Understanding Application-Level Interoperability: Scaling-Out MapReduce Over High-Performance Grids and Clouds

Saurabh Sehgal[1], Miklos Erdelyi[3,4], Andre Merzky[1], Shantenu Jha[*1,2]
[1] *Center for Computation & Technology, Louisiana State University, USA*
[2] *Department of Computer Science, Louisiana State University, USA*
[3] *Department of Computer Science & Systems Technology, University of Pannonia, Veszprem, Hungary*
[4] *Computer & Automation Research Institute of the Hungarian Academy of Sciences*
[*] *Contact Author* sjha@cct.lsu.edu

Abstract

Application-level interoperability is defined as the ability of an application to utilize multiple distributed heterogeneous resources. Such interoperability is becoming increasingly important with increasing volumes of data, multiple sources of data as well as resource types. The primary aim of this paper is to understand different ways and levels in which application-level interoperability can be provided across distributed infrastructure. Our approach is: (i) Given the simplicity of MapReduce, its wide-spread usage, and its ability to capture the primary challenges of developing distributed applications, use MapReduce as the underlying exemplar; we develop an interoperable implementation of MapReduce using SAGA – an API to support distributed programming, (ii) Using the canonical wordcount application that uses SAGA-based MapReduce, we investigate its scale-out across clusters, clouds and HPC resources, (iii) Establish the execution of wordcount application using MapReduce and other programming models such as Sphere concurrently. SAGA-based MapReduce in addition to being interoperable across different distributed infrastructures, also provides user-level control of the relative placement of compute and data. We provide performance measures and analysis of SAGA-MapReduce when using multiple, different, heterogeneous infrastructures concurrently for the same problem instance.

1. Introduction

There are numerous scientific applications that either currently utilize, or need to utilize data and resources distributed over vast heterogeneous infrastructures and networks with varying speeds and characteristics. Many scientific applications are, however, designed with a specific infrastructure; such dependence and tight-coupling to specific resource types and technologies is, in a heterogeneous distributed environment, not an optimal design choice. In order to leverage the flexibility of distributed systems and to gain maximum run-time performance, applications must shed their dependence on single infrastructure for all of their computational and data processing needs. For example, the Sector/Sphere data cloud is exclusively designed to support data-intensive computing on high speed networks, while others, like the distributed file systems GFS/HDFS, assume limited bandwidth among infrastructure nodes [14, 10]. Thus, for applications to efficiently utilize heterogeneous environments, abstractions must be developed for the efficient utilization of and orchestration across such distinct distributed infrastructure.

In addition to issues of performance and scale addressed in the previous paragraph, the transition of existing distributed programming models and applications to emerging and novel distributed infrastructure must be as seamless and as non-disruptive as possible. A fundamental question at the heart of all these 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.

We define Application Level Interoperability (ALI) as a feature that arises, when other than say compiling, there are no further changes required of the application to utilize a new platform. The complexity of providing ALI varies and depends upon the application under consideration. For example, it is somewhat easier for simple "distribution unaware" applications to utilize multiple het-

erogeneous distributed environments, than for applications where multiple distinct and possibly distributed components need to coordinate and communicate.

The Case for Application-level Interoperability. In either case, ALI is not only of theoretical interest. There exist many applications which involve large volumes of data on distributed heterogeneous resources and which benefit from ALI. For examples, the Earth System Grid [1] involves peta to exa-bytes of data, and one thus cannot move all data (given current transfer capabilities), nor compute at a centralized location. Thus there is an imperative to operate on the data *in situ*, which in turn involves computation across heterogeneous distributed platforms as part of the same application.

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. The LEAD application, as part of the VGrADS project provides a prominent example[1]. Due to different data-compute affinity requirement amongst the tasks, some workloads might be better placed on a cloud [2], 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.

Last, but not least, in the rapidly evolving world of clouds, there is as of yet little business motivation for cloud providers to define, implement and support new/standard interfaces. Consequently, there is a case to be made that by providing ALI, such barriers can be overcome and cross-cloud applications can be easily achieved.

Currently, many programming models and abstractions are tied to a specific back-end infrastructure. For example, Google's MapReduce [12], which is tied to Google's file system, or Sphere[15] which is linked to the Sector file system. It is often the case that an application maybe significantly better suited to a specific programming model; similarly a specific programming model maybe optimised for a specific infrastructure. However, where this is not necessarily the case, or more importantly, when different applications or programming models can utilise "non-native" infrastructure, the ability to mix-and-match across the layers of applications, programming models and infrastructure should be supported. Ideally, an application should be able to utilize any programming model, and any programming model should be executable on any underlying infrastructure. Thus there is a need to investigate interoperability of different programming models for the same application on different systems.

We will work with MapReduce and an application based on MapReduce– the canonical wordcount application. We use SAGA — Simple API for Grid Applications" (see Sec. 2) as the programming system for distributed applications. In Ref. [18], we implemented a MapReduce based wordcount application using SAGA. We demonstrated that the SAGA-based implementation is infrastructure independent, whilst still providing control over deployment, distribution, run-time decomposition and data/compute co-location. We demonstrated that SAGA-MapReduce is interoperable on traditional (grids) and emerging (clouds) distributed infrastructure *concurrently and cooperatively towards a solution of the same problem instance.*

The primary focus of this paper is to understand, demonstrate and investigate different types of ALI. We build upon and use SAGA-based MapReduce as an exemplar to discuss multiple levels and types of interoperability that can exist between infrastructures. We use SAGA-MapReduce and SAGA-based Sphere on different infrastructure. We will also show that our approach to ALI helps break the coupling between programming models and infrastructure on the one hand, whilst providing empirically-driven insight about the performance of an application with different programming models.

