Programming Abstractions for Data Intensive Computing on Clouds and Grids

Chris Miceli[12], Michael Miceli[12], Shantenu Jha[123], Hartmut Kaiser[1], Andre Merzky[1],
[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—**MapReduce has emerged as an important data-parallel programming model for data-intensive computing – for Clouds and Grids. However most if not all implementations of MapReduce are coupled to a specific infrastructure. SAGA is a high-level programming interface which provides the ability to create distributed applications in an infrastructure independent way. In this paper, we show how MapReduce has been implemented using SAGA and demonstrate its interoperability across different distributed platforms – Grids, Cloud-like infrastructure and Clouds. We discuss the advantages of programmatically developing MapReduce using SAGA, by demonstrating that the SAGA-based implementation is infrastructure independent whilst still providing control over the deployment, distribution and run-time decomposition. The ability to control the distribution and placement of the computation units (workers) is critical in order to implement the ability to move computational work to the data. This is required to keep data network transfer low and in the case of commercial Clouds the monetary cost of computing the solution low. Using data-sets of size up to 10GB, and up to 10 workers, we provide detailed performance analysis of the SAGA-MapReduce implementation, and show how controlling the distribution of computation and the payload per worker helps enhance performance.**

I. Introduction

The case for effective programming abstractions and patterns is not new in computer science. Coupled with the heterogeneity and evolution of large-scale distributed systems, the fundamentally distributed nature of data and its exponential increase – collection, storing, processing of data, it can be argued that there is a greater premium than ever before on abstractions at multiple levels.

Although Clouds are a nascent infrastructure, with the force-of-industry behind their development and uptake (and not just the hype), their impact can not be ignored. Specifically, with the emergence of Clouds as important distributed computing infrastructure, we need abstractions that can support existing and emerging programming models for Clouds. Inevitably, the unified concept of a Cloud is evolving into different flavours and implementations on the ground. For example, there are already multiple implementations of Google's Bigtable, such as HyberTable, Cassandara, HBase. There is bound to be a continued proliferation of such Cloud-like infrastructure; this is reminiscent of the plethora of grid middleware distributions. Thus application-level support and inter-operability with different Cloud infrastructure is critical. And issues of scale aside, the transition of existing distributed programming models and styles, must be as seamless and as least disruptive as possible, else it risks engendering technical and political horror stories reminiscent of Globus, which became a disastrous by-word for everything wrong with the complexity of Grids.

Application-level programming and data-access patterns remain essentially invariant on different infrastructure. Thus the ability to support application specific data-access patterns is both useful and important [1]. There are however, infrastructure specific features – technical and policy, that need to be addressed. For example, Amazon, the archetypal Cloud System has a well-defined cost model for data transfer across *its* network. Hence, Programming Models for Clouds must be cognizant of the requirement to programmatically control the placement of compute and data relative to each other – both statically and even dynamically. It is not that traditional Grids applications do not have this interesting requirement, but that, such explicit support is typically required for very large-scale and high-performing applications. In contrast, for most Cloud applications such control is required in order to ensure basic cost minimization, i.e., the same computational task can be priced very differently for possibly the same performance. These factors and trends place a critical importance on effective programming abstractions for data-intensive applications for both Clouds and Grids and importantly in bridging the gap between the two. Any *effective* abstraction will be cognizant and provide at least the above features, viz., relative compute-data placement, application-level patterns and interoperabilty.

The primary aim of this work is to establish that SAGA – the Simple API for Grid Applications, is an *effective* abstraction that can support different programming models and is usable on traditional (Grids) and emerging (Clouds) distributed infrastructure. Our approach is to begin with a well understood data-parallel programming pattern (MapReduce) and implement it using SAGA – a standard programming interface. SAGA has been demonstrated to support distributed HPC programming models and applications effectively; it is an important aim of this work to verify if SAGA has the expressiveness to implement data-parallel programming and is capable of supporting acceptable levels of performance (as compared with native implementations of MapReduce). After this conceptual validation, our aim is to use the *same* implementation of SAGA-MapReduce on Cloud systems, and test for inter-operability between different flavours of Clouds as well as between Clouds and Grids.

