An Autonomic Approach to Integrated HPC Grid and Cloud Usage

Hyunjoo Kim[1], Yaakoub el-Khamra[3], Shantenu Jha[2], Manish Parashar[1]

[1] *NSF Center for Autonomic Computing, Dept. of Electrical & Computer Engr., Rutgers University, , NJ, USA*

[2]*Center for Computation & Tech. and Dept. of Computer Science, Louisiana State University, USA*

[3]*Texas Advanced Computing Center, Austin, Texas, USA*

Abstract

Clouds are rapidly joining high-performance Grids as viable computational platforms for scientific exploration and discovery, and it is clear that production computational infrastructures will integrate both these paradigms in the near future. As a result, understanding usage modes that are meaningful in such a hybrid infrastructure is critical. For example, there are interesting *application workflows that can benefit from such hybrid usage modes to, perhaps, reduce times to solutions, reduce costs (in terms of currency or resource allocation), or handle unexpected runtime situations (e.g., unexpected delays in scheduling queues or unexpected failures). The primary goal of this paper is to experimentally investigate, from an applications perspective, how autonomics can enable interesting usage modes and scenarios for integrating HPC Grid and Clouds. Specifically, we used a reservoir characterization application workflow, based on Ensemble Kalman Filters (EnKF) for history matching, and the CometCloud autonomic Cloud engine on a hybrid platform consisting of the TeraGrid and Amazon EC2, to investigate 3 usage modes (or autonomic objectives) – acceleration, conservation and resilience.*

I. Introduction and Motivation

Significant investment and technological advances have established high-performance computational (HPC) Grid [1] infrastructures as dominant platforms for large-scale parallel/distributed computing in science and engineering. Infrastructures such as the TeraGrid (TG), EGEE and DEISA integrate high-end computing and storage systems via high-speed interconnects, and support traditional, batch-queue-based computationally/data intensive high-performance applications.

More recently, Cloud services have been playing an increasingly important role in computational research, and are posed to become an integral part of computational research infrastructures. Clouds support a different although complementary usage model as compared to HPC Grids – one that is based on on-demand access to computing utilities, an abstraction of unlimited computing resources, and a usage-based payment model made users essentially "rent" virtual resources and pay for what they use. Underneath these Cloud services are consolidated and virtualized data centers that provide virtual machine (VM) containers hosting applications from large numbers of distributed users.

It is now clear that production computational Grid infrastructures will integrate these two paradigms providing a hybrid computing environment that integrates traditional HPC Grid services with on-demand Cloud services. The usage models and modes of such a hybrid infrastructure, as well as frameworks for supporting these usage modes, are still not as clear. There are application profiles that are better suited to HPC Grid (e.g., large QM calculations), and others that are more appropriate for Clouds. However, there are also large numbers of applications that have *interesting* workload characteristics and resource requirements, and can benefit from hybrid usage modes and adapting objectives, for example, reduce times-to-solutions, reduce costs (in terms of currency or resource allocation), or handle expected runtime situations (e.g., unexpected delays in scheduling queues or unexpected failures).

Furthermore, developing and running applications in such a hybrid and dynamic computational infrastructure presents new and significant challenges. These include the need for programming systems that can express the hybrid usage modes and associated runtime trade-offs and adaptations, as well as coordination and management infrastructures that can implement them in an efficient and scalable manner. Key issues include decomposing applications, components and workflows, determining and provisioning the appropriate mix of Grid/Cloud resources, and dynamically scheduling them across the hybrid execution environment while satisfying/balancing multiple, possibly changing objectives for performance, resilience, budgets and so on.

The primary goal of this paper is to experimentally investigate, from an applications perspective, interesting usage modes and scenarios for integrating HPC Grids and Clouds, and how they can be effectively enabled using autonomic computing concepts. Our investigation and analysis is driven by a real-world application that is the basis for a large number of physical and engineering science problems. Specifically, we use the reservoir characterization application workflow, which uses Ensemble Kalman Filters (EnKF) for history matching, as the driving application and

[1]Note that this work addresses High-Performance Grids rather than High-Throughput Grids.

investigate how the Amazon EC2 commercial Cloud can be used to complement the TG. The EnKF workflow presents an interesting use-case due to the heterogeneous computational requirements of the individual ensemble members as well as the dynamic nature of the overall workflow.

We also aim to demonstrate how the CometCloud engine [1], [2] and the autonomic application management framework can support these heterogeneous application requirements as well as hybrid usage modes. As part of the autonomic management framework we have implemented a pilot-job capability, but with the novel feature that its workload is policy and objective driven. CometCloud is an autonomic Cloud engine: it enables applications on virtual Cloud infrastructures with re-sizable computing capacity through policy-driven autonomic Cloud bridging (on-the-fly integration of Grids, commercial and community Clouds and local computational environments) and Cloudbursts (dynamic scale-out to address dynamic workloads, spikes in demands, and other extreme requirements).