This paper is structured as follows: Section 2 gives a short overview over those SAGA extensions which enable specifically the ALI work discussed in this paper. Section 3 describes our SAGA-MapReduce implementation. Section 4 discusses the different levels of ALI we investigate and demonstrate, with more details on the experiments provided in Section 5. Section 6 concludes the paper with a discussion of the results.

2. SAGA

The SAGA [19, 16] programming system provides a high level API that forms a simple, standard and uniform interface for the most commonly required

[1] `http://vgrads.rice.edu/presentations/VGrADS_overview_SC08pdf.pdf`

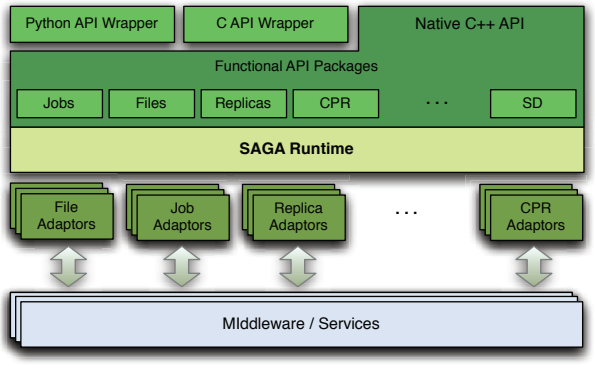

Figure 1: The SAGA run-time engine dynamically dispatches high level API calls to a variety of middlewares.

distributed functionality. SAGA can be used to program distributed applications [3, 4] or tool-kits to manage distributed applications [17], as well as implement abstractions that support commonly occurring programming, access and usage patterns.

Fig. 1 provides an overview of the SAGA programming system's architecture. The SAGA API covers job submission, file access and transfer, as well as logical file management. Additionally there is support for Checkpoint and Recovery (CPR), Service Discovery (SD), and other areas. The API is implemented in C++ and Java, with Python supported as a wrapper. *saga_core* is the main library, which provides dynamic support for run-time environment decision making through loading relevant adaptors. We will not discuss SAGA internals here; details can be found elsewhere [5, 16].

2.1. Interfacing SAGA to Grids and Clouds

SAGA was originally developed primarily for compute-intensive grids. [18] 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 is in part due to the fact that, at least at the application level, much of the "distributed functionality" required for data-intensive applications remains the same. How the respective functionality for grid systems and for EC2 based cloud environments is provided in SAGA is also documented in [18]. We extend SAGA Sector-Sphere [15].

2.1.1. Sector-Sphere Adaptors: Design and Implementation

Sector and Sphere is a cloud framework specifically designed for writing applications able to utilize the stream processing paradigm. Sector is a distributed file system that manages data across physical compute nodes at the file level, and provides the infrastructure to manipulate data. Sphere, on the other hand, provides the framework to utilize the stream processing paradigm for processing the data residing on Sector. The Sphere system is composed of Sphere Processing Engines (SPEs) running on the same physical nodes as the Sector file system.

Applications that utilize the stream processing paradigm define a single common function (aka kernel) that is applied to segments of a given data set. When the application invokes Sphere to process data on Sector, the Sphere system retrieves the stream of data, segments the data and assigns chunks of these segments to the available SPEs for processing.

Sphere allows the user to encode the kernel function in a dynamically linked library written against the Sphere APIs. The SPEs apply this user defined function to its assigned segments and write the processing results back to files in Sector. This stream of output files can be retrieved by the user from Sector, after the processing is complete.

SAGA adaptor overhead. We execute a simple experiment to measure the overhead introduced by submitting Sphere jobs and Sector file operations through SAGA. The Sphere kernel function accepts a buffer of text and utilizes the Sphere framework to hash words into Sphere buckets, using the first letter as the key. One Gigabyte of text data was uploaded to the Sector file system for this test. Furthermore, traces were implemented in the adaptors to measure the exact time spent in SAGA processing and translation before the raw Sphere APIs were called. As seen in Table 1, the SAGA overhead when compared to the overall execution time of the application, is relatively small. After the Sphere API is called through the SAGA adaptor, the execution time can be fully attributed to Sphere.

Thanks to the low overhead of developing SAGA adaptors, we were able to implement the Sector file adaptor, and the Sphere job adaptor for applications to utilize the stream processing paradigm through SAGA. The enhancement of

	total	abs. overhead	rel. overhead
	2.63 min	0.35 min	7.5 %
	3.01 min	0.56 min	5.4 %
	3.07 min	0.37 min	8.3 %
	3.18 min	0.39 min	8.2 %
	3.68 min	0.35 min	10.5 %
	3.76 min	0.48 min	7.8 %
	4.93 min	0.44 min	11.2 %
	5.03 min	0.52 min	9.7 %
	5.33 min	0.38 min	14.0 %
	6.42 min	0.52 min	12.3 %
Mean	4.10 min	0.44 min	9.5 %
Stdev	1.25	0.08	2.6

Table 1: Adaptor overhead measurements from processing 8GB of data with 8 SPEs running the wordcount application on 8 physical nodes on Poseidon (a LONI cluster). The measurement is repeated ten times; all times are in minutes.

