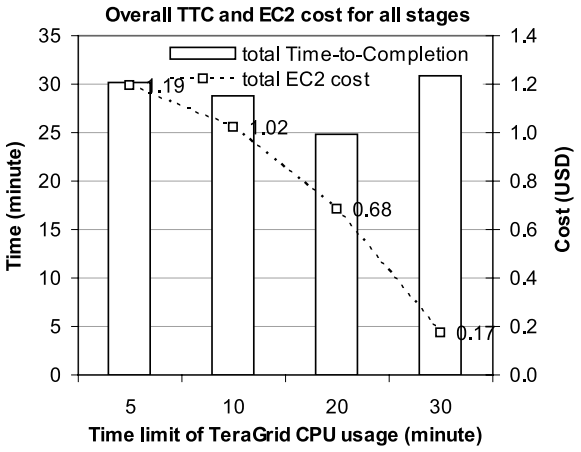


(c) Available budget

Fig. 4. Objective 1: Acceleration. The figures on the left show the overall TTC and EC2 cost and the right figures show the detailed TTC for each stage of (a) the greedy deadline to aggressively meet the deadline, (b) the economical deadline to meet the deadline within budget, and (c) the acceleration within the available budget. The autonomic management framework is able to honor the constraints in all three usage modes: meet the deadline at any cost, meet the deadline within budget and use as much of the budget as possible to reduce the deadline.

Table 2
The number of running workers at the end of each stage for Objective 1: Acceleration

	Deadline or budget	Num of workers			Resource class
		Stage 1	Stage 2	Stage 3	
Greedy deadline	90 min	1	1	1	c1.xlarge
	78 min	2	2	4	c1.xlarge
	60 min	5	5	5	c1.xlarge
	21 min	19	19	19	c1.xlarge
Economical deadline	90 min	2	2	2	c1.medium
	78 min	3	3	3	c1.medium
	60 min	6	5	5	c1.medium
	21 min	36	36	36	c1.medium
Budget limit	1 USD	5	5	0	c1.medium
		1	1	0	m1.small
	2 USD	11	11	11	c1.medium
		1	1	1	m1.small
	3 USD	17	17	17	c1.medium
		1	1	1	m1.small
	4 USD	23	23	23	c1.medium



(a) TTC and EC2 cost

	Test 1	Test 2	Test 3	Test 4
CPU usage limit (min)	5	10	20	30
Num of scheduled VMs (EC2)	7	6	4	1
Num of expected tasks consumed by EC2	111	92	54	14
Consumed tasks by EC2	109	89	49	16

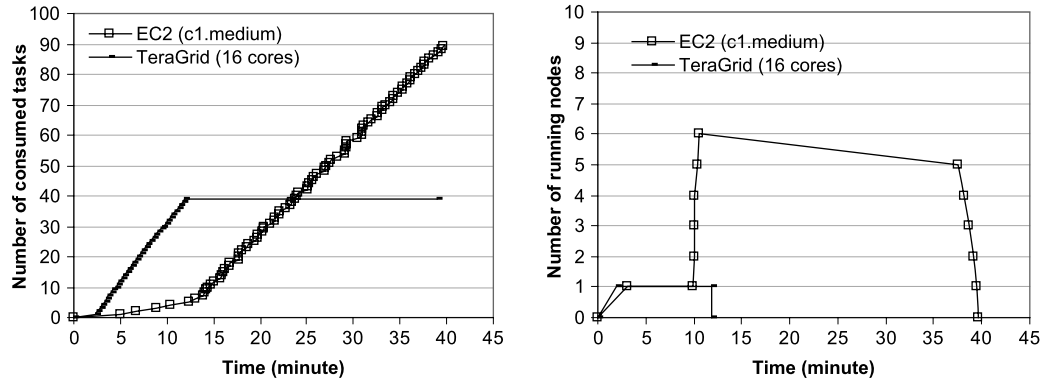
(b) Schedule

Fig. 5. Objective 2: Conservation. EC2 takes the place of TeraGrid to accomplish the target deadline. (a) shows the application TTC and EC2 cost varying the CPU usage limit of TeraGrid and (b) shows the scheduled nodes, tasks and the consumed tasks for EC2.