This paper investigates the following scenarios (or autonomic objectives) for the integration of HPC Grids and Clouds and how an autonomic framework can support them:

- **Acceleration:** This use case explores how Clouds can be used as accelerators to improve the application time-to-completion by, for example, using Cloud resources to alleviate the impact of queue wait times or exploit an additionally level of parallelism by offloading appropriate tasks to Cloud resources, given appropriate budget constraints.
- **Conservation:** This use case investigates how Clouds can be used to conserve HPC Grid allocations, given appropriate runtime and budget constraints.
- **Resilience:** This use case will investigate how Clouds can be used to handle unexpected situations such as an unanticipated HPC Grid downtime, inadequate allocations or unanticipated queue delays.

There have been related research efforts where VMs and regular batch-queue systems have been used together [3], or where regular Grid Metascheduling systems have been extended to include provisioning VMs [4]; however, this work differs from other approaches in a couple of important ways: First, in this work we investigate hybrid models of execution, driven by application need as opposed to a system-integration problem. Second, and at least as important, we perform experiments on production Grid infrastructure, and thus investigate practical system constraints and challenges.

The rest of this paper is organized as follows: the next section provides a detailed introduction to the EnKF-based application and CometCloud. This then provides the basis for a discussion of the architecture of the CometCloud-based autonomic management framework (Section III). A key aim of our paper is to investigate the role of autonomics in determining dynamically the scheduling and pricing trade-offs when using HPC Grids and Clouds collectively.

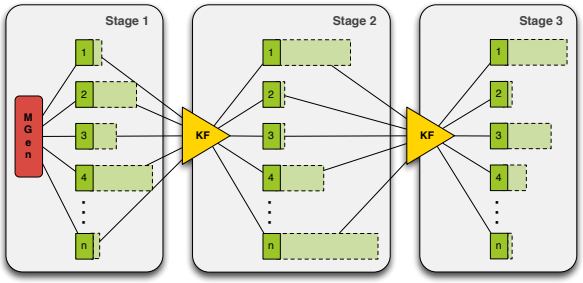

Fig. 1: Schematic illustrating the variability between stages of a typical ensemble Kalman filter based simulation. The end-to-end application consists of several stages; in general at each stage the number of models generated varies in size and duration.

We describe the three usage modes in Section IV, and provide a description and analysis of the experiments we perform to understand the various trade-offs in determining optimal task distributions. We summarize and conclude with a quick overview of future issues and challenges arising from this work.

II. Application Characteristics and Infrastructure Architecture

A. Reservoir Characterization: EnKF-based History Matching

Direct information about any given reservoir is usually gathered through logging and measurement tools including core samples, thus restricted to a small portion of the actual reservoir size, namely the well-bore. For this reason, "history matching" techniques have been developed to "match" actual reservoir production with simulated reservoir production, therefore obtaining a more "satisfactory" set of reservoir models. Ensemble Kalman filters (EnKF) represent a promising approach to history matching [5], [6], [7], [8].

Ensemble Kalman filters are recursive filters that can be used to handle large, noisy data; the data in this case are the results and parameters from ensembles of reservoir models that are sent through the filter to obtain the "true state" of the data. Since the reservoir model varies from one ensemble to another, the run-time characteristics of the ensemble simulation are irregular and hard to predict. Furthermore, during simulations when real historical data is available, all the data from the different ensembles 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.

Due to this fundamental limit on task-parallelism, wildly varying computational requirements between stages and for different tasks in a stage, performing large scale studies for complex reservoirs in a reasonable amount of time would benefit greatly from the use of a wide range of distributed,

high-performance and throughput as well as on-demand computing resources. The simulation components are:

- The Reservoir Simulator: The BlackOil reservoir simulator 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 Cactus Code [9] high performance scientific computing framework and the Portable Extensible Toolkit for Scientific Computing: PETSc [10]. With a few parameter changes, BlackOil is also used for modelling the flow of CO_2; however it does not simulate any geochemical interactions. While adequate as a first order approximation, it is still under intense development to enable it to satisfactorily model geochemical phenomena.
- The Ensemble Kalman filter: Also based on Cactus and PETSc, computes the Kalman gain matrix and updates the model parameters of the ensembles. The Kalman filter requires live 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.

