Application Level Interoperability between Clouds and Grids

Andre Merzky[1], Katerina Stamou[2], Shantenu Jha[123]
[1]*Center for Computation & Technology, Louisiana State University, USA*
[2]*Department of Computer Science, Louisiana State University, USA*
[3]*e-Science Institute, Edinburgh, UK*

***Abstract*—SAGA is a high-level programming interface which provides the ability to develop distributed applications in an infrastructure independent way. In an earlier paper, we discussed how SAGA was used to develop a version of MapReduce which provided the user with the ability to control the relative placement of compute and data, whilst utilizing different distributed infrastructure. In this paper, we use the SAGA-based implementation of MapReduce, and demonstrate its interoperability across Clouds and Grids. We discuss how a range of *cloud adaptors* have been developed for SAGA. The major contribution of this paper is the demonstration – possibly the first ever, of interoperability between different Clouds and Grids, without any changes to the application. We analyse the performance of SAGA-MapReduce when using multiple, different, heterogeneous infrastructure concurrently for the same problem instance; However, we do not strive to provide a rigorous performance model, but to provide a proof-of-concept of application-level interoperability and illustrate its importance.**

I. Introduction

Although Clouds are a nascent infrastructure, there is a ground swell in interest to adapt these emerging powerful infrastructure for large-scale scientific applications [5]. Inevitably, and as with any emerging technology, the unified concept of a Cloud – if ever there was one, is evolving into different flavours and implementations, with distinct underlying system interfaces, semantics and infrastructure. For example, the operating environment of Amazon's Cloud (EC2) is very different from that of Google's Cloud. Specifically for the latter, there already exist multiple implementations of Google's Bigtable, such as HyberTable, Cassandra and HBase. There is bound to be a continued proliferation of such Cloud based infrastructure; this is reminiscent of the plethora of Grid middleware distributions. Thus to safeguard against the proliferation of Cloud infrastructure, application-level support and interoperability for different applications on different Clouds is critical if they are not have the same limited impact on Scientific Computing of Grids. And issues of scale aside, the transition of existing distributed programming models and styles, must be as seamless and as least disruptive as possible. A fundamental question at the heart of all these important considerations, is the question of how scientific applications can be developed so as to utilize as broad a range of distributed systems as possible, without vendor lock-in, yet with the flexibility and performance that scientific applications demand?

Currently, it is unclear what kind of programming models (PM) and programming systems (PS) will emerge for Clouds; this in turn will depend, amongst other things, on the kinds of applications that will come forward to try to utilise Clouds and system-level interfaces that are exposed by Cloud providers. Additionally, there are infrastructure specific features – technical and policy, that might influence the design of PM and PS. For example, EC2 – the archetypal Cloud System, has a well-defined cost model for data transfer across *its* network. Hence, any PM for Clouds must be cognizant of the requirement to programmatically control the placement of compute and data relative to each other – both statically (pre-run time) and at run-time. In general, for most Cloud applications the same computational task can be priced very differently for possibly the same performance; conversely, the same computational task can have very different performance for the same price. It is important for effective scientific application development on Clouds that, any PM or PS should not be constrained to a specific infrastructure, i.e., should support infrastructure interoperability at the application-level.

In Ref [1], we established that the SAGA programming system which provides a standard interface, can support simple, yet powerful programming models. Specifically, we implemented a simple data-parallel programming task (MapReduce) using SAGA; this involved the concurrent execution of simple, embarrassingly-parallel data-analysis tasks. We demonstrated that the SAGA-based implementation is infrastructure independent whilst still providing control over the deployment, distribution and run-time decomposition. Work is underway to extend our SAGA based approach in the near future to involve tasks with complex and interrelated dependencies. Using data-sets of size up to 10GB, and up to 10 workers, we provided detailed performance analysis of the SAGA-MapReduce implementation, and showed how controlling the distribution of computation and the payload-per-worker helped enhance performance.

Having thus established the effectiveness of SAGA, the primary focus of this paper is to use the SAGA-based MapReduce as an exemplar to establish the interoperability aspects of the SAGA programming system. Specifically, we will demonstrate that SAGA-MapReduce is usable on traditional (Grids) and emerging (Clouds) distributed infrastructure *concurrently and cooperatively towards a solution of the same problem instance*. Specifically, our approach is to take SAGA-MapReduce and to

978-0-7695-3677-4/09 $25.00 © 2009 IEEE
DOI 10.1109/GPC.2009.17

use the *same* implementation of SAGA-MapReduce to solve the same instance of the word counting problem, by using different worker distributions over Clouds and Grid systems, and thereby also test for interoperability between different flavours of Clouds as well as between Clouds and Grids.