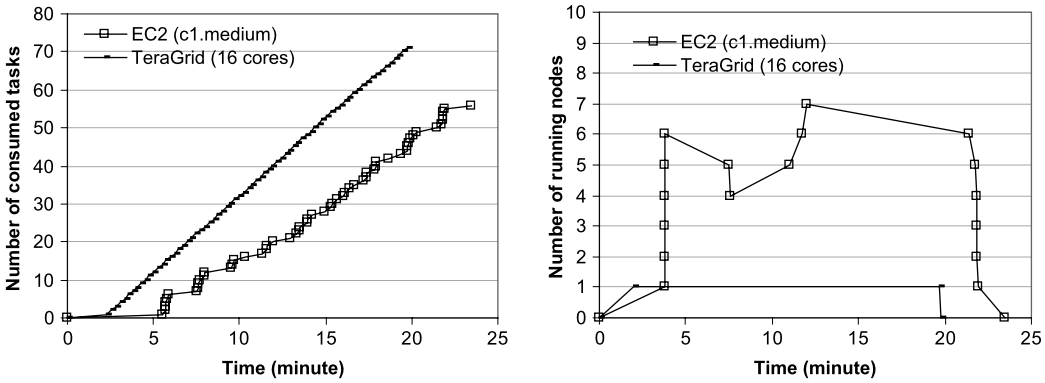
and c1.medium shows fast runtime even with low cost. As shown in Fig. 5(a), all runs are completed within the deadline which means the user objective is not violated. Interestingly, the TTC decreases when we can use more TeraGrid CPU time except at 30 min CPU usage. If EC2 executes more tasks then the total runtime can increase because EC2 takes longer time than TeraGrid to complete a task (the average runtime of tasks on EC2 is 2 min while that on TeraGrid is only 15 s). Furthermore, since EC2 workers consume small tasks first in the experiment, the more tasks EC2 workers consume, the longer tasks they take. This is why the TTC at 20 min CPU usage is faster than at 5 min CPU usage. If TeraGrid consumes most of tasks, the TTC becomes close to the application deadline, 33 min. At 30 min time limit of TeraGrid CPU usage, one EC2 worker is allocated and it consumes only 16 tasks. TeraGrid consumes most of tasks and this causes the TTC becomes close to the deadline.

4.5. *Autonomics with Objective* 3: *Resilience*

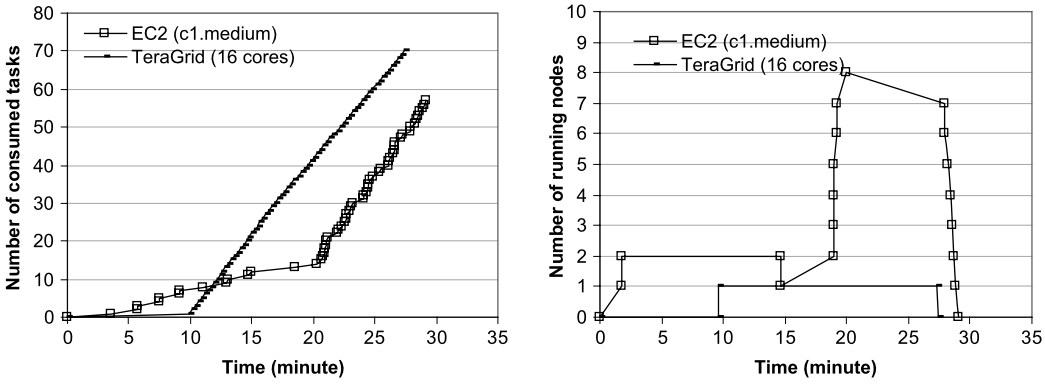
Resilience is a measure of elastic recovery from unexpected failure or delays that would otherwise violate the user objective. We consider TeraGrid failure (or delayed starts) as well as EC2 failure and conducted the experiment for stage 1. Figure 6 shows the changing number of consumed tasks and the number of running nodes on each resource class. The plots show the whole run including TeraGrid waiting time in the queue, EC2



(a) Resilience for TeraGrid failure



(b) Resilience for EC2 failure



(c) Resilience for TeraGrid and EC2 failure

Fig. 6. Objective 3: Resilience. (a) The CPU time limit of TeraGrid is changed from 30 to 10 min (the deadline is set to 33 min). (b) Two EC2 instances are failed at 8 min (the deadline is set to 20 min). (c) The waiting time in the queue of TeraGrid is increased and one EC2 instance is failed at 15 min (the deadline is set to 26 min).

