Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A
MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Understanding performance of distributed data-intensive applications

Christopher Miceli, Michael Miceli, Bety Rodriguez-Milla and Shantenu Jha

Phil. Trans. R. Soc. A 2010 **368**, 4089-4102
doi: 10.1098/rsta.2010.0168

References — **This article cites 1 articles**
http://rsta.royalsocietypublishing.org/content/368/1926/4089.full.html#ref-list-1

Rapid response — Respond to this article
http://rsta.royalsocietypublishing.org/letters/submit/roypta;368/1926/4089

Subject collections — Articles on similar topics can be found in the following collections

e-science (31 articles)

Email alerting service — Receive free email alerts when new articles cite this article - sign up in the box at the top right-hand corner of the article or click **here**

To subscribe to *Phil. Trans. R. Soc. A* go to:
http://rsta.royalsocietypublishing.org/subscriptions

This journal is © 2010 The Royal Society

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

Phil. Trans. R. Soc. A (2010) **368**, 4089–4102
doi:10.1098/rsta.2010.0168

Understanding performance of distributed data-intensive applications

By Christopher Miceli[1], Michael Miceli[1], Bety Rodriguez-Milla[1] and Shantenu Jha[1,2,*]

[1] *Center for Computation & Technology, and* [2] *Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA*

Grids, clouds and cloud-like infrastructures are capable of supporting a broad range of data-intensive applications. There are interesting and unique performance issues that appear as the volume of data and degree of distribution increases. New scalable data-placement and management techniques, as well as novel approaches to determine the relative placement of data and computational workload, are required. We develop and study a genome sequence matching application that is simple to control and deploy, yet serves as a prototype of a data-intensive application. The application uses a SAGA-based implementation of the All-Pairs pattern. This paper aims to understand some of the factors that influence the performance of this application and the interplay of those factors. We also demonstrate how the SAGA approach can enable data-intensive applications to be extensible and interoperable over a range of infrastructure. This capability enables us to compare and contrast two different approaches for executing distributed data-intensive applications—simple application-level data-placement heuristics versus distributed file systems.

Keywords: data-intensive computing; distributed computing; cloud computing; grid computing

1. Introduction

The role of data-intensive computing is increasing in many aspects of science and engineering (Hey 2009) and other disciplines. For example, Google processes around 20 petabytes of data per day (Dean & Ghemawat 2008); trends suggest a continued growth in volumes. In addition to increasing volumes of data, there are several reasons driving the need for computation on distributed data, such as the proliferation of distributed services and the localization of data owing to security and privacy issues. The challenges in developing effective and efficient distributed data-intensive applications that meet a range of design and performance metrics arise from a complex interplay of the challenges in developing distributed applications on the one hand with those in developing data-intensive applications on the other. New programming approaches, cyber-infrastructure and data-management

*Author for correspondence (sjha@cct.lsu.edu).

One contribution of 15 to a Theme Issue 'e-Science: past, present and future II'.

 This journal is © 2010 The Royal Society

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

techniques are required to handle and analyse large data-volumes effectively. In general, at large scales, data placement, scheduling and management need increased attention.

An important design consideration and objective for data-intensive distributed applications and systems is the ability to determine and support an optimal distribution of data and computational workloads. An important challenge is to find broadly applicable strategies to distribute data and computation that can support a range of different mechanisms to achieve this objective. In general, there are many factors that determine the performance of a given application on distributed infrastructure, such as the degree of distribution, granularity of workload decomposition and type of infrastructure to name just a few. In this paper, we investigate two ways to handle the optimal data-computation distribution problem and try to understand their relative performance as a function of these factors.

In the first approach, we encode a simple metric to determine the *workload* placement into a 'heuristic framework'. We contrast this approach with the use of an open-source distributed file system (DFS). A DFS typically controls the *distributed* data placement and provides a uniform interface for accessing files on multiple hosts. But the underlying algorithms, scheduling strategies and implementations vary greatly between different offerings; hence, it is difficult to estimate *a priori* the application-level performance on a given DFS. Thus having the ability to compare and contrast different DFSs for a given application is an important requirement. More generally, the ability to provide repeatable, extensible and verifiable performance tests on different distributed platforms is required in order to understand the interplay and trade-offs between these factors. We hope to provide the first steps towards a comprehensive and a rigorous benchmarking process for distributed data-intensive applications, analogous to the suite of benchmarks for high-performance computing.

In §3, we will establish how the use of adaptors enables SAGA-based applications to make comparisons across platforms effectively. Specifically, an aim of this paper is to present a credible initial benchmarking template, and to suggest ways to answer the question of whether 'to move computational workload to where data resides or to redistribute the data'. As we will show, the actual answer will depend significantly on the specific dataset sizes, algorithm/applications and infrastructure used. Data privacy, security and access policy are crucial issues, but typically are not determinants of performance for distributed applications; thus, we will not consider them in this paper.

