SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems

André Luckow[1], Lukasz Lacinski[1], Shantenu Jha[1,2,3,*],
[1]*Center for Computation & Technology, Louisiana State University, USA*
[2]*Department of Computer Science, Louisiana State University, USA*
[3]*e-Science Institute, Edinburgh, UK*
[*]*Contact Author:* sjha@cct.lsu.edu

Abstract

The uptake of distributed infrastructures by scientific applications has been limited by the availability of extensible, pervasive and simple-to-use abstractions which are required at multiple levels – development, deployment and execution stages of scientific applications. The Pilot-Job abstraction has been shown to be an effective abstraction to address many requirements of scientific applications. Specifically, Pilot-Jobs support the decoupling of workload submission from resource assignment; this results in a flexible execution model, which in turn enables the distributed scale-out of applications on multiple and possibly heterogeneous resources. Most Pilot-Job implementations however, are tied to a specific infrastructure. In this paper, we describe the design and implementation of a SAGA-based Pilot-Job, which supports a wide range of application types, and is usable over a broad range of infrastructures, i.e., it is general-purpose and extensible, and as we will argue is also interoperable with Clouds. We discuss how the SAGA-based Pilot-Job is used for different application types and supports the concurrent usage across multiple heterogeneous distributed infrastructure, including concurrent usage across Clouds and traditional Grids/Clusters. Further, we show how Pilot-Jobs can help to support dynamic execution models and thus, introduce new opportunities for distributed applications. We also demonstrate for the first time that we are aware of, the use of multiple Pilot-Job implementations to solve the same problem; specifically, we use the SAGA-based Pilot-Job on high-end resources such as the TeraGrid and the native Condor Pilot-Job (Glide-in) on Condor resources. Importantly both are invoked via the same interface without changes at the development or deployment level, but only an execution (run-time) decision.

I. Introduction

Distributed infrastructures have been used by many applications to advance understanding in their disciplines. The requirements and characteristics of applications that motivate the usage of distributed infrastructures are very broad, and most often differ from regular, HPC applications in several fundamental ways. Distributed applications often need to be designed for more than simple peak-utilization, e.g., the number of coupled tasks that need to be completed within a time window. Equally important, distributed applications have a much broader range of usage modes. Production Grid Infrastructures (PGIs) as well as the Programming Systems and Tools (PS&T) used to develop distributed applications need to address these and other *fundamental* distributed application characteristics [1].

It is generally accepted that the design and development of distributed applications is a difficult undertaking made somewhat more difficult by the state of distributed infrastructure available to end-users [2]. For example, many programming systems and tools for distributed applications are either incomplete and/or often out-of-phase with requirements, or simply incomplete or inflexible with respect to application needs, e.g. tools that support the master-worker paradigm often only address failures of workers and not of the master. Additionally, tools and development systems often don't support the specific usage modes that may be required for a certain application scenario, with the level of robustness and scalability required, i.e., solutions work well in small or controlled environments, but not at-scale. These and other concerns have motivated developers to "roll out their own" capabilities at one or more levels – application, programming system, middleware, and/or infrastructure level [1], [2] – in turn further adding to an existing large range of tools, programming systems and environments and adding to challenges of providing

interoperability. Thus to the extent possible, PS&T should support all of the following properties: (i) new application domains and usage-modes, (ii) extending the functionality supported, (iii) extension to new infrastructures, (iv) extend across scales of operation, (v) uptake by communities other than the developer (community usage) and, (vi) reuse and support patterns and abstractions for distributed computing.

The extent to which the above design objectives will succeed depends not only on the resulting programming system, but also on the availability of usable and extendable abstractions and their suitability for given production infrastructures. Interestingly, the Pilot-Job abstraction has been widely used across several different PGIs. However, the existing Pilot-Job frameworks are all heavily customized and often tightly coupled to a specific infrastructure, and not extensible or usable across different systems, e.g. there is no such "unifying" and "extensible" tools on the TeraGrid that supports a range of application types and characteristics.

Recently, the usage of virtualization and of on-demand virtual machines has become increasingly popular. These so-called infrastructure-as-a-service Clouds have different advantages compared to traditional Grid systems: users are provided with greater flexibility and have the ability to customize their virtual machine environment. More and more, the need to integrate traditional Grids and Clouds arises. Developing and running applications in such a hybrid and dynamic computational infrastructure presents new and significant challenges [3]. These include the need for programming systems that can express and support the hybrid usage modes, 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, budgets etc.

SAGA is a programming system that provides a high-level, easy-to-use API for accessing distributed resources. SAGA has traditionally been used to design and develop end-user applications, but it is also commonly utilized to develop tools and infrastructures [4] to support distributed applications. In a nutshell it provides the building blocks to develop the abstractions and functionalities to support the characteristics required by distributed applications whether directly, or as tools in interoperable and extensible fashion. The SAGA-based Pilot-Job (BigJob) is such an example; unlike other common Pilot-Job systems, SAGA BigJob natively supports MPI jobs and works on a variety of back-end systems, generally reflecting the advantage of using a SAGA-based approach. But in addition, to the above and as we will show, it has been designed to be extensible, e.g., to support policy-driven pilot-jobs.

The paper is outlined as follows: We will begin the next section with a quick discussion of Pilot-Jobs and some related and frequently used concepts in this paper. We will then present SAGA and outline how it is used to support distributed application developments at multiple levels. In Section IV, we introduce the SAGA-based Pilot-Job – BigJob and discuss its architecture and working over three different backend infrastructures. Although the focus of this paper is on outline the design principles and objectives of the SAGA-based Pilot-Job and experiments to understand its performance, we will present the results of preliminary experiments that exploit the dynamic execution models and form the basis for future work on sophisticated autonomic resource management and allocation.