launching delay, the application runtime and instance termination. Note that the deadline is applied as a constraint and is considered for the ensemble runs. The first scenario is as follows: the user would like to complete all tasks in 33 min and initially the CPU time usage of TeraGrid is limited to 30 min; however, after 5 min, the limit is curtailed to 10 min. Figure 6(a) shows how clouds cover the situation. At the beginning of the run, only one EC2 VM is scheduled based on the initial TeraGrid time limit and the number of consumed tasks by EC2 is slowly increasing. After 5 min of the application (around 8 min in the graph) 5 more

instances are scheduled to achieve the deadline based on the changed TeraGrid time limit and the new nodes start running at 10–11 min. After 10 min, the TeraGrid worker does not consume tasks any more while EC2 nodes keep consuming tasks. Figure 6 (b) shows the second scenario which is two EC2 instances suffer induced catastrophic failure during the run at around 8 min. The deadline is set to 20 min. Initially 6 EC2 instances are scheduled to achieve a 20 minute-deadline and three more instances are added after two instances suffer induced failure. This decision is made based on the remaining time to the deadline and the number of remaining tasks. In the last scenario, the deadline is set to 26 min and one EC2 instance is forcibly terminated during the run around 15 min from startup. (Fig. 6(c)). Furthermore, the TeraGrid worker suffers a 10 min delay in the queue. Hence, the adaptivity manager decides to add 7 nodes to cover the EC2 worker failure and the TeraGrid queue delay. Our results establish that adaptive resource provisioning can compensate for worker failure and honor the application deadline. Note that we do not include EC2 initial launching delay in the application TTC.

5. Related work

InterGrid [9] and metaschedulers [2] interconnect different grids for example, however, there are efforts to include clouds such as Amazon EC2 into their integrated computing infrastructures because the current public clouds use VM-based on-demand resource provisioning which enables customized and isolated computing environments and it makes public clouds attractive. Buyya et al. [8] described an approach of extending a local cluster to cloud resources using schedulers applying different strategies. They simulated the approach and did not include the real run of cloud resources. The research in [27,34] implied the elastic growth of grids to clouds. Vazquez et al. [34] proposed the architecture for an elastic grid infrastructure using GridWay metascheduler and extended grid resources to Globus Nimbus [13]. They experimented with a NAS Grid Benchmark suite and all resource classes were based on Globus (no VM). Ostermann et al. [27] extended a grid workflow application development and computing infrastructure to include cloud resources and experimented with Austrian Grid and an academic cloud installation of Eucalyptus using a scientific workflow application.

Our previous works also considered a hybrid computing infrastructure where we deployed Value-at-Risk financial application on Rutgers cluster and Amazon EC2 in [21] and medical image registration on a combination of private clouds at Rutgers University and the Cancer Institute of New Jersey and EC2 in [19]. Also, we built a hybrid infrastructure with TeraGrid and Amazon EC2 and deployed EnKF application with various workloads in [20]. In these works, on-demand autonomic cloudbursts were enabled using CometCloud [6]. Compared with the above related works which use specific grid based job schedulers and/or resource managers, our work does not depend on any specific type of resource class and therefore can integrate various resource classes such as public/private clouds, grids of desktops, mobile computing environments and so on.

Several economic models for resource scheduling on Grids have been proposed. Recently, a combinatorial auction model [28] was proposed for both grids and clouds. A cost model based on economic scheduling heuristics [24] was investigated for cloud-based streams. An adaptive scheduling mechanism [25] used economic tools such as market, bidding, pricing, etc. on an elastic grid utilizing virtual nodes from clouds. In that case, GridARM [7] and GLARE [30] were used as the resource management systems. However, there are very few economic models for hybrid computing infrastructure.

Some of scheduling techniques developed for grid environment have been applied for virtual machine based cloud environment. OpenNebula [26] uses advance reservation in the Haizea lease manager and Buyya et al. [3] extended SLA management policies for grids to clouds. Adaptive scheduling was proposed in [31] where different loads and resource availability were considered to reshape jobs and resize VMs. Chunlin et al. [5] proposed a cross-layer QoS optimization policy for computational grid because there are different QoS metrics at different layers to be achieved.