In §2, we present a very brief discussion of SAGA, which provides the distributed programming capability to develop the All-Pairs-based application (§3). The work in this paper uses a SAGA-based application that we developed to support genome sequence matching to determine and analyse performance that we developed. SAGA encodes the All-Pairs pattern in a distributed context; we chose All-Pairs as it is a representative and common data-access pattern found in many data-intensive distributed applications.

Section 4 offers an overview of the various parameters used to understand the experimental configurations, discusses the experiments carried out to understand the performance, as well as provides a local analysis of the results. In §5, we review the understanding gained and look ahead.

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

2. SAGA and SAGA-based frameworks for large-scale and distributed computation

The Simple API for Grid Applications (SAGA) provides a simple, POSIX-style application programming interface (API) to the most common distributed functions at a sufficiently high level of abstraction so as to be independent of diverse Grid environments. The SAGA specification defines interfaces for the most common Grid-programming functions grouped as a set of functional packages (figure 1). Some key packages are as follows. (i) File package—provides methods for accessing local and remote file systems, browsing directories, moving, copying and deleting files, setting access permissions, as well as zero-copy reading and writing. (ii) Job package—provides methods for describing, submitting, monitoring and controlling local and remote jobs. Many parts of this package were derived from the largely adopted DRMAA specification. (iii) Stream package—provides methods for authenticated local and remote socket connections with hooks to support authorization and encryption schemes. (iv) Other packages, such as the RPC (remote procedure call) and Replica package.

SAGA provides the basic API to implement distributed functionality required by applications, and is also used to implement higher level APIs, abstractions and frameworks that, in turn, support the development, deployment and execution of distributed applications (El-Khamra & Jha 2009). In the absence of a formal theoretical taxonomy of distributed applications, figure 1 can act as a guide. Using this classification system, there are three types of distributed applications: (i) applications where local functionality is swapped for distributed functionality, or where distributed execution modes are provided, (ii) applications that are naturally decomposable or have multiple components that are then aggregated or coordinated by some unifying or explicit mechanism, and, finally, (iii) applications that are developed using frameworks, where a framework supports specific application characteristics (e.g. hierarchical job submission), and/or recurring patterns (e.g. MapReduce, All-Pairs). SAGA has been used to develop system-level tools and applications of each of these types. In Merzky *et al.* (2009), we discussed how SAGA was used to implement a higher level API to support workflows. In Miceli *et al.* (2009), we discussed how a SAGA-based MapReduce was developed; here, we will discuss a SAGA-based All-Pairs application.

3. All-Pairs: design, development and infrastructure

We use an application based upon an All-Pairs abstraction whose distributed capabilities are developed using SAGA. The All-Pairs pattern was chosen because of its pervasive nature and applicability to many other data-intensive distributed applications. This enables the results to be abstracted to describe and predict different applications in addition to the genome sequence matching with similar structured data-access patterns.

The All-Pairs abstraction applies an operation on two datasets such that every possible pair containing one element from the first set and one element from the second set has some operation applied to it (Moretti *et al.* 2008). Essentially, All-Pairs is a function of two sets, A and B, with number of elements m and n,