B. An Overview of CometCloud

CometCloud is an autonomic computing engine for Cloud and Grid environments. It is based on the Comet [11] decentralized coordination substrate, and supports highly heterogeneous and dynamic Cloud/Grid infrastructures, integration of public/private Clouds and autonomic Cloudbursts. Conceptually, CometCloud is composed of a programming layer, service layer, and infrastructure layer. The infrastructure layer uses the Chord self-organizing overlay [12], and the Squid [13] information discovery and content-based routing substrate build on top of Chord. The routing engine supports flexible content-based routing and complex querying using partial keywords, wildcards, or 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 supports autonomics at the programming and application level. This layer supports a Linda-like [14] tuple space coordination model, and provides a virtual shared-space abstraction as well as associative access primitives. Dynamically constructed transient spaces are also supported to allow applications to explicitly exploit context locality to improve system performance. Asynchronous (publish/subscribe) messaging and event services are also provides 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 and workers 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.

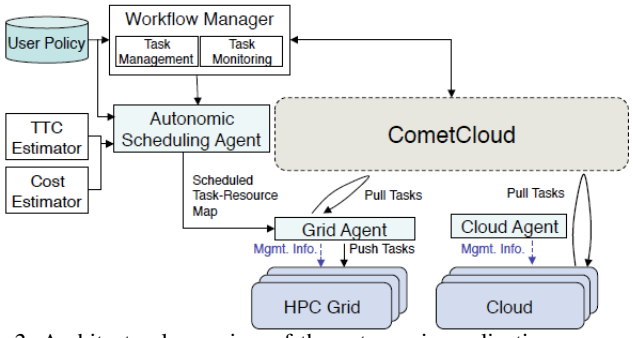

Fig. 2: Architectural overview of the autonomic application management framework.

III. Autonomic Application Management for HPC Grids-Cloud Integration

A schematic overview of the CometCloud-based autonomic application management framework for enabling hybrid HPC Grids-Cloud usage modes is presented in Figure 2. The framework is composed of autonomic managers that coordinate using Comet coordination spaces [11] that span, and can be transparently accessed across the integrated execution environment. The key components of the management framework are described below.

Workflow Manager: The workflow manager is responsible for coordinating the execution of the overall application workflow, based on user-defined polices, using Comet spaces. It includes a workflow planner as well as task monitors/managers. The workflow planner determines the computational tasks that can be scheduled at each stage(s) of the workflow, and once computational tasks are identified, appropriate metadata describing the tasks (including application hints about complexities, data dependencies, affinities, etc.) is inserted into Comet space and the autonomic schedule is notified. The task monitors/manager then monitor and manage the execution of each of these tasks and determines when a stage has completed so the next stage(s) can be initiated.

Estimators: The cost estimators are responsible for translating hints about computational complexity provided by the application into runtime and/or cost estimates on a specific resource. For example, on a Cloud resource such as time/cost estimate may be based on historical data related to specific VM configurations or simple model. A similar approach can also be used for resources on the HPC Grids using specifications of the HPC Grids nodes. However, in the HPC Grids case, waiting time in the batch queue must also be estimated. Tools such as BQP [15], which estimates the probability distribution for the wait times for a job request with a specific size and duration. We use this tools in the autonomic management of the EnKF workflow as

described in the following section.

Autonomic Scheduler: The autonomic scheduler performs key autonomic management tasks. First, it uses application hints to estimate the relative complexities of the tasks to be scheduled and clusters these to identify potential scheduling blocks. It then uses the estimator modules to compute anticipated runtimes for these blocks on available resource classes, and to determine the initial hybrid mix HPC Grids/Cloud resources based on user/system-defined objectives, policies and constraints, e.g., HPC Grids allocations, available budgets, desired QoS, etc., which it communicates to the relevant agents. The autonomic scheduler also communicates resource-class-specific scheduling policies to the agents. For example, tasks that are computationally intensive may be more suitable for a HPC Grids resource, while tasks that require quick turn-around may be more suitable for a Cloud resource. Note that the allocation as well as the scheduling policy can change at runtime.

Grid/Cloud Agents: The Grid/Cloud agents are responsible for provisioning the resources on their specific platforms, configuring *workers* as execution agents on these resources, and appropriately assigning tasks to these workers. The primary task assignment approach supported by the framework is *pull-based*, where workers directly pull tasks from the Comet space based on directives from respective agents, for example, a worker on a Cloud VM may only pull task with computational/memory requirements below some threshold. However this model can be implemented in different ways. For example, on a Cloud, workers on provisioned VMs can directly pull tasks from the space. However, on a typical HPC Grids with a batch queuing system, direct pull may not be possible. In this case a combined push-pull model is used. Specifically, we insert smart "pilot-jobs" [16] containing workers into the batch queues of the HPC Grids systems, which then pull tasks from the Comet space when they are scheduled to run by the queuing system. The smart is a reference to the fact that the pilot-jobs can use local policy and be driven by overall objective to determine the best tasks to take-on. This allows the binding of task to HPC Grids nodes to be delayed allowing the overall scheduling to be more flexible, as well as being able to accommodate any changes in policy or objectives.