An on-demand resource provisioning mechanism based on load was presented in [11]. In contrast, the framework we proposed is based on the user defined objective and constraints which can be defined widely including workloads, time constraints and budget limits. The autonomic scheduler decides on the mix of resource classes and the number of nodes over the hybrid infrastructure depending on the user objectives, the estimated runtime of tasks and the cost calculated from the runtime.

6. Conclusion

In this paper, we proposed a framework for autonomic management of workflow applications on hybrid computing infrastructure. The autonomic manager estimates runtime using a computational model or benchmarks and schedules tasks to achieve the user objectives within the constraints. The adaptivity manager monitors the resources and adjusts the schedules to honor the user objectives. We defined several user objectives such as acceleration of the HPC time-to-completion, conservation of HPC CPU usage, and resilience of grids/clouds nodes using public clouds. We also defined greedy deadline, economical deadline and available budget limit as constraints to achieve the user objectives. We built a hybrid infrastructure with TeraGrid and several instance types of Amazon EC2. We conducted experiments on the hybrid computing infrastructure with varying user objectives and constraints. The results show how public clouds can be used to achieve user performance and cost objectives and recover from unexpected delays and failures. We also showed that the proposed autonomic management framework can support application workflows that consist of heterogeneous tasks with varying computational requirements.

Acknowledgements

The research presented in this paper is supported in part by National Science Foundation via grants numbers IIP 0758566, CCF-0833039, DMS-0835436, CNS 0426354, IIS 0430826 and CNS 0723594, by Department of Energy via grant numbers DE-FG02-06ER54857, DE-FG02-04ER46136 (UCoMS), by a grant from UT Battelle, and by an IBM Faculty Award, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University. Experiments on the Amazon Elastic Compute Cloud (EC2) were supported by a grant from Amazon Web Services and CCT CyberInfrastructure Group grants.

References

[1] S. Balay, K. Buschelman, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith and H. Zhang, PETSc Web page, 2001, available at: http://www.mcs.anl.gov/petsc.

[2] N. Bobroff, L. Fong, S. Kalayci, Y. Liu, J.C. Martinez, I. Rodero, S.M. Sadjadi and D. Villegas, Enabling interoperability among meta-schedulers, in: *CCGRID'08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, Washington, DC, USA, 2008, pp. 306–315.

[3] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg and I. Brandic, Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility, *Future Gener. Comput. Syst.* **25** (2009), 599–616.

[4] N. Carriero and D. Gelernter, Linda in context, *Commun. ACM* **32**(4) (1989), 444–458.

[5] L. Chunlin and L. Layuan, Cross-layer optimization policy for QoS scheduling in computational grid, *J. Netw. Comput. Appl.* **31** (2008), 258–284.

[6] CometCloud, http://www.cometcloud.org.

[7] K. Czajkowski, S. Fitzgerald, I. Foster and C. Kesselman, Grid information services for distributed resource sharing, in: *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing 2001*, IEEE, Piscataway, NJ, USA, 2001, pp. 181–194.

[8] M.D. de Assunção, A. di Costanzo and R. Buyya, Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters, in: *HPDC'09: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, ACM, New York, NY, USA, 2009, pp. 141–150.

[9] M.D. de Assunção, R. Buyya and S. Venugopal, Intergrid: a case for internetworking islands of grids, *Concurrency and Computation: Practice and Experience* **20**(8) (2008), 997–1024.

[10] J. Dean and S. Ghemawat, Mapreduce: simplified data processing on large clusters, *Commun. ACM* **51**(1) (2008), 107–113.

[11] T. Dornemann, E. Juhnke and B. Freisleben, On-demand resource provisioning for BPEL workflows using Amazon's elastic compute cloud, in: *CCGRID'09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, Washington, DC, USA, 2009, pp. 140–147.

[12] Y.Y. El-Khamra, Real-time reservoir characterization and beyond: Cyberinfrastructure tools and technologies, Master's thesis, Louisiana State University, Baton Rouge, LA, 2009.

[13] T. Freeman and K. Keahey, Flying low: Simple leases with workspace pilot, in: *Euro-Par*, 2008, pp. 499–509.

[14] FutureGrid, http://www.futuregrid.org/.

[15] Y. Gu and D.S. Oliver, The ensemble Kalman filter for continuous updating of reservoir simulation models, *Journal of Engineering Resources Technology* **128**(1) (2006), 79–87.

[16] Y. Gu and D.S. Oliver, An iterative ensemble Kalman filter for multiphase fluid flow data assimilation, *SPE Journal* **12**(4) (2007), 438–446.