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

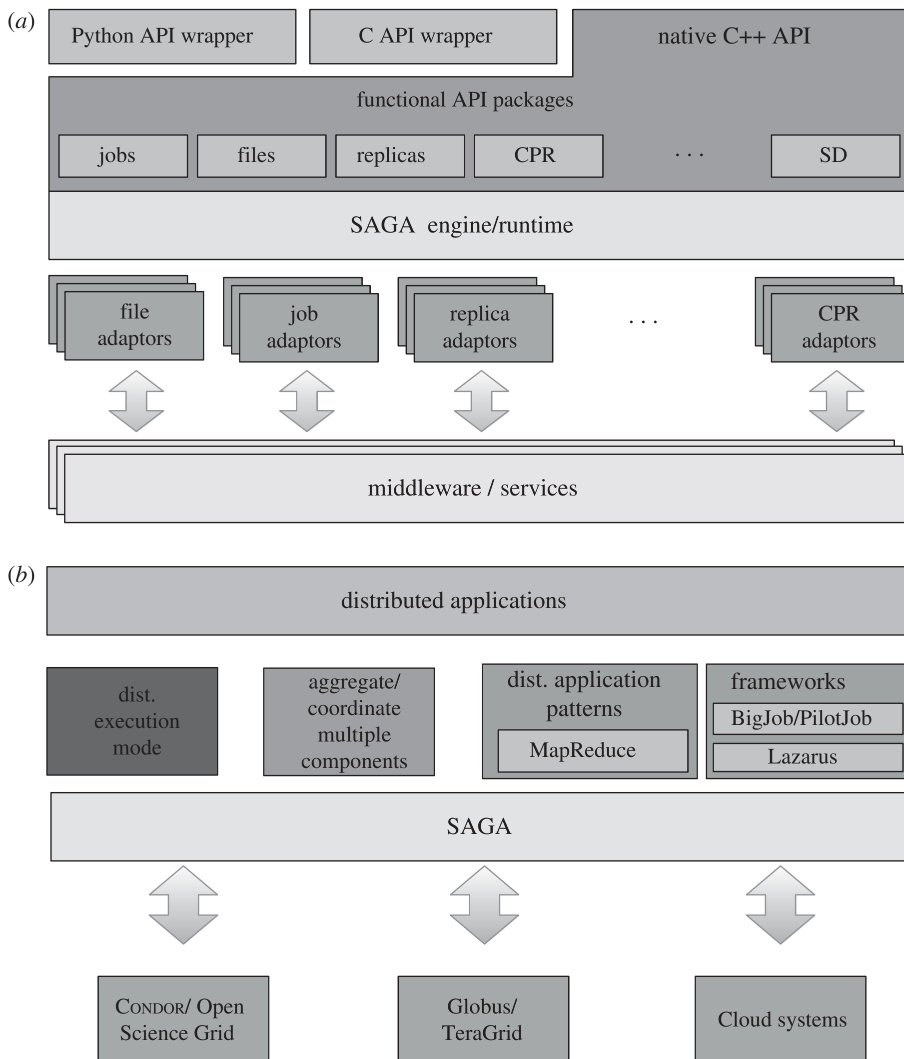

Figure 1. (*a*) A layered schematic of the different components of the SAGA landscape. At the topmost level is the simple integrated API, which provides the basic functionality for distributed computing. The BigJob abstraction is built upon this SAGA layer using Python API bindings. (*b*) The ways in which SAGA can be used to develop distributed applications. The different shaded boxes represent the three different types; frameworks in turn can capture either common patterns or common application requirements/characteristics.

respectively, which creates a matrix P of size $m \times n$. Each element $P_{i,j}$ of the matrix is the result of the operation f applied to the elements A_i and B_j,

$$\text{All-Pairs}(A, B, f) \rightarrow P_{m\times n} \mid P_{i,j} = f(A_i, B_j). \tag{3.1}$$

The result of this application is stored in a matrix similar to figure 2. The application spawns distributed jobs to run sets of these function operations. Examples of problems that fall into this category are image comparison for facial recognition, and genome comparison. The usefulness of All-Pairs comes

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

set A

set B	a1	a2	a3	a4
b1	.28	.38	.74	.12
b2	.40	.60	.83	.90
b3	.75	.28	.61	.77
b4	.16	.92	.98	.24

Figure 2. An example result from an All-Pairs enabled application. Each matrix element describes the similarity between the corresponding sets. (Larger values indicate greater similarity.)

from the ability to easily change the function without whole-scale refactoring. We use a SAGA-based All-Pairs framework and simply implemented the comparison function. The comparison function compares genomes to find the best matching gene in a genome. The function finds the number of similarities among the genes and returns the percentage of the genes that are identical. Despite having a relatively small output $O(\text{KB})$, the genome comparison application can be classified as having a large data throughput as it has large input $O(\text{GB})$ with many data reads.

The problem becomes a two-level assignment problem: first, which pairs to put into an assignment set, and, second, which distributed resource to use to compute on that set. In a distributed context, it is important to consider time to transfer a dataset to an assigned resource. Simple heuristics based upon transfer times and network performance can in conjunction with a knowledge of the affinity of data (set) to a specific resource be used for an efficient assignment.

(*a*) *Infrastructure used*

In the experiments we use the stable, open-source DFS CloudStore (formerly KFS), which is written in C++ and released under the Apache License Version 2.0 (CloudStore 2009). It is inspired by the highly successful Google Filesystem (GFS), which is closed source and unavailable for research (Ghemawat *et al.* 2003). CloudStore was chosen for its high performance focus, C++ implementation, and its source code availability. It also provides a means to automatically replicate data on different hosts to provide efficient data access and fault tolerance. In general, DFSs are useful and effective tools to consider for data-intensive scientific applications, with multiple open-source, reliable file systems now available. DFSs are becoming increasingly common as part of Cloud infrastructures and available machine images. The most common parameters in determining the performance of using a DFS are the performance overhead compared with a normal local file system, the number of replicas of each datum/file and the number of servers. While a DFS removes the responsibility of replica management and data server placement, the abstraction often increases the difficulty in determining where in the DFS the data are being stored.