IV. An Experimental Investigation of HPC Grids-Cloud Hybrid Usage Modes

A. Autonomic execution of EnKF

The autonomic execution of the EnKF workflow on a hybrid HPC Grid-Cloud environment, and specifically on an combination of the TG (the Ranger system) and the Amazon EC2, using the CometCloud-based infrastructure described above proceeds as follows. At each state of the workflow, the workflow manager determines the number of ensemble members at the stage as well as relative computational complexity of each member. It then encapsulates each ensemble member as a task and inserts it into the Comet space. Note that both, the complexity of each ensemble member as well as the number of ensemble members in a stage can vary making the application naturally heterogeneous.

Due to the complexity of the workload, we employ a 2-stage push-pull model and for both stages, there are dynamic decisions that are made. Specifically, in the first-stage, there is a decision to be made about how many workers should be employed and how to distribute these workers to a possibly varying/different number of workers.

Once the tasks to be scheduled within a stage have been identified, the autonomic scheduler analyzes the tasks and their complexities to determine the appropriate mix of TG and EC2 resources that should be provisioned. This is achieved by (1) clustering tasks based on their complexities to generate blocks of tasks for scheduling, (2) estimating the runtime of each block on the available resources using the cost estimator service and (3) determining the allocations as well as scheduling policies for the TG and EC2 based on runtime estimates as well as overall objectives and resource specific policies and constraints (e.g., budgets). In case of the EnKF workflow, the cost estimation consists of a simple function obtained using a priori experimentation, which maps the computational complexity of the task as provided by the application and estimated runtime. Additionally, in case of the TG, the BQP [15] service is used to obtain estimates of queue waiting times and to select appropriate size and runtime duration of the TG request (which are the major determinants of overall queue wait-time).

Once the resource allocations on the TG and EC2 have been determined, the desired resources are provisioned and "ensemble-workers" are started. On the EC2, this consists of launching appropriate VMs with loading custom images. On the TG, ensemble-workers are essentially "pilot jobs" [16] that are inserted into the batch queue.

Once these ensemble-workers start executing, they can directly access the Comet space and retrieve tasks from the space based on the enforced scheduling policy. The policy we employ is simple: TG 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 TG 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 TG compute nodes. While this policy is not optimal, it was sufficient for our study.

During the execution, the workflow manager monitors the executions of the tasks (using the task monitor) to determine progress and to orchestrate the execution of the overall workflow. The autonomic scheduler also monitors the state of the Comet space as well as the status of

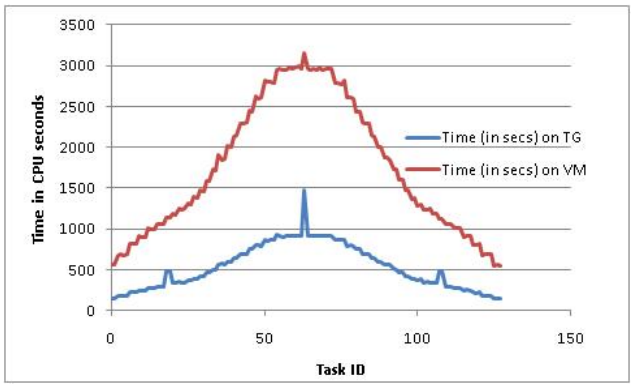

Fig. 3: The distribution of runtimes of ensemble members (tasks) on 1 node (16 processors) of a TG compute system (Ranger) and one VM on EC2.

the resources (using the agents), and determines progress to ensure that the scheduling objectives and policies/constraints are being satisfied, and can dynamically change the resources allocation and scheduling policy as required. Specific implementations of policy and variations of objectives form the basis of our experiments. For example, if the allocation on the TG is not sufficient, the scheduler may increase the number of EC2 nodes allocated on-the-fly to compensate, and modify the scheduling policy accordingly. Such a scenario is illustrated in the experiments below.

B. Experiment Background and Set-Up

The goal of the experiments presented in this section is to investigate how possible usage modes for hybrid HPC Grids-Cloud infrastructure can be supported by a simple policy-based autonomic scheduler. Specifically, we investigate experimentally, implementations of three usage modes – acceleration, conservation and resilience, which are the different objectives of the autonomic scheduler.