II. Pilot-Jobs: Concepts and Related Work

Pilot-Jobs are a useful abstraction for efficiently executing an ensemble of batch jobs without the necessity to queue each individual job. The Pilot-Job itself is a regular Grid job, which is started through a Grid resource manager, such as the Globus GRAM. Once the batch queue assigned the requested resources to the Pilot-Job, the Pilot-Job circumvents the necessity to queue each individual sub-job and is responsible for managing these and assigning them to so-called sub-jobs. That way queuing times for sub-jobs can be reduced and the predictability for application scenarios can be increased. Pilot-Jobs decouple resource allocation from resource binding and allow the efficient utilization of resources. By delaying the resource binding decision into the application-level, dynamic usage modes, e. g. the load-depended sizing of sub-jobs or the dynamic addition of resources to meet deadlines, can be supported.

Pilot-Jobs are an execution abstraction that has been used by many communities to increase the predictability and time-to-solution of such applications. Pilot-Jobs have been used to (i) improve the utilization of resources, (ii) to reduce the net wait time of a collection of tasks, (iii) facilitate bulk or high-throughput simulations where multiple jobs need to be submitted which would otherwise saturate the queuing system, and (iv) as a basis to implement application specific scheduling decisions and policy decisions. For example, on the LHCb computing model, Grid jobs are routed through the DIRAC [5] workload management system (WMS). DIRAC is a pilot-based system where user jobs are queued in the WMS server and the server submits generic pilot scripts to the Grid. Each pilot queries the WMS for a job with resource requirements satisfied by the machine where the pilot script is running. If a compatible job is available, it is pulled from the WMS

and started. Otherwise, the pilot terminates and the WMS sends a new pilot to the Grid. There can be additional functionality that builds upon a pilot-based scheme; for example DIANE [6] a lightweight agent-based scheduling layer can use DIRAC.

As more applications take advantage of dynamic execution, the Pilot-Job concept has grown in popularity and has been extensively researched and implemented for different usage scenarios and infrastructure. Using Condor Glide-In a complete Condor pool can be initiated using the GRAM service; Falkon [7] emphasizes the performance of its task dispatcher. However, both systems have limitations and impose different overheads: most significantly Condor Glide-In [8] requires the start of a complete set of Condor daemons (i.e., Condor must be running) on the set of resources. Falkon does not support MPI applications, and thus not usable for our application scenario.

Nimbus [9] provides a Pilot-Job like abstractions for Clouds. For this purpose, Nimbus allows the launch of auto-configured virtual machine clusters that contain a Torque and Globus installation. The Atlas computing framework developed at CERN also heavily relies on the PanDA Pilot-Job framework to implement resource leases. Using the VIRM API, PanDA was extended to support different virtualization backends, e. g. OpenNebula and Nimbus. However, in contrast to the SAGA-based Pilot Job, both frameworks are strongly coupled to a particular backend infrastructure – Globus in the case of Nimbus and gLite in the case of PanDA. The advantage of the SAGA Pilot-Job is that it allows applications to seamlessly utilize different backend infrastructure, e. g. Condor, Globus and different kinds of Clouds, at the same time.

III. SAGA and SAGA-based Frameworks for Large-Scale and Distributed Computation

SAGA [10] provides a simple, POSIX-style API to the most common Grid functions at a sufficiently high-level of abstraction so as to be independent of the diverse and dynamic Grid environments. The SAGA specification defines interfaces for the most common Grid-programming functions grouped as a set of functional packages (Fig. 1). Some key packages are:

- File package - provides methods for accessing local and remote filesystems, browsing directories, moving, copying, and deleting files, setting access permissions, as well as zero-copy reading and writing.
- Job package - provides methods for describing, submitting, monitoring, and controlling local and remote jobs. Many parts of this package were derived from the largely adopted DRMAA specification.
- Other Packages, such as the RPC (remote procedure call), Replica package and Stream Package.

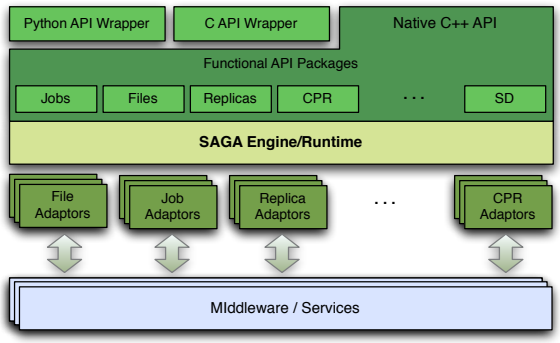

Fig. 1: Layered schematic of the different components of the SAGA landscape. At the topmost level is the simple integrated API which provides the basic functionality for distributed computing. Our BigJob abstraction is built upon this SAGA layer using Python API bindings.

In the absence of a formal theoretical taxonomy of distributed applications, Fig. 2 can act a guide. Using this classification system, there are three types of distributed applications: (i) Applications where local functionality is swapped for distributed functionality, or where distributed execution modes are provided. A simple but illustrative example is an application that uses distributed resources for bulk submission. Here, the application remains unchanged and even unaware of its distributed execution, and the staging, coordination, and management are done by external tools or agents. Most applications in this category are classified as implicitly distributed. (ii) Applications that are naturally decomposable or have multiple components are then aggregated or coordinated by some unifying or explicit mechanism. DAG-based workflows are probably the most common example of applications in this category. Finally, (iii) applications that are developed using frameworks, where a framework is a generic name for a development tool that supports specific application characteristics (e.g. hierarchical job submission), and recurring patterns (e.g. MapReduce, data parallelism) and system functionality. Lazarus [12] provides several autonomic features, such as fault tolerance and dynamic resource selection, specifically for Ensemble-Kalman-Filter application scenarios. SAGA has been used to develop system-level tools and applications for each of these types.