SAGA-MapReduce, along with the implementation of the Sector/Sphere adaptors naturally gives us the opportunity to compare and study these two distinct programming models.

3. SAGA-based MapReduce

Given the simplicity of MapReduce, its widespread usage, and its ability to capture the primary challenges of developing distributed applications, use MapReduce as the underlying exemplar; we choose the SAGA-MapReduce implementation to compare both, different back-end systems (grids, clouds, and clusters) and different programming models (master/slave, Sector-Sphere streams). A simple wordcount application on top of SAGA-MapReduce has been used as a close-to-reality test case, and is described in Sec. 4.1.

3.1. SAGA-MapReduce Implementation

Our implementation of SAGA-MapReduce interleaves the core MapReduce 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 more complicated to extract performance, as some system-level semantics have to be recreated in the application space level. The fact that the implementation is single-threaded proved to be the primary current performance inhibitor to be addressed in the future. However, none of these

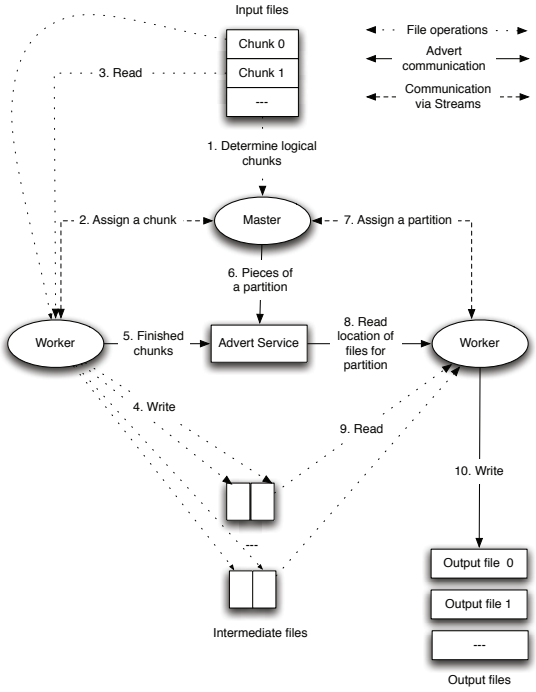

Figure 2: Schema of the SAGA-MapReduce components and their interaction.

complexities are exposed to the end-user, as they remain hidden within the framework.

SAGA-MapReduce exposes a simple interface which 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. – these are generically implemented by the framework. The application consists of two independent process types, 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 workers;
- 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 location of the sorted output files to the workers of the reduce step.

Any specific MapReduce instance is specified by a MR-JobDescription object in which the user speci-

fies Mapper and Reducer classes, input and output paths and data formats. The used InputFormat determines the logical partitions of the input data for the master – that information is then sent to idle workers. A RawRecordReader implementation interprets an InputChunk and provides a record iterator for the Mapper. It is possible to support any kind of data source for which a record oriented view exists, by writing a custom RawRecordReader. The output from the Mapper is further processed by the Partitioner which assigns emitted key/value pairs from the Mapper to Reducers. Finally, a RawRecordWriter writes output data to files.

The master process is readily available to the user and needs no modification to execute different map and reduce functions in the worker processes. The *functionality* for the different steps have to be provided by the user, which means the user has to write the C++ functions implementing the respective MapReduce kernels.

Both the master and the worker processes use the SAGA-API as an abstract interface to the 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 on any backend supported by SAGA (such as locally, globus/-GRAM, EC2, SSH, Condor etc.). The communication between the master and workers is ensured by using the SAGA advert package, abstracting an information database in a platform independent way, and the SAGA stream package, abstracting streaming data access between network endpoints. The master creates logical partitions of the data (referred to as chunking, analogous to Google's MapReduce), so the data-set does not have to be split and distributed manually. The input data can be located on any file system supported by SAGA, such as the local file system, or a distributed file system like HDFS or KFS [11].

3.2. Improving SAGA-based Map-Reduce Performance

The performance enhancements to the SAGA-MapReduce implementation as used and discussed in [18] are based on two important changes: (i) optimizing how sorting is done after the map phase, and (ii) using a serialized binary format instead of plain text for intermediate data storage (also available as input and output format). We expected those improvements to have a relatively small implementation effort, and to show a significant impact on performance. The first change means that, instead of having the master merge and then sort the intermediate data by key before entering the reduce phase, the workers buffer key/value pairs from the map phase and store them in sorted order on disk, doing an in-memory sort before writing. Also, since intermediate key/value pairs from a map worker are already sorted, the reduce workers need to only merge these pairs coming from different map workers, by applying the user-defined reduce function to the merged intermediate key and value list.

The second enhancement applies to the storage of the intermediate key/value pairs in a so-called *sequence file format*. This file format allows storing of serialized key/value objects which can be read and merged much faster in the reduce phase than text data, as there is no need for costly parsing. We used the Google Protocol Buffers library for implementing serialization [6]. The processing of input and output key/value pairs is further enhanced by minimizing unnecessary memory I/O operations, using a zero-copy scheme.

3.3. SAGA-MapReduce Set-Up

As with any application which concurrently spans multiple diverse resources or infrastructures, the coordination between the different application components becomes challenging. The SAGA-MapReduce implementation uses the SAGA advert API for that task, and can thus limit the a-priori information needed for bootstrapping the application: the compute clients (workers) require (i) the contact URL of the used advert service instance, and (ii) 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 by the master via command line parameters to the worker, at start-up time.