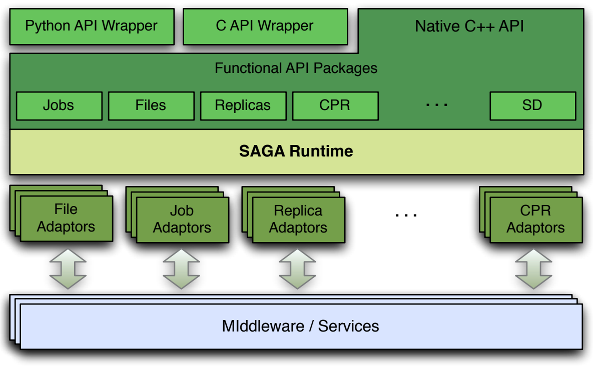

Fig. 1: In addition to the programmer's interface, the other important components of the landscape are the SAGA engine, and functional adaptors.

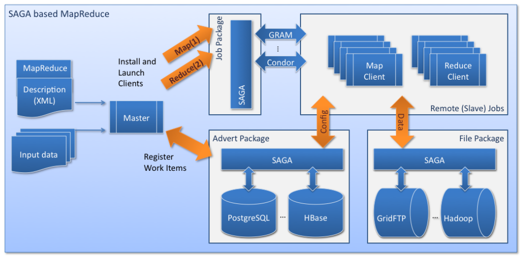

Fig. 2: High-level control flow diagram for SAGA-MapReduce. SAGA uses a master-worker paradigm to implement the MapReduce pattern. The diagram shows that there are several different infrastructure options to a SAGA based application;

II. SAGA

SAGA [11] is 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 [10, 2], tool-kits to manage distributed applications as well as implement abstractions that support commonly occurring programming, access and usage patterns.

Fig. 1 provide a view of the SAGA landscape, 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. Less common, but equally critical, wherever they were required, is the 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. Patterns for Data-Intensive Computing: MapReduce and All-Pairs

In this paper we will demonstrate the use of SAGA in implementing well known programming patterns for data intensive computing. Specifically, we have implemented MapReduce and the All-Pairs [4] patterns, and have used their implementations in SAGA to to solve commonly encountered genomic tasks. We have also developed real scientific applications using SAGA based implementations of these patterns: multiple sequence alignment can be orchestrated using the SAGA-All-pairs implementation, and genome searching can be implemented using SAGA-MapReduce.

MapReduce: MapReduce [9] is a programming framework which supports applications which operate on very large data sets on clusters of computers. MapReduce relies on a number of capabilities of the underlying system, most related to file operations. Others are related to process/data allocation. One feature worth noting in MapReduce is that the ultimate dataset is not on one machine, it is partitioned on multiple machines distributed over a Grid. Google uses their distributed file system (Google File System) to keep track of where each file is located. Additionally, they coordinate this effort with Bigtable.

SAGA-MapReduce Implementation: We have recently implemented MapReduce in SAGA, where the system capabilities required by MapReduce are usually not natively supported. Our implementation 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 our implementation is relatively more complex – it needs to add system semantic capabilities at some level, and it is inherently slower – as it is difficult to reproduce system-specific optimizations to work generically. 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.

The overall architecture of the SAGA-MapReduce implementation is shown in Fig. 2. 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 of the intermediate results, launching and coordinating the map and reduce workers, etc. as 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 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. Fig.3 shows a very simple example of a MapReduce application to count the word frequencies in the input data set. The user provided functions `map` (line 14) and `reduce` (line 25) are invoked by the MapReduce framework during the map and reduce steps. The framework provides the URL of the input data chunk file to the `map` function, which should call the function `emitIntermediate` for each of the generated output key/value pairs (here the word and it's count, i.e. '1', line 19). During the reduce step, after the data has been sorted, this output data is passed to the `reduce` function. The framework passes the key and a list of all data items which have been associated with this key during the map step. The reduce step calls the `emit` function (line 34) for each of the final output elements (here: the word and its overall count). All key/value pairs that are passed to `emit` will be combined by the framework into a single output file.