The Case for Interoperabilty: Interoperability amongst Clouds and Grids can be achieved at different levels. For example, service-level interoperability amongst Grids has been demonstrated by the OGF-GIN group; application-level interoperability (ALI) remains a harder goal to achieve. Clouds provide services at different levels (Iaas, PaaS, SaaS); standard interfaces to these different levels do not exist. With an unsettled and rapidly changing Cloud Computing landscape, it is unclear if the community is ready for standards-based service-level interoperability at the moment. In addition, there is little business motivation for Cloud providers to define, implement and support new/standard interfaces, there is a case to be made that applications would benefit from Cloud interoperability. We argue that by addressing interoperability at the application-level this can be easily achieved for both scientific and enterprise applications. ALI arises when, say other than compiling on a different or new platform, there are no further changes required of the application. Also, ALI provides automated, generalized and extensible solutions to use new resources; in some ways, ALI is strong interoperability, whilst service-level interoperability is weak interoperability. The complexity of providing ALI is non-uniform and depends upon the application under consideration. For example, it is somewhat easier for simple "execution unaware" applications to utilize heterogeneous multiple distributed environments, than for applications with multiple distinct and possibly distributed components. A pre-requisite for ALI is infrastructure independent programming. Google's MapReduce is tied to Google's file-system; Hadoop is intrinsically linked to Hadoop file-system (HDFS), as is Pig.

It can be asked if the emphasis on utilising multiple Clouds or Clouds and Grids concurrently is premature, given that programming models/systems for Clouds are just emerging? In many ways the emphasis on interoperability is an appreciation and acknowledgement of an application-centric perspective – that is, as infrastructure changes and evolves it is critical to provide seamless transition and development pathways for applications and application developers. In addition there exist a wide range of applications that have decomposable but heterogeneous computational tasks. It is conceivable, that some of these tasks are better suited for traditional Grids, whilst some are better placed in Cloud environments. Montage – a very popular Astronomy application provides a prominent example of such an application. Additionally, due to different data-compute affinity requirement amongst the tasks, some workers might be better placed on a Cloud [8], whilst some may optimally be located on regular Grids. Complex dependencies and inter-relationships between sub-tasks make this often difficult to determine before run-time.

Effort directed towards ALI on Clouds/Grids in addition to satisfying basic curiosity of "if and how to interoperate", might also possibly provide different insight into the programming challenges and requirements.

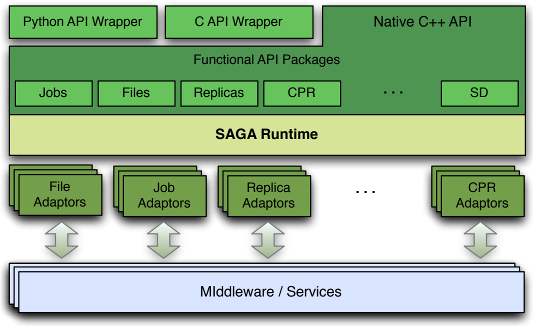

Fig. 1: In addition to the programmer's interface, the other important components of the landscape are the SAGA engine, and functional adaptors.

We focus on MapReduce, which is an application with multiple homogeneous workers (although the data-load per worker can vary); however, it is easy to conceive of an application where workers (tasks) can be heterogeneous, i.e., each worker is different and may have different data-compute ratios. It is worth mentioning that most data-intensive scientific applications fall into this category e.g., high-energy and LIGO data-analysis. Additionally, with different Clouds providers, fronting different Economic Models of computing, it is important to be able to utilise the "right resource", in the right way. We briefly discussed how moving prodigious amounts of data across Cloud networks, as opposed to moving the compute unit could be expensive. As current programming models don't provide explicit support or control for affinity [8], and in the absence of autonomic performance models, the end-user is left with performance management, and with the responsibility of explicitly determining which resource is optimal. Clearly interoperability between different flavours of Clouds, and Clouds and Grids is an important pre-requisite.

II. SAGA

The SAGA [9] programming system contains a high level API that provides a simple, standard and uniform interface for the most commonly required distributed functionality. SAGA can be used to encode distributed applications [6, 2], toolkits to manage distributed applications as well as implement abstractions that support commonly occurring programming, access and usage patterns.

Fig. 1 provides an overview of the SAGA programming system and the main functional areas that SAGA provides a standardized interface to. Based upon an analysis of more than twenty applications, the most commonly required functionality involve job submission across different distributed platforms, support for file access and transfer, as well as logical file support. Additionally there is support for Checkpoint and Recovery (CPR) and Service Discovery (SD). The API is written in C++, with Python, C and Java language support. The *engine* is the main library, which provides dynamic support for run-time environment decision making through loading relevant adaptors. We will not discuss details of SAGA here; details can be found elsewhere [3].

III. Interfacing SAGA to Grids and Clouds

SAGA was originally developed primarily for compute-intensive Grids. This was a user-driven design decision, i.e., the majority of applications that motivated the design and formulation of version 1.0 of the API were HPC applications attempting to utilize distributed resources. Ref [1] demonstrated that in spite of its original design constraints, SAGA can be used to develop data-intensive applications in diverse distributed environments, including Clouds. This in part is due to the fact that much of the "distributed functionality" required for data-intensive applications remains the same – namely the ability to submit jobs to different back-ends, the ability to move files between distributed resources etc. Admittedly, and as we will discuss, the semantics of, say the basic `job_submit()` changes in going from Grid environments to Cloud environments. So how, in spite of such significant changes in the semantics does SAGA keep the application immune to change? The basic feature that enables this capability is a context-aware adaptor that is dynamically loaded. In the remainder of this section, we will describe how, through the creation of a set of simple *adaptors*, the primary functionality of most applications is supported and on which the interoperability is predicated.