It is important to note that SAGA provides the basic API to implement distributed functionality required by applications (typically used directly by the first category of applications), and is also used to implement higher-level APIs, abstractions, and frameworks that, in turn, support the development, deployment and execution of distributed applications [12]. Merzky et al. [13] discusses how SAGA was used to implement a higher-level API to support workflows. In this paper, we will discuss how SAGA can

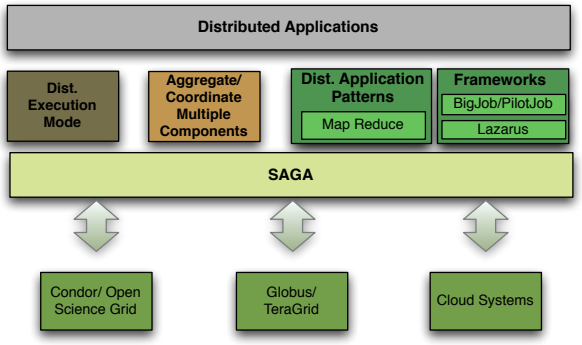

Fig. 2: Showing the ways in which SAGA can be used to develop distributed applications. The different shaded box represent the three different types; frameworks in turn can capture either common patterns or common application requirements/characteristics.

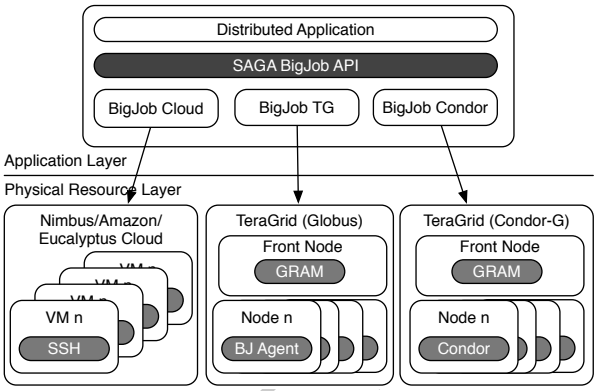

Fig. 3: **Overview of the SAGA-based Pilot Job:** The SAGA Pilot Job API is currently implemented by three different backends - one for Grids, Condor and for Clouds.

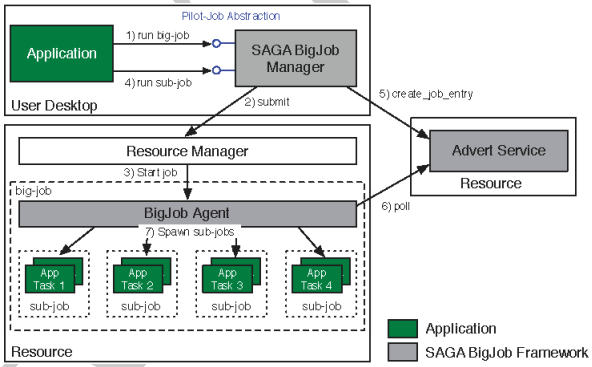

Fig. 4: BigJob Architecture: The core of the framework, the BigJob Manager, orchestrates a set of sub-jobs via a BigJob Agent using the SAGA job and file APIs. The BigJob Agent is responsible for managing and monitoring sub-jobs.

be used to implement runtime frameworks to support the efficient execution of the distributed applications.

IV. BigJob: SAGA-based Pilot-Job Framework

BigJob is a SAGA-based Pilot-Job implementation. In contrast to other Pilot-Job implementations, e.g., Falkon, BigJob natively supports parallel applications (e.g. based on MPI) and works independent of the underlying Grid infrastructure across different heterogeneous backend, e. g. Grids and Cloud, reflecting the advantage of using a SAGA-based approach. Further, the framework is extensible and provides several hooks that can be used to support other resource types and Pilot-Job frameworks.

As shown in Figure 3, BigJob currently provides a unified abstraction to Grids, Condor pools and Clouds. Using the same API, applications can dynamically allocate resources via the big-job interface and bind sub-jobs to these resources. In the next sections, we describe how SAGA interfaces to three different backends, while exposing the same user-level BigJob interface and semantics. A tutorial that describes the BigJob API can be found at [**?**].

A. BigJob for Grids

Figure 4 shows an overview of the SAGA BigJob implementation for computational Grids. The Grid BigJob comprises of three components: (i) the BigJob Manager that provides the Pilot-Job abstraction and manages the orchestration and scheduling of BigJobs (which in turn allows the management of both big-job objects and sub-jobs), (ii) the BigJob Agent that represents the pilot job and thus, the application-level resource manager on the respective resource, and (iii) advert service which is used for communication between the BigJob Manager and Agent.

Applications can utilize the framework via the big-job and sub-job classes. Both interfaces are syntactically and semantically consistent with the other SAGA APIs: a sub-job e. g. is described using the standard SAGA job description; job states are expressed using the SAGA state model. Before running sub-jobs, an application must initialize a big-job object. The BigJob Manager then queues a job, which actually runs a BigJob Agent on the respective remote resource. For this agent a specified number of resources is requested. Subsequently, sub-jobs can be submitted through the BigJob Manager using the job-id of the BigJob as reference. The BigJob Manager ensures that sub-jobs are launched onto the correct resource based upon the specified job-id using the right number of processes.

The Grid BigJob is a SAGA-based framework and heavily utilizes the SAGA job, file and advert APIs. The BigJob Agent is started using the SAGA job API. Communication between the BigJob Agent and BigJob Manager is carried out using the SAGA advert service, a central key/value store. For each new job, an advert entry is created by the BigJob Manager. The agent periodically polls for new jobs.

If a new job is found and resources are available, the job is dispatched, otherwise it is queued.