SAGA MapReduce Word Count Algorithm

```
 1  // Counting words using SAGA-MapReduce
 2  using namespace std;
 3  using namespace boost;

 4  class CountWords
 5    : public MapReduceBase<CountWords> {
 6  public:
 7    CountWords(int argc, char *argv[])
 8      : MapReduceBase<CountWords>(argc, argv)
 9    {}

10    // Separate input into words
11    // Input:  url of input chunk (chk)
12    // Output: separated words and associated
13    //         data (here: '1')
14    void map(saga::url chk) {
15      using namespace boost::iostreams;
16      stream<saga_file_device> in(chk.str());
17      string elem;
18      while(in >> elem)
19        emitIntermediate(elem, "1");
20    }

21    // Count words
22    // Input:  word to count (key)
23    //         list of associated data items
24    // Output: words and their count
25    void reduce(string const& key,
26      vector<string> const& values) {
27      typedef vector<string>::iterator iter;

28      int result = 0;
29      iter end = values.end();
30      for (iter it = values.begin();
31           it != end; ++it) {
32        result += lexical_cast<int>(*it);
33      }
34      emit(key, lexical_cast<string>(result));
35    }
36  };
```

Fig. 3: Counting word frequencies using SAGA-MapReduce. This is the worker-side code.

As shown in Fig. 2 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 to launch the jobs 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 (FS) such as Hadoop-FS (HDFS). All file transfer operations are based on the SAGA file package, which supports a range of different FS and transfer protocols, such as local-FS, Globus/GridFTP, KFS, and HDFS.

All-Pairs: As the name suggests, All-Pairs involve comparing every element in a set to every element in another set. Such a pattern is pervasive and finds utility in many domains – including testing the validity of an algorithm, or finding an anomaly in a configuration. For example, the accepted method for testing the strength of a facial recognition algorithm is to use All-Pairs testing. This creates a similarity matrix, and because it is known which images are the same person, the matrix can show the accuracy of the algorithm.

SAGA All-Pairs Implementation: SAGA All-pairs implementation is very similar to SAGA-MapReduce implementation. The main difference is in the way jobs are run and how the data are stored. In SAGA-MapReduce the final data is stored on many machines – if there is a DFS available, whereas SAGA All-pairs uses the database to also store information about the job. We decided to do this because all data must be available to be useful. We demonstrate the SAGA All-Pairs abstraction using the HDFS and GridFTP to not only show that SAGA allows for many different configurations, but also to see how these different configurations behave. We have also used a distributed data-store – specifically HBase (Yahoo's implementation of Bigtable) in lieu of the traditional Advert Service to store the end-results.

Multiple Sequence Alignment Using All-Pairs: An important problem in Bioinformatics – Multiple Sequence Alignment (MSA), can be reformulated to use All-Pairs pattern. It uses a comparison matrix as a reference to compare many fragment genes to many base genes. Each fragment is compared to every base gene to find the smallest distance – maximum overlap. Distance is computed by summing up the amount of similarity between each nucleotide of the fragment to each one in the base. This is done starting at every point possible on the base.

IV. Interfacing SAGA to Cloud-like Infrastructure: The role of Adaptors

As alluded to, there is a proliferation of Clouds and Cloud-like systems, but it is important to remember that "what constitutes or does not constitute a Cloud" is not universally agreed upon. However there are several aspects and attributes of Cloud systems that are generally agreed upon [5]. Here we will by necessity limit our discussion to two type of distributed file-systems (HDFS and KFS) and two types of distributed structured-data store (Bigtable and HBase). We have developed SAGA adaptors for these, have used SAGA-MapReduce (and All-Pairs) seamlessly on these infrastructure.

HDFS and KFS: HDFS is a distributed parallel fault tolerant application that handles the details of spreading data across multiple machines in a traditional hierarchical file organization. Implemented in Java, HDFS is designed to run on commodity hardware while providing scalability and optimizations for large files. The FS works by having one or two namenodes (masters) and many rack-aware datanodes (slaves). All data requests go through the namenode that uses block operations on each data node to properly assemble the data for the requesting application. The goal of replication and rack-awareness is to improve reliability and data retrieval time based on locality. In data intensive applications, these qualities are essential. KFS (also called CloudStore) is an open-source high-performance distributed FS implemented in C++, with many of the same design features as HDFS.