Our experiments use a single stage EnKF workflow with 128 ensemble members (tasks) with heterogeneous computational requirement. The heterogeneity is illustrated in Figure 3; which is a histogram of the runtimes of the 128 ensemble members within a stage on 1 node of a TG compute system (Ranger), and 1 EC2 core (a small VM instance, 1.7 GB memory, 1 virtual core,160 GB instance storage, 32-bit platform) respectively. The distribution of tasks is almost Gaussian, with a few significant exceptions. These plots also demonstrate the relative computational capabilities of the two platforms. Note that when a task is assigned to a TG compute node, it runs as a parallel application across the node's 16 cores with linear scaling. However on an EC2 node, it runs as a sequential simulation, which (obviously) will run for longer.

We use two key metrics in our experiments: *Total Time to Completion (TTC)*, which is the wall-clock time for the entire (1-stage) EnKF workflow (i.e., all the 128 ensemble members) to complete and the results are consumed by the KF stage, and may include both TG and EC2 execution. The

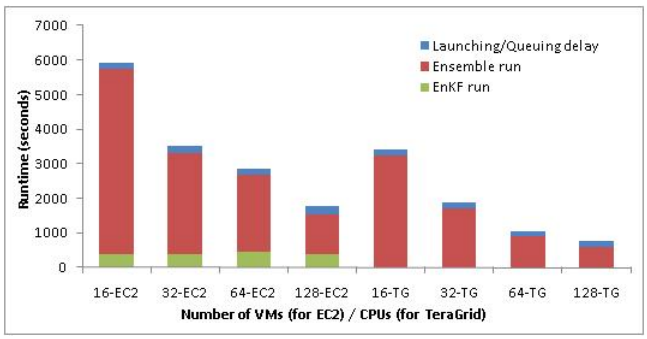

Fig. 4: Baseline TTC for EC2 and TG for a 1-stage, 128 ensemble-member EnKF run. The first 4 bars represent the TTC as the number of EC2 VMs increase; the next 4 bars represent the TTC as the number of CPUs (nodes) used increases.

Total Cost of Completion (TCC), which is the total EC2 cost for the entire EnKF workflow.

Our experiments are based on the assumption that for tasks that can use 16-way parallelism, the TG is the platform of choice for the application, and gives the best performance, but is also the relatively more restricted resource. Furthermore, users have fixed allocation on this expensive resource, which they might want to conserve for tasks that require greater node counts. On the other hand, the EC2 is a relatively more freely available, but is not as capable.

Note that the motivation of our experiments is to understand each of the usage scenarios and their feasibility, behaviors and benefits, and not to optimize the performance of any one scenario (or experiment). In other words, we are trying to establish a proof-of-concept, rather than a systematic performance analysis.

C. Establishing Baseline Performance

The goal of the first set of experiments is to establish a performance baseline for the two platforms considered. The TTC for a 1-stage, 128 ensemble-member EnKF run and for different numbers of VMs on the EC2 and different number of CPUs on the TG are plotted in Figure 4. We can see from the plots that the TTC decreases essentially linearly as the number of EC2 VMs and the the number of TG CPUs increases. The TTC has 3 components – the time that it takes to generate the ensemble-members, the VM start-up time in case of the EC2 or the TG queuing time, and the dominant run-time of the ensemble-members. In case of EC2, the VM start-up is about 160 seconds and remains constant in this experiment, which is because, the VMs are launched in parallel. Note that, in general, there can be large variability, both in the launching times as well as performance of EC2 VM instances. In case of the TG, the queuing time was only about 170 seconds (essentially the best case scenario) since we used the development queue for these experiments. The time required for ensemble member generation also remained constant, and is a small fraction of the TTC. Another important observation is that the TTC on the TG is consistently lower than on EC2 (as expected).

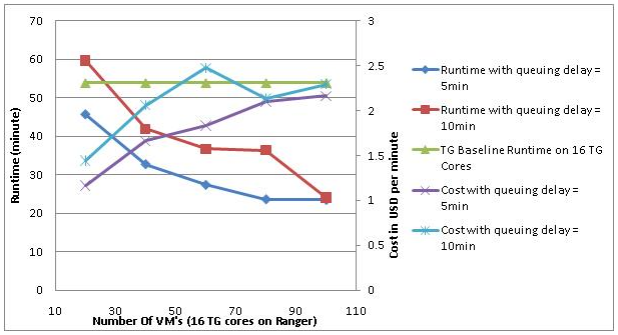

Fig. 5: The TTC and TCC for Objective I with 16 TG CPUs and queuing times set to 5 and 10 minutes. As expected, more the number of VMs that are made available, the greater the acceleration, i.e., lower the TTC. The reduction in TTC is roughly linear, but is not perfectly so, because of a complex interplay between the tasks in the work load and resource availability.