A. Adaptors: Design and Implementation

1) Local Adaptors: Although SAGA's default local adaptors do not have much to do with interoperability, their importance for the implementation of other remote adaptors will become clear later on. The local job adaptor utilizes `boost::process` (on Windows) and plain `fork/exec` (on Unix derivatives) to spawn, control and watch local job instances. The local file adaptor uses `boost::filesystem` classes for filesystem navigation, and `std::fstream` for local file I/O.

2) SSH adaptors: The SSH adaptors are based on three different command line tools, namely `ssh`, `scp` and `sshfs`. Further, all ssh adaptors rely on the availability of SSH security credentials for remote operations. The SSH context adaptor implements some mechanisms to (a) discover available key-pairs automatically, and (b) to verify the validity and usability of the found and otherwise specified credentials.

`ssh` is used to spawn remote job instances, for which the SSH job adaptor instantiates a *local* `saga::job::service` instance, and submits the respective SSH command lines to it. The local job adaptor described above then takes care of process I/O, detachment, etc. A significant drawback of this approach is that several SAGA methods act upon the local ssh process instead of the remote application instance, which is far from ideal. Some of these operations can be migrated to the remote hosts, via separate ssh calls, but that process is complicated due to the fact that ssh does not report the remote process ID back to the local job adaptor. We circumvent this problem by setting a uniquely identifying environment variable for the remote process, which allows us to identify the process.

`sshfs` is used to access remote files via ssh services. `sshfs` is a user space file system driver which uses FUSE, and is available for MacOS, Linux, and some other Unix derivatives. It enables a remote file system to be mounted into the local namespace, and transparently forwards all file access operations via ssh to the remote host. The ssh file adaptor uses the local job adaptor to call the sshfs process, to mount the remote filesystem, and then forward all file access requests to the local file adaptor, which operates on the locally mounted file system. The ssh adaptor thus translates URLs from the ssh namespace into the local namespace, and back.

`scp` is used by both the ssh job and file adaptor to transfer utility scripts to the remote host, e.g. to check for remote system configuration, or to distribute ssh credentials.

3) SSH/SSHFS credential management: When starting a remote application via ssh, we assume valid SSH credentials (i.e. private/public key pairs, or gsi credentials etc.) are available. The type and location of these credentials is specified by the local application, by using the respective `saga::context` instances. In order to facilitate home-calling, i.e. the ability of the remotely started application to use the same ssh infrastructure to call back to the original host, e.g. by spawning jobs in the opposite direction, or by accessing the original host's file system via sshfs, we install the originally used ssh credential in a temporary location on the remote host. The remote application is informed about these credentials, and the ssh context adaptor picks them up by default, so that home-calling is available without the need for any application level intervention. Also, a respective entry to the local `authorized_keys` file is added.

4) AWS adaptors: SAGA's AWS (Amazon Web Service) adaptor suite is an interface to services which implement the cloud web service interfaces as specified by Amazon. These interfaces are not only used by Amazon to allow programmatic access to their Cloud infrastructures – EC2 and S3, amongst others, but are also used by several other Cloud service providers, such as Eucalyptus[7] and Nimbus. The AWS job adaptor is thus able to interface to a variety of Cloud infrastructures, as long as they adhere to the AWS interfaces.

The AWS adaptors do not directly communicate with the remote services, but instead rely on Amazon's set of java based command line tools. Those are able to access the different infrastructures, when configured correctly via specific environment variables. The AWS job adaptor uses the local job adaptor to manage the invocation of the command line tools, e.g. to spawn new virtual machine (VM) instances, to search for existing VM instances, etc. Once a VM instance is found to be available and ready to accept jobs, a ssh job service instance for that VM is created, and henceforth takes care of all job management operations. The AWS job adaptor is thus only responsible for VM discovery and management – the actual job creation and operations are performed by the ssh job adaptor (which in turn utilizes the local job adaptor).

The security credentials used by the internal ssh job service instance are derived from the credentials used to create the VM instance; upon VM instance creation, an AWS keypair is used to authenticate the user against her 'cloud account'. That keypair is automatically registered at the new VM instance to allow for remote ssh access. The AWS context adaptor collects both the public and private AWS keys[1], creates a respective ssh context, and thus allows the ssh adaptors to perform job and file based SAGA operations on the VM instance.