Bigtable and HBase: Bigtable [6] is a type of database system created by Google to have better control over scalability and performance than other databases. The main difference is that it is meant to store extremely large datasets, into the petabytes, over thousands of machines. It is well integrated with MapReduce. Due to the success of Bigtable, HBase was developed as an open source alternative to Bigtable for use with Hadoop. Both HBase and Bigtable split up large tables and replicate them over many machines to avoid node failure.

There exist many other implementations of both distributed FS (such as Sector) and of distributed data-store (such as Cassandra and Hybertable); for the most part they are variants on the same theme technically, but with different language and performance criteria optimizations. Hypertable is an open-source implementation of Bigtable; Cassandra is a Bigtable clone but eschews an explicit coordinator (Bigtable's Chubby, HBase's HMaster, Hypertable's Hyperspace) for a P2P/DHT approach for data distribution and location and for availability. In the near future we will be providing adaptors for Sector[1] and Cassandra[2]. And although Fig. 1 explicitly maps out different functional areas for which SAGA adaptors exist, there can be multiple adaptors (for different systems) that implement that functionality; the SAGA run-time dynamically loads the correct adaptor, thus providing both an effective abstraction layer as well as an interesting means of providing interoperability between different Cloud-like infrastructure. As testimony to the power of SAGA, the ability to create the relevant adaptors in a lightweight fashion and thus extend applications to different systems with minimal overhead is an important design feature and a significant requirement so as to be an effective programming abstraction layer.

[1]http://sector.sourceforge.net/

[2]http://code.google.com/p/the-cassandra-project/

V. SAGA: An interface to Clouds and Grids

The total time to completion (T_c) of a SAGA-MapReduce job, can be decomposed into three primary components: t_{pp} defined as the time for pre-processing – which in this case is the time to chunk into fixed size data units, and to possibly distribute them. This is in some ways the overhead of the process. 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. t_{coord} is indicative of the time that it takes to assign chunks to workers and scales as the number of workers increases. In general:

$$T_c = t_{pp} + t_{comp} + t_{coord} \tag{1}$$

To establish the effectiveness of SAGA as a mechanism to develop distributed applications, and the ability of SAGA-MapReduce to be provide flexibility in distributing compute units, we have designed the following experiment set[3] :

1) Both SAGA-MapReduce workers (compute) and data-distribution are local. Number of workers vary from 1 to 10, and the data-set sizes varying from 1 to 10GB.
2) SAGA-MapReduce workers compute local (to master), but using a distributed FS (HDFS)
3) Same as Exp. #2, but using a different distributed FS (KFS); the number of workers varies from 1-10
4) SAGA-MapReduce using distributed compute (workers) and distributed file-system (KFS)
5) Distributed compute (workers) but using local file-systems (using GridFTP for transfer)

SAGA-MapReduce on Grids: We begin with the observation that the efficiency of SAGA-MapReduce is pretty close to 1, actually better than 1 – like any good (data) parallel applications should be. For 1GB data-set, T_c = 659s and for 10GB T_c = 6286s. The efficiency remains at or around 1, even when the compute is distributed over two machines: 1 worker at each site: T_c = 672s, T_c = 1081s and T_c =2051s for 1, 2 and 4GB respectively; this trend is valid even when the number of workers per site is more than 1.

Fig. 4 plots the T_c for different number of active workers on different data-set sizes; the plots can be understood using the framework provided by Equation 1. For each data-set (from 1GB to 10GB) there is an overhead associated with chunking the data into 64MB pieces; the time required for this scales with the number of chunks created. Thus for a fixed chunk-size (as is the case with our set-up), t_{pp} scales with the data-set size. As the number of workers increases, the payload per worker decreases and this contributes to a decrease in time taken, but this is accompanied by a concomitant increase in t_{coord}. However, we will establish that the increase in t_{coord} is less than the decrease in t_{comp}. Thus the curved decrease in T_c can be explained by a speedup due to lower payload as the number of workers increases whilst at the same time

[3]We have also distinguished between SAGA All-Pairs using Advert Service versus using HBase or Bigtable as distributed data-store, but due to space constraints we will report results of the All-Pairs experiments elsewhere.