In this paper we will use BigJob to support HPC applications, however, BigJob has also been used to support high-throughput runs of HPC applications on the TeraGrid. Specifically, El-Khamra et al. [12] uses BigJob to support a very large number of ensembles with highly variable run-times, as well as heterogenous resource requirements. In Luckow et al. [14] although an efficient scale-out behaviour was established, all resources utilized were TeraGrid resources, and thus accessible using a uniform Globus interface. Another use cases that have used BigJob on Grid resources include applications involving parameter sweeps and which require the execution of a very large number of simulations in parallel. Having outlined the general principles and design objectives of the BigJob framework, and established its use on production grid infrastructure for a range of applications, we will focus on extending its capabilities to work with Clouds and other distributed infrastructure.

B. BigJob and Condor Glide-in

Condor provides a native Pilot-Job functionality (Condor Glide-in); in fact, Condor has been instrumental in the uptake and popularization of the Pilot-Job concept. The challenge here is to implement BigJob as an overlay over Condor's native Pilot-Job with minimal disruption. However the approach presented here shows how to implement the BigJob idea referring Condor only.

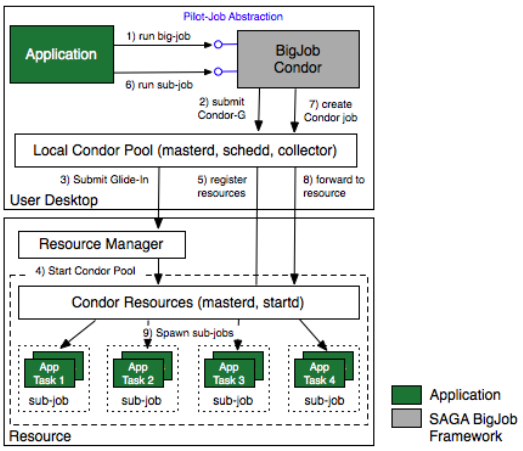

Fig. 5: Schematic showing how the BigJob interface is used to submit jobs to a Condor Glide-in without any changes in semantics or usage mode of the Condor Glide-in. The application submits the sub-jobs (which are the 8 replicas) to a BigJob object, which then submits to the available condor pool. The Condor Glide-in then takes responsibility of managing the resources that constitute the pool and distributing sub-jobs to them.

Figure 5 illustrates the architecture of the Condor BigJob. A Condor Pilot-Job from an application point of view is a request for a set of resources that can later be accessed via the local Condor pool. To initiate a Pilot-Job, the application must specify a GRAM endpoint of a Grid resource, a number of requested nodes and a wall time (step 1). The Condor BigJob translates this request to a Condor-G job, which is managed using the SAGA Condor adaptor. Condor is then responsible for submitting the job to the GRAM of the specified remote resource (step 3). Once the Condor-G job becomes active, it start the Condor daemons `master` and `startd` on the remote resources (step 4). During the initialization these daemons register the allocated resources with the Condor pool on the user's desktop creating a coherent computing resource to submit sub-jobs to (step 5). As with the Grid BigJob sub-job objects can be created via the Pilot-Job abstraction. The Condor BigJob maps these sub-jobs to the Condor pool and starts them via the SAGA Condor adaptor (steps 6-8). Also, sub-jobs can already be submitted before any node joins the Condor pool. These jobs will automatically wait until scheduler is able to find any "Unclaimed" resources matching a sub-job's requirements.

The idea of an advert service as a central point of collecting information about sub-jobs can be substituted by a Condor queue. This approach gives some advantages. BigJob can utilize Condor scheduling and matchmaking. It allows the control in an unified way of the state of Pilot-Jobs and sub-jobs as Condor jobs. The advert service still can be applied and is an answer to a problem of central coordination point if many different backends (Condor pool, Clouds, Grids) are used to scale-out.

Thanks to the native capabilities of the Condor Pilot-Job, the Condor BigJob is able to support some enhanced features, e. g. the management of multiple Pilot-Jobs per big-job object or the capability to automatically reschedule jobs to alternative resources in case of failures. The capability to use multiple Pilots-Jobs is in particular useful since the usage of multiple clusters can reduce the time-to-completion T_C for many application. For example, if one of the requested clusters is heavily loaded, the waiting time of the Pilot-Job can be very long. In general, the more resources used, the lower the time until the first Pilot-Job and thus, the first sub-jobs becomes active

C. BigJob for Clouds

At the execution level, Clouds differ from Clusters/-Grids in at least a couple of different ways. In Cloud environments, user-level jobs are not typically exposed to a scheduling system; a user-level job consists of requesting the instantiation of a virtual machine (VM). Virtual machines are either assigned to the user or not (this is an important attribute that provides the illusion of infinite resources). The assignment of job to a VM must be done by the user (or a middleware layer as BigJob). In contrast,

user-level jobs on Grids and clusters are exposed to a scheduling system and are assigned to execute at a later stage. Also a description of a Grid job typically contains an explicit description of the workload; in contrast for Clouds a user level job usually contains the container (description of the resource requested), but does not necessarily include the workload. In other words, the physical resources are not provisioned to the workload but are provisioned to the container. This model is quite similar to resource reservations where you obtain a "container" of resources to which you can later bind jobs to. Interestingly, at this level of formulation, Pilot-Jobs attempt to provide a similar model of resource provisioning as Clouds natively offer. The Cloud usage model naturally maps to the Pilot-Job abstraction where resource allocation is done within the big-job and the resource binding is conducted dynamically for each sub-job.

As shown in Figure 3 the Cloud BigJob implements the Pilot-Job abstraction for Cloud resources. In the following, we discuss the internals of the BigJob Cloud implementation. Initially it is required that the user prepares and uploads a virtual machine image for respective Cloud environment. This image should contain the application and possibly the application data. Once the image is setup, the user can initialize a `bigjob_cloud` object specifying the Cloud environment, the image-id of the prepared VM image as well as the number and type of the VMs.