[17] Hadoop, http://hadoop.apache.org/core/.

[18] R.E. Kalman, A new approach to linear filtering and prediction problems, *Transactions of the ASME – Journal of Basic Engineering* **82**(Series D) (1960), 35–45.

[19] H. Kim, M. Parashar, D. Foran and L. Yang, Investigating the use of autonomic cloudbursts for high-throughput medical image registration, in: *10th IEEE/ACM International Conference on Grid Computing 2009*, IEEE, Piscataway, NJ, USA, 2009, pp. 34–41.

[20] H. Kim, Y. el Khamra, S. Jha and M. Parashar, An autonomic approach to integrated HPC grid and cloud usage, in: *Fifth IEEE International Conference on e-Science, 2009, e-Science'09*, December 2009, pp. 366–373.

[21] H. Kim, S. Chaudhari, M. Parashar and C. Marty, Online risk analytics on the cloud, in: *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009, CCGRID'09*, IEEE, Piscataway, NJ, USA, 2009, pp. 484–489.

[22] X. Li, C. White, Z. Lei and G. Allen, Reservoir model updating by ensemble Kalman filter – practical approaches using grid computing technology, in: *Petroleum Geostatistics 2007*, Cascais, Portugal, August 2007.

[23] P. Marshall, K. Keahey and T. Freeman, Elastic site: Using clouds to elastically extend site resources, in: *IEEE International Symposium on Cluster Computing and the Grid*, IEEE, Piscataway, NJ, USA, 2010, pp. 43–52.

[24] P. Martinaitis, C. Patten and A. Wendelborn, Remote interaction and scheduling aspects of cloud based streams, in: *5th IEEE International Conference on E-Science Workshops, 2009*, IEEE, Piscataway, NJ, USA, 2009, pp. 39–47.

[25] L. Nie and Z. Xu, An adaptive scheduling mechanism for elastic grid computing, in: *SKG'09: Proceedings of the 2009 Fifth International Conference on Semantics, Knowledge and Grid*, IEEE Computer Society, Washington, DC, USA, 2009, pp. 184–191.

[26] Open Nebula Project, http://www.opennebula.org/doku.php.

[27] S. Ostermann, R. Prodan and T. Fahringer, Extending grids with cloud resource management for scientific computing, in: *10th IEEE/ACM International Conference on Grid Computing 2009*, IEEE, Piscataway, NJ, USA, 2009, pp. 42–49.

[28] A. Ozer and C. Ozturan, An auction based mathematical model and heuristics for resource co-allocation problem in grids and clouds, in: *Fifth International Conference on Soft Computing, Computing with Words and Perceptions in System Analysis, Decision and Control, 2009, ICSCCW 2009*, IEEE, Piscataway, NJ, USA, 2009, pp. 1–4.

[29] C. Schmidt and M. Parashar, Squid: Enabling search in DHT-based systems, *J. Parallel Distrib. Comput.* **68**(7) (2008), 962–975.

[30] M. Siddiqui, A. Villazon, J. Hofer and T. Fahringer, Glare: A grid activity registration, deployment and provisioning framework, in: *Proceedings of the ACM/IEEE SC 2005 Conference on Supercomputing 2005*, IEEE, Piscataway, NJ, USA, 2005, p. 52.

[31] A. Sodan, Adaptive scheduling for qos virtual machines under different resource allocation – first experiences, in: *Job Scheduling Strategies for Parallel Processing*, E. Frachtenberg and U. Schwiegelshohn, eds, Lecture Notes in Computer Science, Vol. 5798, Springer, Berlin/Heidelberg, pp. 259–279.

[32] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek and H. Balakrishnan, Chord: A scalable peer-to-peer lookup protocol for internet applications, in: *ACM SIGCOMM*, ACM, New York, NY, USA, 2001, pp. 149–160.

[33] D. Thain, T. Tannenbaum and M. Livny, Distributed computing in practice: The condor experience, *Concurrency and Computation: Practice and Experience* **17** (2005), 2–4.

[34] C. Vazquez, E. Huedo, R. Montero and I. Llorente, Dynamic provision of computing resources from grid infrastructures and cloud providers, in: *Workshops at the Grid and Pervasive Computing Conference, 2009, GPC'09*, IEEE, Piscataway, NJ, USA, 2009, pp. 113–120.