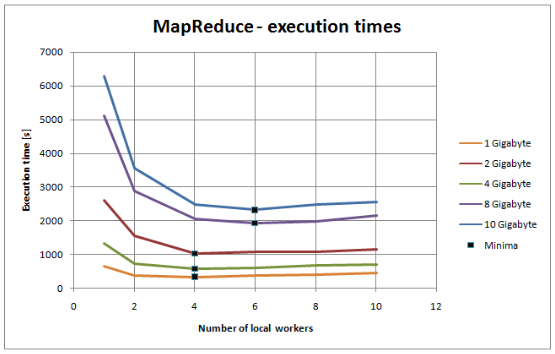

Fig. 4: Plots showing how the T_c for different data-set sizes varies with the number of workers employed. For example, with larger data-set sizes although t_{pp} increases, as the number of workers increases the workload per worker decreases, thus leading to an overall reduction in T_c. The advantages of a greater number of workers is manifest for larger data-sets.

the t_{coord} increases; although the former is linear, due to increasing value of the latter, the effect is a curve. The plateau value is dominated by t_{pp} – the overhead of chunking etc, and so increasing the number of workers beyond a point does not lead to a further reduction in T_c .

To take a real example, we consider two data-sets, of sizes 1GB and 5GB and vary the chunk size, between 32MB to the maximum size possible, i.e., chunk sizes of 1GB and 5GB respectively. In the configuration where there is only one chunk, t_{pp} should be effectively zero (more likely a constant), and T_c will be dominated by the other two components – t_{comp} and t_{coord}. For 1GB and 5GB, the ratio of T_c for this boundary case is very close to 1:5, providing strong evidence that the t_{comp} has the bulk contribution, as we expect t_{coord} to remain mostly the same, as it scales either with the number of chunks and/or with the number of workers – which is the same in this case. Even if t_{coord} does change, we do not expect it to scale by a factor of 5, while we do expect t_{comp} to do so.

SAGA-MapReduce on Cloud-like infrastructure: Accounting for the fact that time for chunking is not included, Yahoo's MapReduce takes a factor of 2 less time than SAGA-MapReduce (Fig. 5). This is not surprising, as SAGA-MapReduce implementations have not been optimized, e.g., SAGA-MapReduce is not multi-threaded. Experiment 5 (Table I) provides insight into performance figure when the same number of workers are available, but are either all localized, or are split evenly between two similar but distributed machines. It shows that to get lowest T_c, it is often required to both distribute the compute and lower the workload per worker; just lowering the workload per worker is not good enough as there is still a point of serialization (usually local I/O). When coupled with the advantages of a distributed FS, the ability to both distribute compute and data provides additional performance advantage, as shown by the values of T_c for both distributed compute and DFS cases in Table I.

SAGA-MapReduce on Clouds: Thanks to the low overhead of developing adaptors, SAGA has been deployed on three Cloud Systems – Amazon, Nimbus [7] and Eucalyptus [8] (we have a local installation of Eucalyptus, referred to as GumboCloud). On EC2, we created custom virtual machine (VM) image with preinstalled SAGA. For Eucalyptus and Nimbus, 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.

There are several aspects to Cloud Interoperability. A simple form of interoperability – more akin to inter-changeable – is that any application can use either of the three Clouds systems without any changes to the application: the application simply needs to instantiate a different set of security credentials for the respective runtime environment, aka cloud. Interestingly, SAGA provides this level of interoperability quite trivially thanks to the adaptors.

By almost trivial extension, SAGA also provides Grid-Cloud interoperability, as shown in Fig. 6 and 7, where exactly the same interface and functional calls lead to job submission

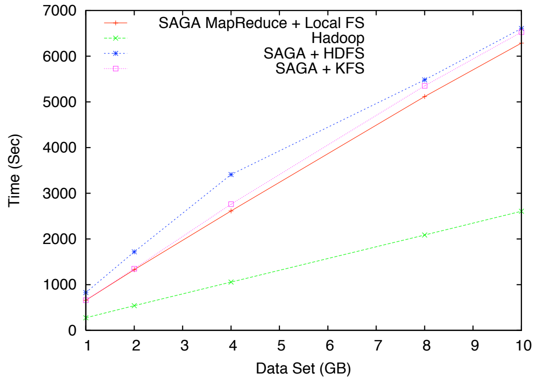

Fig. 5: T_c for SAGA-MapReduce using one worker (local to the master) for different configurations. The label "Hadoop" represents Yahoo's MapReduce implementation; T_c for Hadoop is without chunking, which takes several hundred sec for larger data-sets. The "SAGA MapReduce + Local FS" corresponds to the use of the local FS on Linux clusters, while the label "SAGA + HDFS" corresponds to the use of HDFS on the clusters. Due to simplicity, of the Local FS, its performance beats distributed FS when used in local mode.