D. Objective I: Using Clouds as *Accelerators* for HPC Grids

In this usage scenario, we explore how Clouds can be used as accelerators for HPC Grid work-loads. Specifically, we explore how EC2 can be used to accelerate an application running on the TG. The workflow manager inserts ensemble tasks into the Comet space and ensemble workers, both on the TG and EC2, pull tasks based on the defined policy described above and execute them.

In these experiments, we used 16 TG CPUs (1 node on Ranger) and varied the number of EC2 nodes from 20 to 100 in steps of 20. The average queuing time was set to 5 and 10 minutes. These values can be conservatively considered as the typical wait time for a job of this size, on the normal production queue of the TG [15]. The VM start up time on EC2 was once again about 160 seconds. The resulting TTC for hybrid usage mode are plotted in Figure 5. The plots also show the *best case* TG TTC for 16 CPUs (1 TG compute node) using the development queue and the 170 second wait time. The plots clearly show acceleration for both wait times – that is the hybrid-mode TTC is lower than the TTC when only the TG is used. The exception is the experiment with 20 VMs and 10 minute wait time, where we see a small slow down that is due to the fact the some long-running task get scheduled onto the EC2 nodes causing the TG nodes to be starved. Another interesting observation is that acceleration is greater for the 5 minute wait time as compared to the 10 minute wait time. This is once again because when the queuing delay is 10 minutes, fewer tasks are executed by the more powerful TG CPUs. Also note that for the 100 VM case, the TTC is the same for both, the 5 minute and 10 minute wait times because in these cases most of the tasks are consumed by EC2 nodes.

These observations clearly indicate that the acceleration achieved is sensitive to the relative performance of HPC Grids and cloud resources and the number of HPC Grid resources used and the queuing time. For example, when we increased the number of CPUs to 64 and used a 5 minute queuing time, no acceleration was observed as TG CPUs are significantly faster than the EC2 nodes and 64 TG CPUs are capable of finishing all the ensemble tasks before the EC2 nodes can finish *any* task.

CPU-Time Limit (Mins)	25	50	100	200	300
TG (Tasks)	1	3	6	14	19
EC2 (Tasks)	127	125	122	115	109
EC2 (Nodes (VMs))	90	88	85	78	74
EC2 (TTC (Mins.))	28.94	28.57	27.83	26.48	26.10
EC2 (TCC (USD))	2.22	2.18	2.05	2.00	1.94

TABLE I: Distribution of tasks across EC2 and TG, TTC and TCC, as the CPU-minute allocation on the TG is increased.

Note that the figure also presents the TCC for each case. This represents the actual cost as function of the CPU time used; the billing time may be different. For example, on the EC2, the granularity for billing is CPU-hours so the billed cost can be higher.

It is important to understand that the acceleration arising from the ability to utilize Clouds and Grids is not just a case of "throwing more resources" at the problem, but rather demonstrates how two different resource types and underlying resource management paradigms can complement one another and results in a lowering of the overall TTC.

Objective II: Using Clouds for *Conserving* CPU-Time on the TeraGrid

When using specialized and expensive HPC Grid resources such as the TG, one often encounters the situation that a research group has a fixed allocation for scientific exploration/computational experiment, and typically this is in the form of a fixed allocation of the number of CPU-hours on a machine. The question then is, can one use Cloud resources to offload tasks that perhaps don't need the specialized capabilities of the HPC Grid resource to conserve such an allocation, and what is the impact of such an offloading on the TTC and TCC. Note that one could also look at conservation from the other side and conserve expenditure on the Cloud resources based on a budget, but here we investigate just the former.

Specifically, in the experiments in this scenario, we constraint the number of CPU-minutes assigned to a run and use EC2 nodes to compensate and enable the application to run to completion. The aim is to determine the number of tasks as a consequence of the constraint, that will run on either the TG, and what number will be taken up by EC2 *when attempting the quickest solution possible*, and what is the impact on the TTC and TCC. We repeat the experiment increasing the number of CPU-minutes assigned. The results of the experiments are presented in Table I.