Having received a request, the BigJob Manager initializes the requested resources, i. e. it start the requested number of virtual machines. In contrast to earlier work [**?**], we do not rely on the SAGA/AWS adaptor for this purpose mainly due to the lack of support for VM clusters and its limited runtime configurability. In particular the efficient support for groups of VMs is crucial for a pilot-job implementation. The Cloud BigJob addresses these limitations. The architecture encapsulates Cloud specifics, such as the used security mechanism, the communication protocol and the required setup tasks, into a Cloud resource adaptor. Currently, Amazon EC2, Eucalyptus and Nimbus Clouds are supported. The resource adaptor communicates with the different Cloud environments via the command line tools provided by the respective Cloud infrastructure. Despite some similarities between these Clouds and the trend toward standardization on the EC2 API, there are subtle differences, e. g. in the allocation of IP addresses, the SSH key handling, the metadata management or the network setup. The Cloud BigJob attempts to hide most of these differences from the user. However, the user is still required to perform some manual steps, such as setting up the VM image and keys, prior to using the Cloud BigJob. A further standardization of such Cloud environments, e. g. within the OCCI efforts [15], is necessary to minimize the necessary steps.

The Cloud BigJob supports basic fault tolerance features: failed requests e. g. are automatically re-executed n times. If not sufficient resources can be allocated, the BigJob changes to the state *failed* and terminates all previously started VMs. Once all requested resource have been successfully allocated, the BigJob manager prepares each virtual machine for further job executions e. g. by setting up all SSH keys, which is a pre-requisite for running MPI applications. Having finished all preparation tasks, the BigJob changes into the state *Running*.

After the running state has been reached, the manager starts to process sub-jobs. For this purpose the SAGA/SSH adaptor is used [**?**]. Unprocessed sub-jobs are stored in a FIFO queue. As resources become available, sub-jobs are assigned to these. For parallel jobs the manager generates a nodefile, which is then used for launching the application. Once the preparation work has been finished, the manager spawns the sub-job using the SAGA job API and the SSH adaptor. Progress of all sub-jobs is monitored by the manager using the standard mechanisms provided by the SAGA job API and a monitoring thread. Once a job terminates, i. e. it changes to the state *Done* or *Failed*, the allocated resources are freed. A new sub-job from the queue can then be assigned to newly available resources.

Since all meta-data of the assigned Cloud VM, in particular the internal and external IPs of the VMs, is available via the Cloud APIs of the supported Cloud environments, in contrast to the Grid BigJob it is not necessary to deploy any agents on the Cloud machines. The state of all VMs can simply be maintained by the manager; remote launching can be done via the SSH protocol.

V. Usage Modes and Analysis

The aim of this section is not to perform detailed systematic performance measures and analysis, but to illustrate the representative usage modes BigJob can support, and to explain how they are supported. We discuss two scenarios that are representative of how applications would typically use multiple distributed infrastructures. We focus our attention on replica-based Molecular Dynamics (MD) simulations and a workload as 8 replicas of a Hepatatis-C virus (HCV) model, each replica running for 500 timesteps.

A. Hepatatis Virus: Replica-based Solution

Several classes of applications are well suited for distributed environments. Probably the best known and most powerful examples are those that involve an ensemble of decoupled tasks, such as simple parameter sweep applications [16]. Here we outline a scientific basis for focussing on an ensemble of (parallel HPC) MD simulations. Ensemble based approaches represent an important and promising

attempt to overcome the general limitations of insufficient time-scales, as well as specific limitations of inadequate conformational sampling arising from kinetic trappings. The fact that one single long-running simulation can be substituted for an ensemble of simulations, make these ideal candidates for distributed environments. This thus provides an important general motivation for researching ways to support scale-out and thus enhance sampling and to thereby increase "effective" time-scales studied.

The physical system we investigate using replica-based techniques is the HCV internal ribosome entry site and is recognized specifically by the small ribosomal subunit and eukaryotic initiation factor 3 (eIF3) before viral translation initiation. This makes it a good candidate for new drugs targeting HCV. The initial conformation of the RNA is taken from the NMR structure (PDB ID: 1PK7). By using multiple replicas, the aim is to enhance the sampling of the conformational flexibility of the molecule as well as the equilibrium energetics.

B. Resources used in Experiments

Simulations were performed on a range of supercomputing resources on TeraGrid machines, shared TeraGrid-LONI (Louisiana Optical Network Initiative) [17] resources and smaller LONI clusters such as Eric, Oliver, Louie and Poseidon (512 cores). We also used Amazon's EC2 and Nimbus as part of the ScienceCloud at Argonne.

Resource I: TeraGrid/LONI Cluster (QB): To evaluate the performance of BigJob, several experiments have been conducted on different LONI resources. The resources used are: QueenBee (QB), Poseidon and Oliver. For accessing these resources Globus GRAM and an underlying Torque resource manager and Moab scheduler are used.

Resource II: Condor-Pool of LONI Clusters: The LONI Linux clusters: Poseidon, Oliver and Louie are used as Condor Glide-In resources. The base Condor pool was deployed on the user's desktop (Latin). Due to the lack of support for the parallel universe in the Condor Glide-In middleware, we had to modify our setup and could not use the original Glide-In. Rather, we startup a specially configured Condor pool manually on the requested remote nodes using a shell script staged-in from a user's desktop. This script evaluates the environment variables $PBS_NODEFILE to extract the resource list assigned to the Pilot-Job from the local resource manager (in this case Torque). Then, the Condor `master` and `startd` daemons are started via SSH on all assigned nodes. The binaries for the `master` and `startd` daemons are from the standard Condor 7.3.2 package. We adjusted the Condor configuration files to point to the Condor pool on the user's desktop and to support the parallel universe. Additionally, setting the NUM_CPUS parameter to 1 ensures that whole nodes are assigned to jobs. To obtain comparable results no other parameter, especially no timing related parameter, was changed. Some parameter tuning could potentially decrease the startup time from 331 sec on average to less than 60 sec. While we observed some startup overhead, further experiments using multiple Glide-Ins distributed across different LONI resources indicated that Condor scales well: T_C for our scenario of 8 replicas was well reduced with the number of additional resources that have been dynamically added to the Condor pool.