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

For example: in a typical configuration, three worker instances may be started; the first could be started via GRAM and PBS on qb.teragrid.org, the second on a pre-instantiated EC2 image (with known instance id), and the third on a dynamically deployed EC2 instance (no instance id given). Note that the start-up times for the individual workers

may vary over several orders of magnitudes, depending on PBS queue waiting time and virtual machine (VM) start-up time. That mechanism both minimizes time-to-solution, and maximizes resilience against worker loss. A parameter controls the number of workers created per compute node; varying that parameter permits compute and communication interleaving, which may lead to an increase in the overall system utilization (even in the absence of precise knowledge of the target system).

Future enhancements to SAGA-MapReduce might include asynchronous master-worker communication, and off-loading coordination tasks from the advert service to a stream-based protocol. For example, information about the pieces of intermediate data to be processed in the reduce phase could be directly interchanged via streams. The advert service could be used for maintaining master state, in order to make the master failure-recoverable. The advert service itself is a candidate for performance engineering and optimization. For example, in case of SAGA-MapReduce we observed that the latency of the advert service used during computation has a significant impact on time-to-solution. The results reported herein have been obtained by employing a local advert service solely used for our experiments; using a shared capabilty, having several concurrent users can possibly double the time-to-solution in configurations having high coordination overhead, i.e., in case of a small chunk size or large number of workers.

4. Application Level Interoperability: Three-levels

The motivation of ALI across multiple, heterogeneous and distributed resources follows from large-scale scientific applications, such as the Earth Science Grid and LEAD. However for simplicity of treatment and to focus on the levels of interoperability, we will use a simple, self-contained application, that has also become the *canonical* MapReduce application-driver – wordcount.

4.1. Application: Wordcount

We use our own implementations of the well known wordcount application for our experiments. Wordcount has a well understood run-time and scaling behaviour, and thus serves us well for focusing the tests on the used frameworks and middlewares.

The MapReduce based wordcount implementation is described in [18]. For the SAGA-Sphere version of wordcount we implemented two kernel functions. The first one is responsible for hashing the words in the data set into different "buckets". The standard C++ collate hashing function was used for this purpose. The second kernel function reads each hash bucket, sorts the words in memory and outputs the final count of the words in the data set. For example, a file containing the words `('bread' 'bee' 'bee' 'honey')` would be hashed into buckets as `('bread' 'bee' 'bee')` and `('honey')`. The second kernel function would read these intermediate bucket files, sort the words, and produce the result `('bread 1', 'bee 2', 'honey 1')`. The Sphere system is responsible for assigning files for processing, synchronization, and writing output results back to Sector.

4.2. Interoperability Types

Using the wordcount application, we will demonstrate three types of application level interoperability. We outline them here:

4.2.1. Type I: Application Interoperability via adaptors

As discussed, SAGA provides the ability to load a wide-range of system-specific adaptors dynamically. Thus a simple form of interoperability, possibly specific to applications developed using SAGA, is that an application can use any distributed system without changes to the application, thus experiencing cloud-cloud or grid-cloud interoperability. We refer to this as Type I interoperability.

Thanks to the relative simplicity of developing SAGA adaptors, SAGA has been successful interfaced to three cloud systems – Amazon's EC2, Eucalyptus [13] (a local installation of Eucalyptus at LSU) and Nimbus [7]; and also to a multitude of grid based environments, including TeraGrid, LONI and NGS. SAGA based applications are thus inherently able to utilize this form of ALI.

4.2.2. Type II: Application Interoperability using programming models

Interoperability at a higher level than adaptors is both possible and often desirable. An application can be considered interoperable if it is able to switch between back-end specific programming models. We will discuss an example where the wordcount application is implemented so that it can utilize either a Sector-Sphere framework via SAGA,

or the SAGA-MapReduce framework for generic grid and cloud back-ends.

4.2.3. Type III: Application Interoperability using different programming models for concurrent execution

At another level, an application can also be considered interoperable when it executes multiple programming models *concurrently* over diverse back-ends; this provides an opportunity for infrastructure specific performance utilization. We demonstrate that a wordcount application uses both Sector-Sphere and SAGA-MapReduce when spanning multiple back-ends. The challenges of having different parts of an application execute concurrently using different programming models is conceptually different to loading different adaptors concurrently. Thus we describe this as a separate type of interoperability.

4.3. Experimental Setup

Simulations were performed on shared TeraGrid-LONI (Louisiana Optical Network Initiative) [8] resources running Globus and ssh; on GumboGrid, a small cluster at LSU running Eucalyptus; on Amazon's EC2; and on a bare 50 node cluster of the Hungarian Academy of Sciences; A significant set of experiments were performed on FutureGrid (`http://www.futuregrid.org`) and its IBM iDataPlex (India), which as 256/1024 nodes/cores (for full details see: `http://futuregrid.org/hardware`).

Jobs are started via the respectively available middleware, via SAGA's job API. Data exchange is either performed via streams, or via SAGA's file transfer API, which can dynamically switch between the various available protocols, such as Sector, GridFTP or HDFS.

For cloud environments, 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 Debian style Linux distributions (using `apt-get` for software installation), 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 preconfigured images).

For experiments in this paper, we prepared custom VM images with pre-installed prerequisites. We utilize preparation scripts solely for some fine tuning of parameters: for example, to deploy custom saga.ini files, or to ensure the finalization of service startups before application deployment.

Deploying SAGA-MapReduce framework and the wordcount application on different grids, clouds or clusters requires adapting the configuration to the specific environment. 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.