Note that there is an important semantic difference between 'normal' (e.g. Grid based) and 'Cloud' job services in SAGA: a normal job service is assumed to have a lifetime which is completely independent from the application which accesses that service. For example, a GRAM gatekeeper has a lifetime of days and weeks, and allows a large number of applications to utilize it. An AWS job service however points to a potentially volatile resource, or even to a non-existing resource – the resource needs then to be created on the fly. There are two important implications. Firstly, the startup time for a AWS job service is typically much larger than other remote job service, at least in the case where a VM is created on the fly: the VM image needs to be deployed to some remote resource, the image must be booted, and potentially needs to be configured to enable the hosting of custom applications[2]. The second implication is that the *end* of the job service lifetime is usually of no consequence for normal remote job services. For a dynamically provisioned VM instance, however, it raises the question if that instance should be closed down, or if it should automatically shut down after all remote applications finish, or even if it should survive for a specific time, or forever. Ultimately, it is not possible to control these VM lifetime attributes via the current SAGA API (by design). Instead, we allow one of these policies to be chosen either implicitly (e.g. by using special URLs to request dynamic provisioning), or explicitly over SAGA config files or environment variables[3]. Future SAGA extensions, in particular Resource Discovery and Resource Reservation extensions, may have a more direct and explicit notion of resource lifetime management.

```
SAGA Job Launch via GRAM gatekeeper
{ // contact a GRAM gatekeeper
  saga::job::service     js;
  saga::job::description jd;
  jd.set_attribute (``Executable'', ``/tmp/my_prog'');
  // translate job description to RSL
  // submit RSL to gatekeeper, and obtain job handle
  saga:job::job j = js.create_job (jd);
  j.run ():
  // watch handle until job is finished
  j.wait ();
 } // break contact to GRAM
```

Fig. 2: Job launch on a Grid via GRAM

B. Globus Adaptors

SAGA's Globus adaptor suite is amongst the most-utilized adaptors. As with SSH, security credentials are expected to be managed out-of-bound, but different credentials can be utilized by pointing `saga::context` instances to them as needed. Other than the AWS and ssh adaptors, the Globus adaptors do not rely on command line tools, but rather link directly against the respective Globus libraries: the Globus job adaptor is thus a GRAM client, the Globus file adaptor a GridFTP client. In experiments, non-Cloud jobs were started using either GRAM or SSH. In either case, file I/O was performed either via ssh, or via a shared Lustre filesystem – the GridFTP functionality were not used for experiments in this paper.

```
SAGA create a VM instance on a Cloud
{// create a VM instance on Eucalyptus/Nimbus/EC2
 saga::job::service     js;
 saga::job::description jd;
 jd.set_attribute (``Executable'', ``/tmp/my_prog'');
 // translate job description to ssh command
 // run the ssh command on the VM
 saga:job::job j = js.create_job (jd);
 j.run ():
 // watch command until done
 j.wait ();
} // shut down VM instance
```

Fig. 3: Job launch on a VM via ssh

IV. SAGA-based MapReduce

After SAGA-MapReduce we have also developed real scientific applications using SAGA based implementations of patterns for data-intensive computing: multiple sequence alignment can be orchestrated using the SAGA-All-pairs implementation, and genome searching can be implemented using SAGA-MapReduce (see Ref. [1]).

A. SAGA-MapReduce Implementation

Our implementation of SAGA-MapReduce interleaves the core logic with explicit instructions on where processes are to be scheduled. The advantage of this approach is that our implementation is no longer bound to run on a system providing the appropriate semantics originally required by MapReduce, and is portable to a broader range of generic systems as well. The drawback is that it is relatively more complicated to extract performance.The fact that the current implementation is single-threaded currently is a primary factor for slowdown. Critically, however, none of these complexities are transferred to the end-user, and they remain hidden within the framework. Also many of these are due to the early-stages of SAGA and incomplete implementation of features, and not a fundamental limitation of the design or concept of the interface or programming models that it supports.

This simple interface provides the complete functionality needed by any MapReduce algorithm, while hiding the more complex functionality, such as chunking of the input, sorting the intermediate results, launching and coordinating the workers, etc. as these are implemented by the framework. The application consists of two independent processes, a master and worker processes. The master process is responsible for:

- Launching all workers for the map and reduce steps as described in a configuration file provided by the user
- Coordinating the executed workers, including the chunking of the data, assigning the input data to the workers of the map step, handling the intermediate data files

[1]The public key needs to be collected from the remote instance.

[2]The AWS job adaptor allows the execution of custom startup scripts on newly instantiated VMs, to allow for example, the installation of additional software packages, or to test for the availability of certain resources.

[3]only some of these policies are implemented at the moment.

produced by the map step and passing the names of the sorted output files to the workers of the reduce step, and collecting the generated outputs from the reduce steps and relaunching single worker instances in case of failures,

The master process is readily available to the user and needs no modification for different Map and Reduce functions to execute. The worker processes get assigned work either from the map or the reduce step. The functionality for the different steps have to be provided by the user, which means the user has to write 2 C++ functions implementing the required MapReduce algorithm.