Interestingly, given that there several tasks that take less than 1 minute on 1 compute node on the TG (which requires

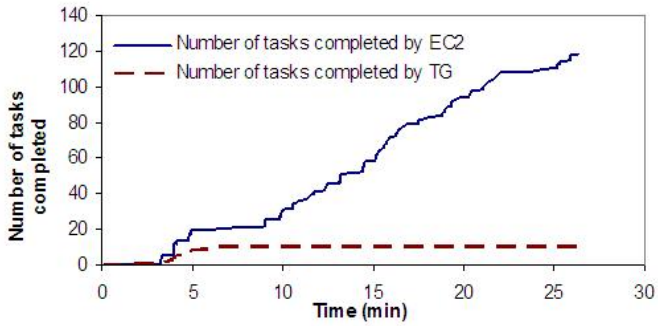

Fig. 6: Allocation of tasks to TG CPUs and EC2 nodes for usage mode III. As the 16 allocated TG CPUs become unavailable after only 70 minutes rather than the planned 800 minutes, the bulk of the tasks are completed by EC2 nodes.

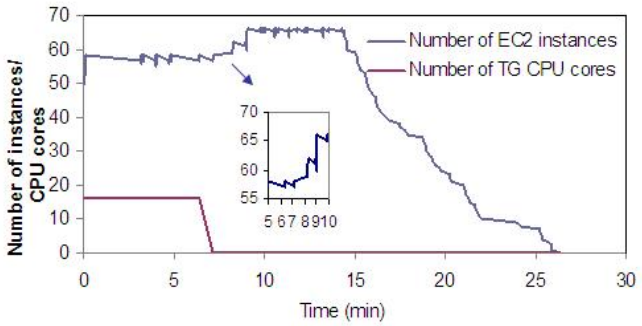

Fig. 7: Number of TG cores and EC2 nodes as a function of time for usage mode III. Note that the TG CPU allocation goes to zero after about 70 minutes causing the autonomic scheduler to increase the EC2 nodes by 8.

16 CPU-mins, the unit of allocation), when the number of CPU-minute allocated is 25 mins, the TG gets exactly 1 task, and as we increase the allocation, the number of task pulled by TG CPUs increases in increments of 1 task per 16 CPU-minute (or 1 node-minute).

These performance figures provide the basis for determining the sensitivity (of a given workload) to maximum CPU-mins (on TG). For example, we show that a factor of 12 decrease in max CPU-minutes can (300 reduced to 25), thanks to hybrid mode, lead to only a 10% increase in TTC. (26.10 to 28.94)

Objective III: Response to Changing Operating Conditions (*Resilience*)

Usage mode III, or *resilience*, addresses the situation where resources that were initially planned for, become unavailable at runtime, either in part or in entirety. For example, on several TG machines, there are instances when jobs with a higher-priority have to be given priority (i.e. right-of-way [17]), or perhaps it could just be that there is some unscheduled maintenance needed. Typically, when such situations arise, applications either just wait (significantly) longer in the queue or may be unceremoniously terminated and have to be rerun. The objective of this usage mode is to investigate how Cloud services can be used to address such situations and allow the system/application to respond to a dynamic change in availability of resources.

This scenario also address another situation that is becoming more relevant as applications have more complex and dynamic. For such applications, it is often not possible to accurately estimate the CPU-time required to run to completion, and users either have to rely on trial and error and allow several jobs to terminate due to inadequate CPU-time, or overestimate the required time and request more time that is actually needed. Both options lead to poor utilization. The *resilience* usage mode can also be applied to this scenario, where a shortfall in requested CPU-time can be compensated using Cloud resources.

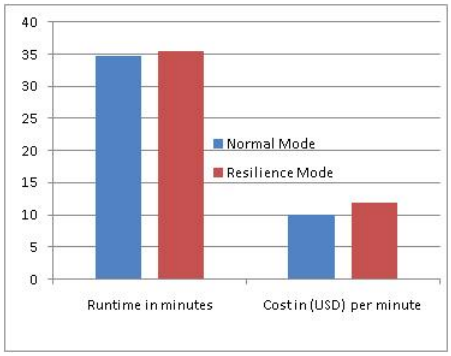

Fig. 8: Overheads of resilience on TTC and TCC.

In the set of experiments conducted for this usage mode, we start by requesting an adequate amount of TG CPU-time. However, at runtime, we trigger an event indicating that the available CPU-time on the TG has changed, causing the autonomic scheduler to re-plan and reschedule. Specifically, in our experiments, the autonomic scheduler begins by requesting 16 TG CPUs for 800 minutes. However, after about 50 minutes of execution (i.e., 3 Tasks were completed on the TG), the scheduler is notified that only 20 CPU-minutes remain, causing it to re-plan and as a result increase the allocation of EC2 nodes to maintain acceptable TTC.