It must be noted that we utilized different SAGA-MapReduce versions for the described experiments: the work described in this paper spans more than 18 months, and the SAGA-MapReduce implementation has simply evolved over time. As our primary goal is to demonstrate interoperability, and not to document maximal performance, we consider those results valid nonetheless.

5. Experiments

5.1. Type I ALI: Interoperability via Adaptors

In Ref. [18], 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 work presented here extends this, and establishes that SAGA-MapReduce can provide cloud-cloud interoperability and cloud-grid interoperability. We performed 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 and 1:4, respectively.
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 TeraGrid) 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. 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. Any further analysis is considered out of scope for this paper.

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 `saga::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 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.

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_{pre}, t_{comp} and t_{coord}. Here t_{pre} is defined as pre-processing time, which covers the time to chunk the data into fixed size data units, to distribute them, and also to spawn the job. t_{pre} does not include the time required to start VM instances. 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.

Table 2 shows performance measurements for a variety of worker placement configurations. 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 indicated 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. Note that the spawning times depends on the number of VMs, even if it does not include the VM startup times.

Table 3 shows data from our interoperability tests. 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.

#workers		Data size	T_s	T_{sp}	$T_s - T_{sp}$
TG	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 2: Performance data for different configurations of worker placements.

We find that in our experiments 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 3 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.

# workers			Size	T_s	T_{sp}	$T_s - T_{sp}$
TG	AWS	Eucal.	(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 3: Performance data for different configurations of worker placements on TG, Eucalyptus-Cloud and EC2.

The original SAGA-MapReduce version (as used for the experiments presented above) physically chunked the input data files. Our evolved version, however, creates logical chunks (i.e., no file writing takes place). It is thus fair to compare their time-to-solution performance by subtracting the chunking time from the early-version's job completion time. Fig. 3 shows the thus corrected performance data for the early-version and enhanced version of SAGA-MapReduce. 8 workers were spawned via the SAGA SSH adaptor on 8 physical machines, data were exchanged through a shared NFS file system. The figure shows that the SAGA-MapReduce enhancements make a difference for larger data sets. This can be attributed to the fact that the more efficient shuffle phase implementation, which reduces disk I/O and CPU usage in the reduce phase by doing only a merge, outperforms the old implementation, which performed a merge-sort of all the intermediate output files.

5.2. Type II ALI: Application Performance Using SAGA-based Sphere and MapReduce

The various experiments described in this section have been performed in two configurations – determined by the distribution (or lack thereof) of the computational resources and data-access method. In the first configuration ("local"), all application processes used a single compute node, with local data access. In the second configuration ("dist."), the application processes were distributed over multiple physical nodes, and utilized remote data access techniques, such as distributed file-systems, or distributed file-transfer.

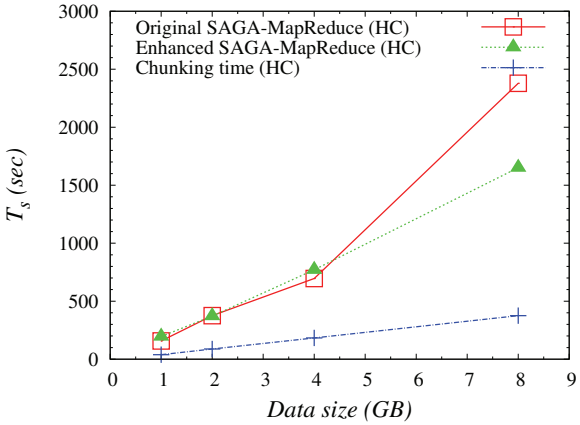

Figure 3: Comparison of enhanced SAGA-MR performance versus early-version of SAGA-MR on HC using 8 workers running on 8 physical machines. Jobs were launched via SSH and used NFS for file operations. Chunking data shown here is for the original SAGA-MapReduce.

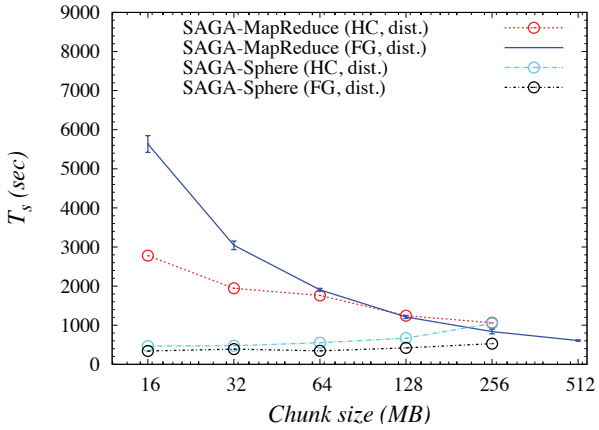

Figure 4: Performance of SAGA-MapReduce and SAGA-Sphere when varying chunk size while keeping the amount of processed data constant at 4GBs. Data and computation were distributed for these experiments.

5.2.1. Experiment I – Varying chunk sizes

For the SAGA Sector-Sphere based wordcount, Sector maintains and tracks data at the file level. To experiment with different chunk sizes, the data files (totalling 4GB) were split manually into smaller chunks before the wordcount application was launched. In this set of experiments, we vary the chunk size from 16 MB to 256 MB, while keeping the number of SPEs constant at 8, and the total data size constant at 4 GB. Each SPE is running on a separate physical node in the cluster. These results are presented in Fig. 4. Note that both data and computation were distributed for these experiments.