Both the master and the worker processes use the SAGA-API as an abstract interface to the used infrastructure, making the application portable between different architectures and systems. The worker processes are launched using the SAGA job package, allowing the jobs to launch either locally, using Globus/GRAM, Amazon Web Services, or on a Condor pool. The communication between the master and the worker processes is ensured by using the SAGA advert package, abstracting an information database in a platform independent way (this can also be achieved through SAGA-Bigtable adaptors). The Master process creates partitions of data (referred to as chunking, analogous to Google's MapReduce), so the data-set does not have to be on one machine and can be distributed; this is an important mechanism to avoid limitations in network bandwidth and data distribution. These files could then be recognized by a distributed File-System, such as HDFS.

B. SAGA-MapReduce Set-Up

When deploying compute clients on a *diverse* set of resources, the question arises if and how these clients need to be configured to function properly in the overall application scheme. SAGA-MapReduce compute clients (workers) require two pieces of information to function: (a) the contact address of the advert service used for coordinating the clients, and for distributing work items to them; and (b) a unique worker ID to register with in that advert service, so that the master can start to assign work items. Both information are provided via command line parameters to the worker, at startup time.

The master application requires the following additional information: i) a set of resources where the workers can execute, ii) location of the input data, iii) the location of the output data, and iv) the contact point for the advert service for coordination and communication.

In a typical configuration file, for example, three worker instances could be started; the first could be started via gram and PBS on qb1.loni.org, second started on a pre-instantiated EC2 image (instance-id `i-760c8c1f`), and finally will be running on a dynamically deployed EC2 instance (no instance id given). Note that the startup times for the individual workers may vary over several orders of magnitudes, depending on the PBS queue waiting time and VM startup time. The mapreduce master will start to utilize workers as soon as they are able to register themselves, so will not wait until all workers are available. That mechanism both minimizes time-to-solution, and maximizes resilience against worker loss.

The scheme `any` acts here as a placeholder for SAGA, so that the SAGA engine can choose an appropriate adaptor. The master would access the file via the default local file adaptor. The Globus clients may use either the GridFTP or SSH adaptor for remote file success (but in our experimental setup would also succeed using the local file adaptor, as the Lustre FS is mounted on the cluster nodes), and the EC2 workers would use the ssh file adaptor for remote access. Thus, the use of the placeholder scheme frees us from specifying and maintaining a concise list of remote data access mechanisms per worker. Also, it facilitates additional resilience against service errors and changing configurations, as it leaves it up to the SAGA engine's adaptor selection mechanism to find a suitable access mechanism at runtime. A simple parameter can control the number of workers created on each compute node; as we will see by varying this parameter, the chances are good that compute and communication times can be interleaved, and that the overall system utilization can increase (especially in the absence of precise knowledge of the execution system).

V. SAGA-MAPREDUCE ON CLOUDS AND GRIDS

There are several aspects to interoperability. A simple form of interoperability is that any application can use any Clouds systems without changes to the application: the application simply needs to instantiate a different set of security credentials for the respective runtime environment. We refer to this as Cloud-Cloud interoperability. By almost trivial extension, SAGA also provides Grid-Cloud interoperability, as shown in Fig. 2 and 3, where exactly the same interface and functional calls lead to job submission on Grids or on Clouds. Although syntactically identical, the semantics of the calls and back-end management are somewhat different. As discussed, SAGA provides interoperability quite trivially thanks to the dynamic loading of adaptors. Thanks to the low overhead of developing adaptors, SAGA has been deployed on three Cloud Systems – Amazon, Eucalyptus [7] (we have a local installation of Eucalyptus at LSU – GumboCloud) and Nimbus. In this paper, we focus on EC2, Eucalyptus and the TeraGrid (TG).

A. Deployment Details

In order to fully utilize cloud infrastructures for SAGA applications, the VM instances need to fulfill a couple of prerequisites: the SAGA libraries and its dependencies need to be deployed, need some external tools which are used by the SAGA adaptors at runtime – such as ssh, SCP, and sshfs. The latter needs the FUSE kernel module to function – so if remote access to the cloud compute node's file system is wanted, the respective kernel module needs to be installed as well. There are two basic options to achieve the above, either a customized VM image which includes all the software that is used, or the respective packages that are installed after VM instantiation (on the fly). Hybrid approaches are possible too.

We support the runtime configuration of VM instances by staging a preparation script to the VM after its creation, and executing it with root permissions. In particular for apt-get linux distribution, the post-instantiation software deployment is actually fairly painless, but naturally adds a significant amount of time to the overall VM startup (which encourages the use of asynchronous operations). For experiments in this paper, we prepared custom VM images with all prerequisites pre-installed. We utilize the preparation script solely for some fine tuning of parameters: for example, we are able to deploy custom saga.ini files, or ensure the finalization of service startups before application deployment[4].

Eucalyptus VM images are basically customized Xen hypervisor images, as are EC2 VM images. Customized in this context means that the images are accompanied by a set of metadata which tie it to a specific kernel and ramdisk images. Also, the images contain specific configurations and startup services which allow the VM to bootstrap cleanly in the respective Cloud environment, e.g. to obtain the necessary user credentials, and to perform the wanted firewall setup etc. As these systems all use Xen based images, a conversion of these images for the different cloud systems is in principle straightforward. But sparse documentation and lack of automatic tools however, make it challenging, at least to the average end user. Compared to that, the derivation of customized images from existing images is well documented with well supported tools – as long as the target image is to be used on the same Cloud system as the original one.