Resource III: Cloud Environments: For our experiments we used different Cloud environments: the Nimbus Science Cloud of the University of Chicago and the Amazon EC2 environment is used. In general, each Cloud has its characteristics. The Nimbus Cloud is accessed via the WSRF interface, while for EC2 the Amazon command line client is used. Each Nimbus VM provides 2 virtual cores and 3.7 GB memory. Amazon offeres different VM types with up to 8 cores. We used the largest VM type (m2.4xlarge) with 8 cores and 68.4 GB of memory and the m1.large instance type with 2 cores and 7.5 GB, which compares best to the virtual machine provided by Nimbus.

C. BigJob: Performance Overheads

Initially, we analyze the overheads resulting from the usage of BigJob across different infrastructures. Typically, the main overhead when using BigJob results from the queueing time at individual resources for Grids and from the VM creation time for Cloud environments. Figure 6 compares the average startup times of the different Pilot-Job backends for Grids, Condor pools and Clouds. Each experiment was repeated at least 10 times.

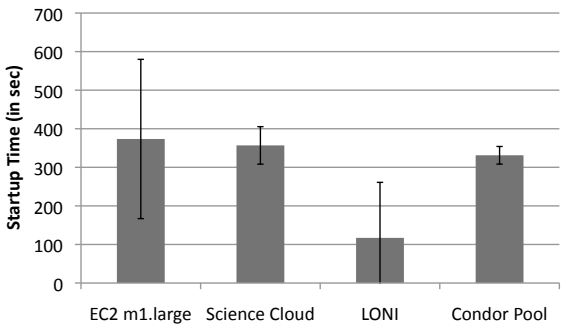

Fig. 6: **BigJob Startup Times:** In Grids the startup time greatly depends upon the queuing time at the local resource manager. However, in our experiments also Clouds showed a high fluctuation in the queueing time.

Interestingly for the experiments conducted, the startup times for the Cloud environments were observed to be larger than queue-waiting time for the LONI resource Poseidon, which coincidentally was very lightly loaded during the course of these experiments. Obviously, the startup of a VM involves higher overheads than spawning a

job on an already running machine: a resource for the VM must be allocated, the VM must be staged to the target and booted up. The following experiments will also show that especially the already oversubscribed intra Cloud network and thus, the staging of the VM can be a bottleneck. Also, there is a large fluctuation in particular in the EC2 environment probably caused by the fact that insufficient resources were available at certain times.

Also, Condor Glide-In and thus, the SAGA-Condor BigJob shows some overheads compared to the Grid BigJob. These overheads can mainly be attributed to the Condor architecture. Condor requires several lightweight daemons to be run on the target resource. The overhead is mainly caused by the time required to setup the pool, propagate the pool information to the main Condor pool and the Condor matchmaking, resource allocation and job spawning mechanism. And although our somewhat limited performance analysis suggests that a BigJob can be launched in a shorter period of time, it is probably true that Condor Glide-in can scale to support a much greater number of workers/sub-jobs. But the take home message is that in spite of a further layer of abstraction and the advantages of interoperability and extensibility that arise as a consequence, basic performance is and can be preserved.

In the following sections we show that the SAGA BigJob is able to support different usage modes. The first scenario involves running 8 replicas on different infrastructure configurations, and determining the T_C . We compute T_C for each configuration 4 times. For the second scenario, we introduce an upper-bound on the acceptable T_C (defined as T_{max}) – as the maximum permissible time to solution and analyze the set of resources utilized in order to complete the workload in T_{max}. Each scenario has several sub-scenarios, of which we discuss selected sub-scenarios.

D. Scenario A: T_C for Workload for Different Resource Configurations

1) T_C for single Resource Usage: In the first example, a given workload – a NAMD simulation consisting of approximately 50,000 atoms – was run for 500 timesteps at different infrastructures. For this experiment the Charm++ [18] version of NAMD was used; Charm++ is in this case just a wrapper layer around MPI (so as to address dependencies).

In most cases the TeraGrid and LONI machines outperform the Cloud resources. This is not surprising, and simply reflects the complications of running MPI-style over multiple virtualized resources. Cloud resources were able to outperform HPC resources in some cases, e. g., a single replica run required 5 sec less time on the largest available EC2 instance with 68.4 GB of memory and 8 virtual cores than on Poseidon or QB.

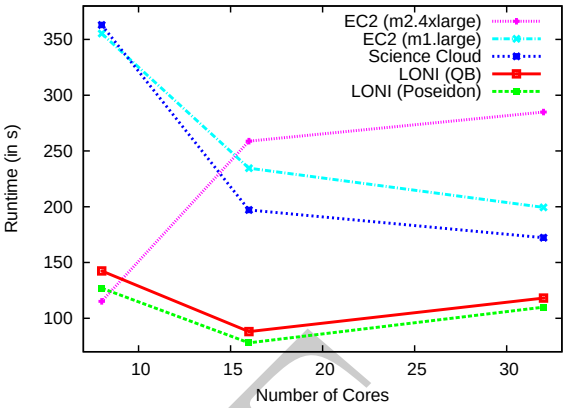

Fig. 7: **NAMD Runtimes on Different Resource Types:** The graph shows that native HPC resource generally outperform Cloud resources in particular when running applications across multiple nodes. However, the new high memory eight core EC2 instance type was able to complete a replica run faster than QB or Poseidon.