As evident from Fig. 4, a correlation exists between the chunk sizes and performance of Sphere. As the chunk sizes increase, the time-to-solution also increases. In particular, we observe that performance declines for chunk sizes larger than 64 MB. On the Hungarian cluster, the time-to-solution increased by 21 % going from 64 MB chunk size to 128 MB chunk size, and by 56 % going from 128 MB chunk size to the 256 MB chunk size. On the India system of the FutureGrid cluster, the time-to-solution increased by 22 % going from 64MB to 128MB and 25 % going from 128MB to 256MB.

We performed the same set of experiments with SAGA-MapReduce based wordcount and observe a completely different performance trend as the chunk size varied. We use an HDFS file system running data nodes on each of the 8 workers and set the number of reduce tasks to 8. In case of SAGA-MapReduce, performance increases, i.e., time-to-solution decreases, with larger chunk sizes, reaching Sphere's performance at the 256 MB data point.

This can be attributed to the fact that for SAGA-MapReduce, t_{coord} is dominated by the number of chunks (map tasks). The larger the chunk size, the smaller the number of map tasks; and provided the data work-load assigned to each worker is not too high, t_{coord} decreases with decreasing number of map tasks (increasing chunk size).

These experiments have been run on both the Hungarian cluster (HC) and on the India system of FutureGrid (FG), which have rather different hardware characteristics. Fig. 4 shows the discussed trends are independent of the backend, and depend on the programming model used and its implementation. The experiment's standard deviations are usually below 10% of the measured data values, and have been plotted where they exceed the plot point size. For SAGA-MapReduce, the data points for small chunk sizes varied significantly, as shown in the graphs. The latter can be attributed to the fact that a high number of map-tasks cause contention due to the advert service, i.e. they require higher coordination overhead. Additionally, for small chunk sizes (32MB and less), there are many intermediate files produced, which take a lot

of time to process in the reduce phase (issuing reduce tasks takes much longer because more advert accesses are needed). In general, due to the finer granularity of a workload, failures are more common.

5.2.2. Experiment II – Varying Workers

Having understood the influence of the chunk size on performance, the next set of experiments aim to understand how varying the number of workers effects the time-to-solution. These experiments are carried out on both the HC and India. Fig. 5 shows the results for SAGA-Sphere, Fig. 6 for SAGA-MapReduce on FutureGrid. The chunk size was set constant at 64 MB, the data size again 4GB.

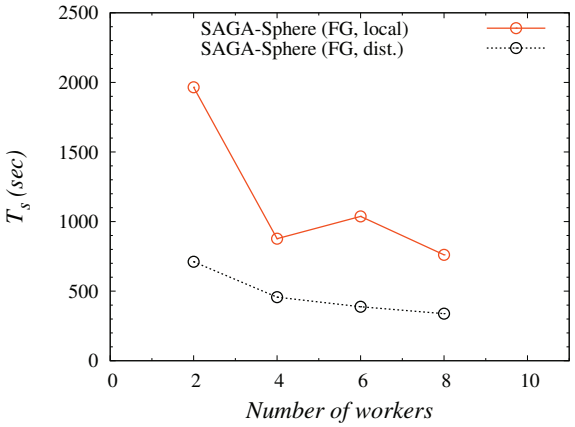

Figure 5: Comparison of SAGA-Sphere performance when varying the number of workers between 2 and 10 in two configurations: (1) local data and computation and (2) distributed data and computation.

For the local configuration, we launch Sector and Sphere on a single physical node. For the distributed configuration, we launch Sector and Sphere on one physical node per SPE. For the dataset sizes considered, we observe good performance over 4 to 6 distributed workers, after which the coordination costs due to the number of SPEs starts to get high, with a concomitant increase in time-to-solution. This is a nice but simple demonstration of the advantage of distribution (logically distributed in this case, if not physically distributed). Sector can maintain file replicas to achieve optimal data distribution between SPEs and minimize synchronization overhead. For the purpose of our experiments, we limited Sector to not create any replicas.

We performed similar measurements via SAGA-MapReduce. We set the number of reduce tasks to

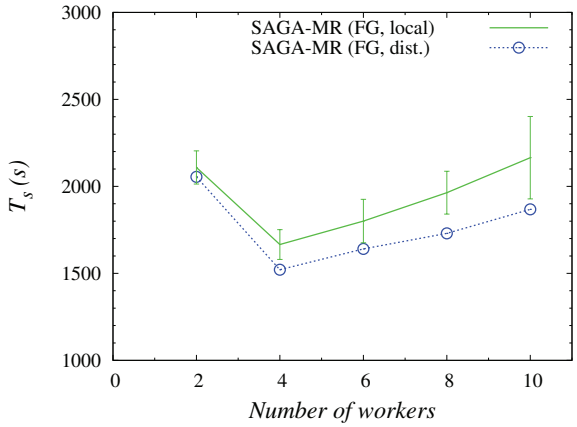

Figure 6: Comparison of SAGA-MapReduce when varying the number of workers between 2 and 10 in two configurations: (1) local data and computation and (2) distributed data and computation.

be equal to the number of workers spawned. For the local configuration we launch workers on the machine running the master as separate processes and use the local file system for file operations. For the distributed configuration we use HDFS as the distributed file system and launch jobs using the SAGA SSH job adaptor. As can be seen in Fig. 6 SAGA-MapReduce does not scale as expected in the distribute configuration. This is most likely due to the increased coordination costs that arise with an increasing number of workers.