In executing SAGA-MapReduce on different Clouds, the change lies in the run-time system and deployment architecture. For example, when running SAGA-MapReduce on EC2, the master process resides on one VM, while workers reside on different VMs. Depending on the available adaptors, Master and Worker can either perform local I/O on a global/distributed file system, or remote I/O on a remote, non-shared file system. Application deployment and configuration (as discussed above) are also performed via sshfs. On EC2, we created a custom virtual machine (VM) image with pre-installed SAGA. For Eucalyptus, a boot strapping script equips a standard VM instance with SAGA, and SAGA's prerequisites (mainly boost). To us, a mixed approach seemed most favourable, where the bulk software installation is statically done via a custom VM image, but software configuration and application deployment are done dynamically during VM startup.

B. Experiments

In an earlier paper (Ref [1]), we performed tests to demonstrate how SAGA-MapReduce utilizes different infrastructures and provides control over task-data placement; this led to insight into performance on "vanilla" Grids. The primary aim of this work is to establish, via well-structured and designed experiments, the fact that SAGA-MapReduce has been used to demonstrate Cloud-Cloud interoperability and Cloud-Grid interoperability. In this paper, we perform the following experiments:

1) We compare the performance of SAGA-MapReduce when exclusively running on a Cloud platform to that when on Grids. We vary the number of workers (1 to 10) and the data-set sizes varying from 10MB to 1GB.
2) For Clouds, we then vary the number of workers per VM, such that the ratio is 1:2; we repeat with the ratio at 1:4 – that is the number of workers per VM is 4.
3) We then distribute the same number of workers across two different Clouds - EC2 and Eucalyptus.
4) Finally, for a single master, we distribute workers across Grids (QueenBee on the TG) and Clouds (EC2 and Eucalyptus) with one job per VM.

It is worth reiterating, that although we have captured concrete performance figures, it is not the aim of this work to analyze the data and provide a performance model. In fact it is difficult to understand performance implications, as a detailed analysis of the data and understanding the performance will involve the generation of "system probes", as there are differences in the specific Cloud system implementation and deployment. For example, in EC2 Clouds there exists the notion of availability zone, which is really just a control on which data-center/cluster the VM is placed. In the absence of explicit mention of the availabilty zone, it is difficult to determine or assume that the availability zones for multiple, distributed workers are the same. However, for GumboCloud, it can be established that the same cluster is used and thus it is fair to assume that the VMs are local with respect to each other. Similarly, without explicit tests, it is often unclear whether data is local or distributed. It should also be assumed that for Eucalpytus based Clouds, data is also locally distributed (i.e. same cluster with respect to a VM), whereas for EC2 Clouds this cannot be assumed to be true for every experiment/test. In a nutshell without adjusting for different system implementations, it is difficult to rigorously compare performance figures for different configurations on different machines. At best we can currently derive trends and qualitative information.

It takes SAGA about 45s to instantiate a VM on Eucalyptus and about 200s on average on EC2. We find that the size of the image (say 5GB versus 10GB) influences the time to instantiate an image, but is within image-to-image instantiation time fluctuation. Once instantiated, it takes from 1-10s to assign a job to an existing VM on Eucalyptus, or EC2. The option to tie the VM lifetime to the `job_service` object lifetime is a configurable option. It is also a matter of simple configuration to vary how many jobs (in this case workers) are assigned to a single VM. The default case is 1 worker per VM; the ability to vary this number is important – as details of actual VMs can differ as well as useful for our experiments.

C. Results and Analysis

The total time to solution (T_s) of a SAGA-MapReduce job, can be decomposed as the sum of three primary components – t_{over}, t_{comp} and t_{coord}. The first term t_{over}, is defined as the time for pre-processing – which is the time to chunk into fixed size data units, to distribute them and also to spawn the job. This is in some ways the overhead of the process (hence

[4]For example, when starting SAGA applications are started before the VM's random generator is initialized, our current uuid generator failed to function properly – the preparation script checks for the availability of proper uuids, and delays the application deployment as needed.

Number-of-Workers		Data size	T_s	T_{spawn}	$T_s - T_{spawn}$
TeraGrid	AWS	(MB)	(sec)	(sec)	(sec)
4	-	10	8.8	6.8	2.0
-	1	10	4.3	2.8	1.5
-	2	10	7.8	5.3	2.5
-	3	10	8.7	7.7	1.0
-	**4**	10	13.0	10.3	2.7
-	4 (1)	10	11.3	8.6	2.7
-	4 (2)	10	11.6	9.5	2.1
-	2	100	7.9	5.3	2.6
-	**4**	100	12.4	9.2	3.2
-	10	100	29.0	25.1	3.9
-	**4 (1)**	100	16.2	8.7	7.5
-	**4 (2)**	100	12.3	8.5	3.8
-	6 (3)	100	18.7	13.5	5.2
-	8 (1)	100	31.1	18.3	12.8
-	8 (2)	100	27.9	19.8	8.1
-	8 (4)	100	27.4	19.9	7.5

TABLE I: Performance data for different configurations of worker placements. The master places the workers on either Clouds or on the TeraGrid (TG). The configurations – separated by horizontal lines, are classified as either all workers on the TG or having all workers on EC2. For the latter, unless otherwise explicitly indicated by a number in parenthesis, every worker is assigned to a unique VM. In the final set of rows, the number in parenthesis indicates the number of VMs used. It is interesting to note the significant spawning times, and its dependence on the number of VM, which typically increase with the number of VMs. T_{spawn} does not include instantiation of the VM.