Configuration		data size	work-load/worker	T_c
compute	data	(GB)	(GB/W)	(sec)
local	local-FS	1	0.1	466
distributed	local-FS	1	0.1	320
distributed	DFS	1	0.1	273.55
local	local-FS	2	0.25	673
distributed	local-FS	2	0.25	493
distributed	DFS	2	0.25	466
local	local-FS	4	0.5	1083
distributed	local-FS	4	0.5	912
distributed	DFS	4	0.5	848

TABLE I: Table showing T_c for different configurations of compute and data. The two compute configurations correspond to the situation where all workers are either placed locally or workers are distributed across two different resources. The data configurations arise when using a single local FS or a distributed FS (KFS) with 2 data-servers. It is evident from performance figures that an optimal value arises when distributing both data and compute.

on Grids or on Clouds. Although syntactically identical, the semantics of the calls and back-end management are somewhat different. For example, for Grids, a `job_service` instance represents a live job submission endpoint, whilst for Clouds it represents a VM instance created on the fly. It takes SAGA about 45 seconds to instantiate a VM on Eucalyptus, and about 90 seconds on EC2. Once instantiated, it takes about 1 second to assign a job to a VM on Eucalyptus, or EC2. It is a configurable option to tie the VM lifetime to the `job_service` object lifetime, or not.

We have also deployed SAGA-MapReduce to work on Cloud platforms. It is critical to mention that the SAGA-MapReduce code did not undergo any changes whatsoever. 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 systems. In our current implementation, the VMs hosting the master and workers share the same ssh credentials and a shared file-system (using sshfs/FUSE). Application deployment and configuration (as discussed above) are also performed via that sshfs. Due to space limitations we will not discuss the performance data of SAGA-MapReduce with different data-set sizes and varying worker numbers.

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. 6: Job launch via Gram

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. 7: Job launch via VM

VI. Conclusion

We have demonstrated the power of SAGA as a programming interface and as a mechanism for codifying computational patterns, such as MapReduce and All-Pairs. Patterns capture a dominant and recurring computational mode; by providing explicit support for such patterns, end-users and domain scientists can reformulate their scientific problems/applications so as to use these patterns. This provides further motivation for abstractions at multiple-levels. We have shown the power of abstractions for data-intensive computing by demonstrating how SAGA, whilst providing the required controls and supporting relevant programming models, can decouple the development of applications from the deployment and details of the run-time environment.

VII. 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”. This work would not have been possible without the efforts and support of other members of the SAGA team. We also acknowledge internal resources of the Center for Computation & Technology (CCT) at LSU and computer resources provided by LONI. Michael Miceli was supported by a Google Summer of Code grant.

References

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

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

[3] SAGA Project Page, http://saga.cct.lsu.edu.

[4] All-Pairs: An Abstraction for Data Intensive Cloud Computing, Christopher Moretti et al, IEEE IPDPS, 2008.

[5] R. Buyya, and et al, *Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities*, Keynote Paper, Proc. of the 10th IEEE Intl Conf. on HPCC, Sept. 25-27, 2008, China.

[6] Bigtable: A Distributed Storage System for Strctured Data, Fay Chang, and et al, OSDI'06: 7th Symp. on Operating System Design and Implementation, Nov 06.

[7] NIMBUS http://workspace.globus.org/.

[8] Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems (EUCALYPTUS), http://eucalyptus.cs.ucsb.edu/.

[9] Jeffrey Dean and Sanjay Ghemawat. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Sys tems Design & Implementation*, pages 137–150, Berkeley, CA, USA.

[10] 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.

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