Since each map task writes as many files as the number of reduce tasks at the same time, and each reduce task needs to read from as many files as the number of input chunks, the number of concurrent disk I/O increases very quickly; this can cause a bottleneck when performing computations on one physical node (I/O system). According to Fig. 6 the optimal number of workers (for 4GB and a chunk size of 64MB) is 4 for the local configuration.

On FutureGrid, the local configuration performance shows similar trends to the distributed configuration; there is essentially a fixed (consistent) difference in the time-to-solution between the local and distributed configurations. To a first approximation, this can be explained by the fact that any gains in I/O cost by distributing data are compensated for by the overhead of setting up the distributed workers. This is consistent with the fact that as smaller worker counts (and thus lower coor-

dination and set-up overhead), the distributed and local configurations have comparable performance. It is fair to assume that as data-set sizes become larger, the distributed case will gradually perform better.

The standard deviations for these runs have again been below 10% in most cases, and have been plotted where they exceed the plot point size. In general the SAGA-MapReduce and SAGA-Sphere performance for the the HC shows the same trends as on FutureGrid, e.g., "saturation" (i.e. limits of IO and memory) with increasing worker numbers.

Discussion:. As evident from the data plots in Fig. 4, certain behavioral trends for SAGA-Sphere and SAGA-MapReduce emerge. In Experiment 1, where we keep the number of workers constant and vary the chunk sizes, the trends between SAGA-MapReduce and Sphere are reversed: the performance of SAGA-Sphere deteriorates with increasing chunk sizes, while the performance of SAGA-MapReduce improves. This behavior suggests that SAGA-MapReduce's synchronization overhead to manage smaller chunk sizes compared to the speed up achieved through parallelism is much higher. In the case of the wordcount application, SAGA-MapReduce appears to be more suitable for coarse grained computations. SAGA-Sphere, on the other hand, yields better performance from smaller chunks sizes (a larger amount of files) making it suitable for finer grained computations with better data distribution.

We instrumented the Sphere code to analyze it's behavior in more detail. The processing of larger files (i.e. reading the file from disk to memory, computing the hash for the words contained in the file, as well as writing the file into the Sphere output buffer) did not contribute significantly to the time taken for solution. We did observe that once the Sphere framework took over the responsibility of processing the output buffer, it took more time to process larger buffers than smaller ones. Identifying exactly what module is responsible for this bottleneck is out of the scope of this paper.

In Experiment 2, where we keep the chunk size constant at 64 MB, SAGA-Sphere exhibits a trend where adding more SPEs for the distributed configuration has a positive impact on performance. However, at the 8 SPEs and 10 SPEs data points, we see a decline in performance, possibly due to high synchronization costs between the workers. What is interesting to notice are the two data points at 128 MB chunk size and at 16 MB chunk size for SAGA-Sphere in Fig. 4 for the distributed HC and FG environments. Reducing the chunk size results in a greater number of files, and thus provides the opportunity for better data distribution; we notice an almost 30% improvement in performance for HC and 19% for FutureGrid. This further confirms our supposition that good data distribution had a major impact on Sphere's performance for the wordcount application.

5.3. Type III ALI: Concurrent Interoperability

We discuss a third type of interoperability in this section, where SAGA-MapReduce and SAGA-Sphere are used in conjunction to solve the wordcount problem. We first use the 'netperf' utility to measure the throughput from the client host to the SAGA-MapReduce master node and SAGA-Sphere master node. In our case, the throughput measured to the two nodes was approximately equal (935 MB/s to SAGA-Sphere and 925 MB/s to SAGA-MapReduce). Based on these metrics, we split the 4.0 GB data set into two equal 2.0 GB parts. This is to ensure that the data transfer time to both masters is approximately the same. We configured both systems to utilize 4 workers each and 64 MB chunk sizes. The data transfer time to the Sector cloud took a total of 97.8 seconds and 10.4 seconds to the SAGA-MapReduce master node. The longer transfer time to Sector can be credited to the overhead incurred from registering the files in Sector. The Sector upload utility was used to transfer the data.

We found that SAGA-Sphere took a total of 441.3 seconds to process the 2.0 GB data, while SAGA-MapReduce took a total of 769 seconds. Aggregating the output results from the two systems took a negligible amount of time (only 0.9 seconds). The data was already sorted and hence could be merged in almost constant time. The above simple experiment of combining two varied programming models for solving a common problem paves the way to further investigation into smarter data and compute placement techniques. The total time taken to execute the wordcount application in this case was approximately 877.9 seconds. It is interesting to note that this performance measure lies between 1329.96 seconds for 8 SPEs and 716 seconds for 8 SAGA-MapReduce workers at a 64 MB chunk size.

6. Discussion & Conclusions

As alluded to in the opening section, the volume and the degree-of-distribution of data is increasing rapidly; this is consistent with the fact that it is not possible to localize peta-bytes of data, let alone exa-bytes. This imposes a need for applications to work across a range of heterogeneous distributed infrastructures, possibly using several different programming models. Thus, on the one hand there is a need to decouple programming models from infrastructures, and provide a range of programming models at the application developer's disposal. On the other hand, in order to build empirical models, or validate existing predictions of performance, it is important to establish & experiment with programming models and data-oriented algorithms (e.g., streaming) on a range of systems. A critical and necessary step to achieve both is to provide application-level interoperability as discussed.