Number-of-Workers			Size	T_s	T_{spawn}	$T_s - T_{spawn}$
TG	AWS	Eucalyptus	(MB)	(sec)	(sec)	(sec)
-	1	1	10	5.3	3.8	1.5
-	2	2	10	10.7	8.8	1.9
-	1	1	100	6.7	3.8	2.9
-	2	2	100	10.3	7.3	3.0
1	-	1	10	4.7	3.3	1.4
1	-	1	100	6.4	3.4	3.0
2	**2**	-	10	7.4	5.9	1.5
3	3	-	10	11.6	10.3	1.6
4	4	-	10	13.7	11.6	2.1
5	5	-	10	33.2	29.4	3.8
10	10	-	10	32.2	28.8	2.4

TABLE II: Performance data for different configurations of worker placements on TG, Eucalyptus-Cloud and EC2. The first set of data establishes Cloud-Cloud interoperability. The second set (rows 5- 11) shows interoperability between Grids-Clouds (EC2). The experimental conditions and measurements are similar to Table 1.

the subscript). Another component of the overhead is the time it takes to instantiate a VM (T_{spawn}). Our performance figures take the net instantiation time into account and thus normalize for multiple VM instantiation – whether serial or concurrent started-up. In fact, for data we report in Table 1 and 2, the spawning time does not consider instantiation, i.e., the job is dynamically assigned an existing VM; thus numbers indicate relative performance and are amenable to direct comparison irrespective of the number of VMs. t_{comp} is the time to actually compute the map and reduce function on a given worker, whilst t_{coord} is the time taken to assign the payload to a worker, update records and to possibly move workers to a destination resource; in general, t_{coord} scales as the number of workers increases.

We find that t_{comp} is typically greater than t_{coord}, but when the number of workers gets large, and/or the computational load per worker small, t_{coord} can dominate (internet-scale communication) and increase faster than t_{comp} decreases, thus overall T_s can increase for the same data-set size, even though the number of independent workers increases. The number of workers associated with a VM also influences the performance, as well as the time to spawn; for example – as shown by the three lower boldface entries in Table 1, although 4 identical workers are used depending upon the number of VMs used, T_c (defined as $T_S - T_{spawn}$) can be different. In this case, when 4 workers are spread across 4 VMs (i.e. default case), T_c is lowest, even though T_{spawn} is the highest; T_c is highest when all four are clustered onto 1 VM. When exactly the same experiment is performed using data-set of size 10MB, it is interesting to observe that T_c is the same for 4 workers distributed over 1 VM as it is for 4 VMs, whilst when the performance for the case when 4 workers are spread-over 2 VMs out-perform both (2.1s).

Table 2 shows performance figures when equal number of workers are spread across two different systems; for the first set of rows, workers are distributed on EC2 and Eucalyptus. For the next set of rows, workers are distributed over the TG and Eucalyptus, and in the final set of rows, workers are distributed between the TG and EC2. Given the ability to distribute at will, we compare performance for the following scenarios: (i) when 4 workers are distributed equally (i.e., 2 each) across a TG machine and on EC2 (1.5s), with the scenarios when, (ii) all 4 workers are either exclusively on EC2 (2.7s), (iii) or all workers are on the TG machine (2.0s) (see Table 1, boldface entries on the first and fifth line). It is *interesting* that in this case T_c is lower in the distributed case than when all workers are executed locally on either EC2 or TG; we urge that not too much be read into this, as it is just a coincidence that a *sweet spot* was found where on EC2, 4 workers had a large spawning overhead compared to spawning 2 workers, and an increase was in place for 2 workers on the TG. Also it is worth reiterating that for the same configuration there are experiment-to-experiment fluctuations (typically less than 1s). The ability to enhance performance by distributed (heterogeneous) work-loads across different systems remains a distinct possibility, however, we believe more systematic studies are required.

VI. Discussion

Experience: In addition to problems alluded to in earlier footnotes, we mention two challenges faced. We found that the VM images get corrupted, if for some reason SAGA-MapReduce does not terminate properly. Also given local firewall and networking policies, we encountered problems in initially accessing/addressing the VMs directly.