On the HPC resources T_C decreased when using up to 16 cores. With more cores a slight increase can be observed. In most EC2 and Nimbus scenarios the T_C decreased with the number of instances and cores used. When using the largest EC2 instance [19] and increase of T_C can be observed. This behavior is particularly caused by the slow interconnects between these instances. Generally those interconnects are heavily oversubscribed in particular in the EC2 environment causing this massive slowdown. Even accounting for differences in the hardware, it is fair to say that MPI jobs across multiple VMs are less likely to have desired performance as a single MPI job across one VM composed of multiple-cores. In the following experiments we will use replica jobs with the size of 8 cores as basis. In this scenario the cost/performance ratio is particularly for Clouds optimal.

2) T_C for Collective Resource Utilization: In this scenario and as proof of scale-out capabilities, we use SAGA BigJob to run replicas across different types of infrastructures. At the beginning of the experiment a particular set of Pilot-Jobs is started in each environment. Once a Pilot-Job becomes active, the application assigns replicas to this job. We measure T_C for different resource configurations using a workload of 8 replicas each running on 8 cores. The following setups have been used

- Scenario 1 (A1): Resource I and III – Clouds and GT2 based Grids.
- Scenario 2 (A2): Resource II and III – Clouds and Condor Grids.
- Scenario 3 (A3): Resource I, II and III – Clouds, GT2 and Condor Grids.

For this experiment the LONI clusters: Poseidon and Oliver are used as Grid and Condor resources and Nimbus as

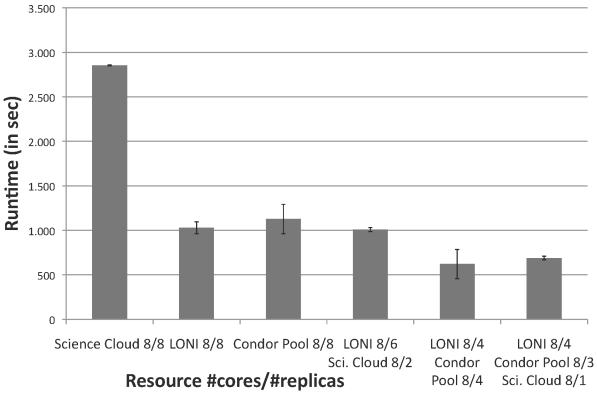

Fig. 8: **A1 - A3: Collective Usage of Grid, Condor and Cloud Resources for Workload of 8 Replicas:** The experiments showed that if the Grid and Condor resource Poseidon has only a light load, no benefits for using additional Cloud resources exist. However, the introduction of an additional Condor or Grid resource significantly decreases T_C .

Cloud resource.

Figure 8 shows the results. For the first three bars, only one infrastructure was used to complete the 8-replica workload. Running the whole scenario in the Science Cloud resulted in a quite poor but predictable performance – the standard deviation for this scenario is very low. The LONI resources are about 3 times faster than the Science Cloud, which corresponds to our earlier findings. The performance of the Condor and Grid BigJob is similar, which can be expected since the underlying physical LONI resources are the same. Solely, a slightly higher startup overhead can be observed in the Condor runtimes.

In the next set of 3 experiments, multiple resources were used. For Scenario A1 (the fourth bar from left), 2 replicas were executed on the Science Cloud. The offloading of 2 replicas to an additional Cloud resource resulted in a light improvement of T_C compared to using just LONI resources. Thus, the usage of Cloud resources must be carefully considered since T_C is determined by the slowest resource, i. e. Nimbus. As described earlier, the startup time for Nimbus images is in particular for such short runs significant. Also, NAMD performs significantly worse in the Nimbus Cloud than on Poseidon or Oliver. Since the startup time on Nimbus averages to 357 sec (see Figure 6) and each 8 core replica runs for about 363 sec (see Figure 7) at least 720 sec must be allowed for running a single replica on Nimbus. Thus, it can be concluded that if resources in the Grids or Condor pool are instantly available, it is not reasonable to start additional Cloud resources. However, it must be noted that there are virtual machines types with a better performance available, e. g. in the Amazon Cloud. These VMs are usually associated with higher costs (up to 2.40 $ per CPU hour) than the Science Cloud VMs. For a further discussion of cost trade-offs for scientific computations in Clouds see Deelman et al. [20].

E. Scenario B: Investigating Workload Distribution for a Given T_{max}

Given that Clouds provide the illusion of infinite capacity, or at least queue wait-times are non-existent, it is likely that when using multiple resource types and with loaded Grids/Clusters (e.g., TeraGrid is currently over-subscribed and typical queue wait times often exceed 24hours), most sub-jobs will end up on the Cloud infrastructure. Thus, in Scenario B, the resource assignment algorithm we use is as follows: We submit tasks to non-Cloud resources first and periodically monitor the progress of the tasks. If insufficient jobs have finished when time equal to T_X has elapsed (determined per criteria outlined below), than we move the workload to utilize Clouds. The underlying basis is that Clouds have an explicit cost associated with them and if jobs can be completed on the TG/Condor-pool while preserving the performance constraints, we opt for such a solution. However, if queue loads prevent the performance requirements being met, we move the jobs to a Cloud-resource, which we have shown has less fluctuation in T_C of the workload.

For this experiment we integrated a progress manager that implements the described algorithm into the replica application. The user has the possibility to specify a maximum runtime and a check interval. At the beginning of each check interval the progress manager compares the number of jobs done with the total number of jobs and estimates the total number of jobs that can be completed within the requested timeframe. If the total number of jobs is higher than this estimate, the progress monitor instantiates another BigJob object request additional Cloud resources for a single replica. In this scenario, each time an intermediate target is not met four additional Nimbus VMs sufficient for running another eight core replica are instantiated. Table I summarizes the results.