The fundamental aim and contribution of this paper has been to demonstrate and understand several types of appliation-level interoperability, and the levels at which such interoperability can be provided. Although driven by proof-of-capability experiments and results therein, there are deeper questions that motivate this work and define the research methodology. For example, for a given application, it is not known *a priori* how to engineer production-grade application-level interoperabilty, i.e., at what level and how it should be provided. Through our implementation and analysis of the three levels, we establish that different performance challenges and functional capabilities arise; understanding these "trade-offs" is important in order to reason and thus implement interoperabilty at the "right level". It is worth noting, that the impact of application-level interoperabilty goes beyond just the "practical" ability to utilize multiple production-grade cyberinfrastructure: it is an important step towards understanding general-purpose programming models.

In this paper, we analyze three levels of interoperability. All three levels were implemented using SAGA-MapReduce, which from an execution perspective is a relatively straight-forward application. In general, SAGA gives us the opportunity to experiment with different programming models on different infrastructure and with varying configuration. At the lowest level, SAGA-MapReduce demonstrates how to decouple the development of applications from the deployment details of the run-time environment (Type I ALI). 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. With implementations of the two application frameworks – SAGA based Sector-Sphere and the SAGA-MapReduce implementation, we also demonstrated Type II ALI: applications can seamlessly switch between back-ends by switching frameworks encapsulating different programming models. Finally, by concurrently using Sector-Sphere MapReduce and SAGA-MapReduce, we demonstrated Type III ALI, allowing the application to span a wide variety of back-ends concurrently, and efficiently.

Our approach does not confine us to MapReduce and applications based upon MapReduce; SAGA is also capable of supporting additional programming models, like Dryad. We are also developing applications with non-trivial data-access, transfer and scheduling characteristics & requirements, and deploying them on different underlying infrastructure guided by heuristics to seek optimised performance. This analysis is done through developing performance models of transferring data between frameworks, as well as the distribution of the computing resources in the environment. Based on this analysis, the data is placed efficiently, and a subset of nodes and frameworks maybe chosen to perform the necessary computations. The shuffled data is also cached for future computations. We have embarked on the creation of components that facilitate intelligence and flexibility in data placement relative to the computational resource [9]. These components are connected in frameworks using SAGA, thus furthering the agenda of general-purpose programming models with efficient run-time support that can utilize multiple heterogeneous resources.

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, while ME acknowledges support from the grant OTKA NK 72845. We also acknowledge internal resources of the Center for Computation & Technology (CCT) at LSU and computer resources provided by LONI/TeraGrid for QueenBee. We thank Chris Miceli, Michael Miceli, Katerina Stamou &

Hartmut Kaiser for their collaborative efforts on early parts of this work. We thank Sharath Maddineni for useful discussions and Pradeep Mantha for help with image preparation. We thank Mario Antonioletti and Neil Chue Hong for supporting this work through GSoC-2009 (OMII-UK Mentor Organization). This document was developed with support from the National Science Foundation (NSF) under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] The Earth System Grid, http://www.earthsystemgrid.org/home.htm.

[2] S Jha, A Merzky and G Fox, Clouds Provide Grids With Higher Levels of Abstractions and Support for Explicit Usage Modes, Concurrency and Computation: Practises and Engineering Vol 21, no 8, 1087-1108 (2009).

[3] S Jha, *et al*, Design and Implementation of Network Performance Aware Applications Using SAGA and Cactus, pp 143-150, IEEE Conference on e-Science 2007, Bangalore, ISBN 978-0-7695-3064-2 DOI 10.1109/E-SCIENCE.2007.28.

[4] S Jha *et al*, Developing Adaptive Scientific Applications with Hard to Predict Runtime Resource Requirements, Proceedings of TeraGrid 2008 Conference (Performance Challenge Award).

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

[6] Protocol Buffers. Google's Data Interchange Format. `http://code.google.com/p/protobuf`.

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

[8] Louisiana Optical Network Initiative `http://www.loni.org`.

[9] C Miceli, M Miceli, B. Rodgriguez-Milla and S Jha, Understanding Performance of Distributed Data-Intensive Applications, Phil. Trans. R. Soc. A 13 September 2010 vol. 368 no. 1926 4089-4102 (doi: 10.1098/rsta.2010.0168).

[10] Dhruba Borthaku. The Hadoop Distributed File System: Architecture and Design. Retrieved from `http://hadoop.apache.org/common/`, 2010.

[11] Cloudstore. Cloudstore distributed file system (formerly, Kosmos file system). `http://kosmosfs.sourceforge.net/`.

[12] 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.

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

[14] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):43, 2003.

[15] Y. Gu and R. L. Grossman. Sector and Sphere: the design and implementation of a high-performance data cloud. *Royal Society of London Philosophical Transactions Series A*, 367:2429–2445, May 2009.

[16] H. Kaiser, A. Merzky, S. Hirmer, and G. Allen. The SAGA C++ Reference Implementation. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06) - Library-Centric Software Design (LCSD'06)*, Portland, OR, USA, October, 22-26 2006.

[17] A. Luckow, S. Jha, A. Merzky, B. Schnor, and J. Kim. Reliable Replica Exchange Molecular Dynamics Simulation in the Grid using SAGA CPR and Migol. In *Proceedings of UK e-Science 2008 All Hands Meeting*, Edinburgh, UK, 2008.

[18] C. Miceli, M. Miceli, S. Jha, H. Kaiser, and A. Merzky. Programming abstractions for data intensive computing on clouds and grids. In *Cloud 2009, Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 478–483, May 2009.

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