The results of the experiments are plotted in Figures 6– 8. Figure 6 shows the cumulative number of tasks completed by TG and EC2 over time. As per the original plan, the expected distribution of tasks was 63:65 for TG:EC2. However, this distribution changes as the TG resource becomes unavailable, see in Fig. 7, causing EC2 nodes to take up a much large proportion of the tasks. Based on the scheduling policy used, the autonomic scheduler decided that the best TTC will be achieved by increasing the number of EC2 nodes by 8, from 58 originally allocated to 64. The resulting TTC and TCC are plotted in Figures 8.

As is probably obvious, the ability to support resilience is a first-step in the direction towards graceful fault-tolerance, but we will discuss fault-tolerance in a separate work.

V. Conclusion and Future Work

Given that production computational infrastructures will soon provide a hybrid computing environment that integrates traditional HPC Grid services with on-demand Cloud services, understanding the potential usage modes of such hybrid infrastructures is important. In this paper, we experimentally investigated, from an application's perspective, possible usage modes for integrating HPC Grids and Clouds as well as how autonomic computing can support these modes. Specifically, we used an EnKF workflow with the CometCloud autonomic Cloud engine on a hybrid platform, to investigate 3 usage modes (i.e., autonomic objectives) – acceleration, conservation and resilience. Note that our objective in the experiments presented was to understand each of the usage scenarios and their feasibility, behaviors and benefits, and not to optimize the performance of any one scenario (or experiment).

The results of experiments demonstrate that a policy driven autonomic substrate, such as the autonomic application workflow management framework and *pull-based* scheduling supported by CometCloud, can effectively support such the usage modes (objectives) investigated here. We show that the desired objectives studied are both feasible and beneficial. The approach and infrastructure used, can manage the heterogeneity and dynamics in the behaviors and performance of the platforms (i.e., EC2 and TG). Our results also demonstrated that defining appropriate policies that govern the specifics of the integration is critical to realizing the benefits of the integration (and can be non-trivial to formulate).

We note that the scenarios presented in this paper are in no way comprehensive, nor are the experiments complete. In fact, our work only begins to scratch the surface in examining the role that autonomics can play in the effective integration of distinct infrastructure paradigms with very different capabilities, QoS offerings and economic considerations. Some immediate extensions to this work that are already underway include more detailed experimentation focusing on better understanding the various parameters, their inter-dependencies, correlations, as well as a sensitivity analysis. Of immediate interest is the variation of QoS in EC2 nodes that manifested itself in abnormally long start-up times and skewed experimental results.

It will be interesting to extend the concepts and understanding gained from this work to high-throughput Grids, where typically, the individual resources involved will have similar performance to EC2, than HPC Grids used here. Additionally the objectives for high-throughput Grids would also be different i.e., maximizing the number of tasks, and so would be the policies required to support the objective.

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. SJ and MP would like to the e-Science Institute for supporting the Research theme on Distributed Programming Abstractions.

References

[1] H. Kim, M. Parashar, L. Yang, and D. Foran, "Investigating the use of cloudbursts for high throughput medical image registration," in *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (Grid 2009)*, 2009.

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

[3] B. Sotomayor, K. Keahey, and I. Foster, "Combining batch execution and leasing using virtual machines," in *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, 2008, pp. 87–96.

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

[5] R. E. Kalman, "A new approach to linear filtering and prediction problems." [Online]. Available: http://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf

[6] Y. Gu and D. S. Oliver, "An iterative ensemble kalman filter for multiphase fluid flow data assimilation," *SPE Journal*, vol. 12, no. 4, pp. 438–446, 2007.

[7] 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.

[8] Y. Gu and D. S. Oliver, "The ensemble kalman filter for continuous updating of reservoir simulation models," *Journal of Engineering Resources Technology*, vol. 128, no. 1, pp. 79–87, 2006.

[9] Cactus Framework. [Online]. Available: {http://www.cactuscode.org}

[10] 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, http://www.mcs.anl.gov/petsc.

[11] Z. Li and M. Parashar, "A computational infrastructure for grid-based asynchronous parallel applications," in *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2007, pp. 229–230.

[12] 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*, 2001, pp. 149–160.

[13] C. Schmidt and M. Parashar, "Squid: Enabling search in dht-based systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 7, pp. 962–975, 2008.

[14] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[15] J. Brevik, D. Nurmi, and R. Wolski, "Predicting bounds on queuing delay for batch-scheduled parallel machines," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 110–118.

[16] Thain D, Tannenbaum T and Livny M 2005 Distributed Computing in Practice: The Condor Experience Concurrency - Practice and Experience 17 2-4 323-56.

[17] "SPRUCE: Special PRiority and Urgent Computing Environment," http://spruce.teragrid.org/.