Result	# Occurrences	Average T_C
No VM started	6	7.8 min
1 VMs started	1	36.4 min
2 VMs started	1	47.3 min
3 VMs started	2	44.2 min

TABLE I: Usage of Cloud Pilot-Jobs to Ensure Deadline

In the investigated scenario, we configured a maximum runtime of 45 minutes and a progress check interval of 4 minutes. We repeated the same experiment 10 times at different times of the day. In 6 out of 10 cases the scenario completed in about 8 minutes. However, the fluctuation in particular in the waiting time on typical Grid resources can be very high. Thus, in 4 cases it was necessary to start additional VMs to meet the application deadline. In two cases 3 Pilot-Jobs each with 8 cores had to be started, and in one case a single Pilot-Job was sufficient. In a single

case the deadline was missed solely due to the fact that not enough Cloud resources were available, i. e. we are only able to start 2 instead of 3 Pilot-Jobs.

VI. Conclusion and Future Directions

Programming systems and tools often define a Production Grid Infrastructure, and thereby the types and scope of applications that can use them. The ability to utilize existing tools and abstractions, without wholesale refactoring of applications and application usage-modes is important, if not critical in facilitating the uptake of PGIs. Ironically, most applications have been developed to hide from heterogeneity and dynamism of PGI and not embrace them, thus forfeiting the performance advantages that can arise. BigJob – the SAGA based Pilot-Job framework – provides support dynamic execution across a wide range of heterogeneous infrastructures. In this paper, we have shown through a series of experiments using parallel MD simulations as an exemplar, that the right design decisions and abstractions can help provide seamless application-level interoperability across rather different infrastructures – from the TeraGrid to emerging Cloud infrastructures whilst preserving performance. BigJob is being used to integrate otherwise disparate PGI such as the TeraGrid and the OSG, as well as the TeraGrid and EGEE. Thus, these developments will have a practical impact.

We plan to extend this work in two important directions. The first is the use of a more sophisticated coordination mechanism – that supports asynchronous task coordination (e.g. Comet [21]). In this paper, we have focussed on replica simulations, but not on replica-exchange simulations – which involve coordination between the different tasks and intelligence in their placements. It will be interesting to note the performance gains that accrue as a result of such asynchronous coordination on production infrastructure for the replica-exchange problems but over production and extreme-scale infrastructures as made possible by developments outlined in this paper. Also, having established the basic infrastructure to submit tasks to, and manage multiple, heterogenous resources using a simple, single interface, an important second extension of this work is to use more sophisticated task-placement decisions or resource utilization decision making. This in turn is where autonomic features and the ability to easily extend current capabilities will play important roles.

Acknowledgements

Important funding for SAGA has been provided by the UK EPSRC grant number GR/D0766171/1 (via OMII-UK) and HPCOPS NSF-OCI 0710874. SJ acknowledges the e-Science Institute, Edinburgh for supporting the research theme, "Distributed Programming Abstractions" and theme members for shaping many important ideas. This work has also been made possible thanks to the internal resources of the Center for Computation & Technology at Louisiana State University and computer resources provided by LONI. We thank Yaakoub El Khamra for help with initial experiments with the Condor infrastructure. We thank Joohyun Kim (CCT) for assistance with the RNA models. We thank FutureGrid and ScienceCloud for Nimbus resources.

References

[1] S. Jha et al., *Programming Abstractions for Large-scale Distributed Application s*, Submitted to ACM Computing Surveys; draft at http://www.cct.lsu.edu/~sjha/publications/dpa_surveypaper.pdf.

[2] Critical Perspectives on Large-Scale Distributed Applications and Production Grids, Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (GRID09), 2009 http://www.cct.lsu.edu/~sjha/dpa_publications/dpa_grid2009.pdf.

[3] "An Autonomic Approach to Integrated HPC Grid and Cloud Usage", H. Kim, Y. el-Khamra, S. Jha and M. Parashar, accepted for IEEE eScience 2009, Oxford.

[4] http://saga.cct.lsu.edu/projects/tools-and-infrastructure.

[5] A. Tsaregorodtsev et al., DIRAC: A community Grid solution, J. Phys.: Conf. Ser. 119 (2008), p. 062048.

[6] J. T. Moscicki, "Distributed analysis environment for HEP and interdisciplinary applications," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 502, no. 2-3, pp. 426 – 429, 2003.

[7] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: A Fast and Light-Weight TasK ExecutiON Framework," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[8] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, July 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1015617019423

[9] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky computing," *IEEE Internet Computing*, vol. 13, no. 5, pp. 43–51, 2009.

[10] http://saga.cct.lsu.edu.

[11] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith, "A Simple API for Grid Applications (SAGA)," OGF Document Series 90, http://www.ogf.org/documents/GFD.90.pdf.

[12] Y. El-Khamra and S. Jha, "Developing autonomic distributed scientific applications: a case study from history matching using ensemblekalman-filters," in *GMAC '09: Proceedings of the 6th international conference industry session on Grids meets autonomic computing*. New York, NY, USA: ACM, 2009, pp. 19–28.

[13] A. Merzky, K. Stamou, S. Jha and D. S. Katz, A Fresh Perspective on Developing and Executing DAG-Based Distributed Applications: A Case-Study of SAGA-based Montage, accepted for IEEE Conference on eScience 2009, Oxford. http://www.cct.lsu.edu/~sjha/dpa_publications/saga_montage_G09.pdf.

[14] A. Luckow, S. Jha, J. Kim, A. Merzky, and B. Schnor, "Adaptive Distributed Replica–Exchange Simulations," in *Theme Issue of the Philosophical Transactions of the Royal Society A*, vol. 367, 2009.

[15] Thijs Metsch et. al, "Open Cloud Computing Interface Specification," http://forge.ogf.org/sf/go/doc15731, 2009, open Grid Forum.

[16] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The apples parameter sweep template: User-level middleware for the grid," *Sci. Program.*, vol. 8, no. 3, pp. 111–126, 2000.

[17] http://www.loni.org.

[18] L. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," Champaign, IL, USA, 1993.

[19] http://aws.typepad.com/aws/2009/10/two-new-ec2-instance-types-additional-memory.html.

[20] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–12.

[21] 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*, 2007, pp. 229–230.