Programming Models for Clouds: We began this paper with a discussion of programming systems/model for Clouds, and the importance of support for relative data-compute placement. Ref [8] introduced the notion of *affinity* for Clouds; it is imperative that any programming model/system be cognizant of the notion of affinity. We have implemented the first steps in a PM which provides easy control over relative data-compute placement; a possible next step would be to extend SAGA to support affinity (data-data, data-compute). There exist other emerging programming systems like Dryad, Sawzall and Pig, which could be used in principle to support the notion of affinity as well as develop/use MapReduce; however we re-emphasise that the primary strength of SAGA in addition to supporting affinity is, i) infrastructure independence, ii) it is general-purpose and extensible, iii) provides greater control to the end-user if required. Distinguish the infrastructure independence of SAGA-MapReduce with the reliance of Google's MapReduce [4] on a number of capabilities of the underlying system – mostly related to file operations, but including system features related to process/data allocation. Google use their distributed file system (Google File System) to keep track of where each file is located. Additionally, they coordinate this effort with Bigtable.

Simplicity versus Completeness: There exist both technical reasons and social engineering problems responsible for low uptake of Grids. One universally accepted reason is the complexity of Grid systems – the interface, software stack and underlying complexity of deploying distributed application. But this is also a consequence of the fact that Grid interfaces tend to be "complete" or very close thereof. For example, while certainly not true of all cases, consider the following numbers, which we believe represent the above points well: Globus 4.2 provides, in its Java version, approximately 2,000 distinct method calls. The complete SAGA Core API [9] provides roughly 200 distinct method calls. The SOAP rendering of the Amazon EC2 cloud interface provides, approximately 30 method calls (and similar for other Amazon Cloud interfaces, such as Eucalyptus [7]). The number of calls provided by these interfaces is no guarantee of simplicity of use, but is a strong indicator of the extent of system semantics exposed. But to a first approximation, (simplicity of) interface determines the programming models that can be supported. Thus there is the classical trade-off between simplicity and completeness.

VII. Conclusion and Some Future Directions

SAGA-MapReduce demonstrates how to decouple the development of applications from the deployment and details of the run-time environment. It is critical to reiterate that using this approach applications remain insulated from any underlying changes in the infrastructure – not just Grids and different middleware layers, but also different systems with very different semantics and characteristics, whilst being exposed to the important distributed functionality. MapReduce has trivial data-parallelism, so in the near future we will develop applications with non-trivial data-access, transfer and scheduling characteristics and requirements, and deploy different parts on different underlying infrastructure guided by optimal performance. EC2 and Eucalyptus although distinct systems have similar interfaces; we will work towards developing SAGA based applications that can use very different infrastructures, e.g., Google's AppEngine, such that SAGA-MapReduce uses Google's Cloud infrastructure. Finally, it is worth mentioning that computing in the Clouds for this project cost us upwards of $300 to perform these experiments on EC2.

VIII. Acknowledgments

SJ acknowledges UK EPSRC grant number GR/D0766171/1 for supporting SAGA and the e-Science Institute, Edinburgh for the research theme, "Distributed Programming Abstractions". SJ also acknowledges financial support from NSF-Cybertools and NIH-INBRE Grants. This work would not have been possible without the efforts and support of other members of the SAGA team. In particular, SAGA-MapReduce was originally written by Chris and Michael Miceli, as part of a Google Summer of Code Project, with assistance from Hartmut Kaiser. We also thank Hartmut for great support during the testing and deployment phases of this project. We are greatful to Dmitrii Zagorodnov (UCSB) and Archit Kulshrestha (CyD group, CCT) for the support in deployment with Eucalyptus. KS would like to acknowledge the LSU Networking Infrastructure & Research IT Enablement team for their help and support. We also acknowledge internal resources of the Center for Computation & Technology (CCT) at LSU and computer resources provided by LONI/TeraGrid for QueenBee.

References

[1] C. Miceli et al, Programming Abstractions for Data-Intensive Computing on Clouds and Grids, submitted to International Workshop on Cloud Computing (Cloud 2009) held in conjunction with CCGrid 2009, Shangai. Draft available at http://www.cct.lsu.edu/~sjha/publications/saga_data_intensive.pdf.

[2] S. Jha et al Developing Large-Scale Adaptive Scientific Applications with Hard to Predict Runtime Resource Requirements, *Proceedings of TeraGrid08*, available at http://tinyurl.com/5du32j.

[3] SAGA Web-Page: http://saga.cct.lsu.edu.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[5] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: the montage example. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[6] S Jha et al. Design and Implementation of Network Performance Aware Applications Using SAGA and Cactus. In *Accepted for 3rd IEEE Conference on eScience2007 and Grid Computing, Bangalore, India.*, 2007.

[7] Daniel Nurmi *et al*. The Eucalyptus Open-source Cloud-computing System. October 2008.

[8] S. Jha, A. Merzky, and G. Fox. Using Clouds to Provide Grids Higher-Levels of Abstraction and Explicit Support for Usage Modes. *Accepted in Concurrency and Computation: Practice and Experience*, 2009.

[9] T Goodale *et al* . A Simple API for Grid Applications (SAGA). http://www.ogf.org/documents/GFD.90.pdf.