We use a heuristic-based method, which differs from a DFS in that it determines where data are located and, based upon application-level information, decides where the work should be placed. Determining data location can be as

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

simple as looking at the IP address of the worker and finding where it is located, or as complicated as using network analysis tools to determine the optimal data transfer minimization time. For file transfer during these heuristic framework-based experiments, we use GridFTP—a tool that can support high-performance transfers (GridFTP 2009).

For the experiments, to use both CloudStore and GridFTP within one application, we wrote adaptors for SAGA that implement the file system package. SAGA allows the application to handle seamlessly the DFS and GridFTP-based data stores on clouds and grids, enabling us to compare both. For accurate comparisons, we must consider the overhead introduced by SAGA. We have measured a slight difference in times when using SAGA compared with native applications; however, the adaptor implementations grew at the same rate as GridFTP and CloudStore usage without SAGA.

4. Experiments and performance analysis

We designed three types of experiments in order to understand the interplay of different determinants of performance and make a meaningful comparison of CloudStore's behaviour to heuristic-based data placement and file management. To help understand the issues at play, we define the time to completion t_c as

$$t_c = t_x + t_{I/O} + t_{compute}, \tag{4.1}$$

where t_x is the pre-processing time, the dominant component of which is the time for transfer, $t_{I/O}$ is the time it takes to read and write files, and $t_{compute}$ is the time it takes the comparison function to run. We focus on three variables that influence the value of t_c: degree of distribution, data dependency and workload. The degree of distribution (D_d) is defined as the number of resources that are used for a given computation/problem. For example, if data are distributed over three machines, $D_d = 3$; if data are distributed over three machines but the computational tasks are distributed over four machines, $D_d = 4$. We try to answer questions such as: How does t_c for KFS compare with a heuristic-based approach as dataset sizes vary and the degree of distribution changes?

(a) *Experimental configuration*

As explained before, for the experiments, we use an All-Pairs implementation that uses SAGA. An XML configuration file defines various initial parameters of the All-Pairs implementation. The configuration file defines the location of data that constitute the two input sets, the grouping of pairs from these sets to be provided to the compute resources, and the available machines that will perform the operation on these sets of pairs. The application takes these groups of pairs and maps them to a computational resource dynamically at run-time.

Furthermore, variables external to the All-Pairs implementation also influence experimental results. The experiments can be completely described by a system configuration (S_i) that is captured by tuple of the form

$$(c_s, N_c, M_c, fs, m, r), \tag{4.2}$$

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

Table 1. The machine configurations M_c (tuple 4.2) that we used in the experiments, for one, two and three machines. Both c and d can have yes/no (Y/N) values. A $c = Y$ means the machine X_i does computation, and a $d = Y$ means the machine has data stored. For *C4* and *C5*, we divide the data equally among the machines.

configurations	$X(c, d)$; c = compute, d = data storage	description
C1	$X_1(Y, Y)$	X_1 computes and stores data
C2	$X_1(Y, N), X_2(N, Y)$	X_1 computes, X_2 stores data
C3	$X_1(Y, Y), X_2(N, Y)$	X_1 computes, X_1 and X_2 store data
C4	$X_1(Y, Y), X_2(Y, Y)$	X_1 and X_2 compute and store data
C5	$X_1(Y, Y), X_2(Y, Y), X_3(Y, Y)$	X_1, X_2 and X_3 compute and store data

where c_s is the total amount of data in each file of a set (i.e. c_s = chunk size); N_c is the number of workload assignments generated; M_c represents machine configurations *C1, C2, C3, C4* or *C5*; *fs* is the type of file system used; m is the method used to access that filesystem; and r is the degree of replication used in the experiment (with a default value of 1). However, for CloudStore, we investigate performance with $r = 1$, 2 and 3.

Furthermore, each machine configuration (M_c) is a comma-separated list of the machine configurations of the following form: $X(c, d)$, where X is a shorthand reference for the computational resource. c shows if the computational resource X was used in the computational workloads/calculations, and d if the computational resource X assisted in data storage; both have a yes/no (Y/N) value (see table 1).

In the experiments, we have three (fs, m) configurations and five X configurations. The (fs, m) configurations are (local, local), (local, GRIDFTP) and (CloudStore, direct). By direct we mean CloudStore controls the data access. The X configurations are enumerated in table 1. For one machine, $C1 = X_1(Y, Y)$, where resource X_1 is used for both data and the computing workloads; for two machines, we have three configurations; and for three machines we only work with only one configuration. N_c is a very important configuration parameter as it determines the granularity of work, which in turn determines the ability for distribution. If N_c is too small, there may exist idle resources unable to assist the workload while those that are participating could be overloaded.

A sample description of an experiment will now be explained: (287 MB, $N_c = 8$, *C2*, CloudStore, $r = 1$) shows that each element of a set is 287 MB in size (i.e. $c_s = 287$ MB); we have eight assignments; the computational resource X_1 supports computational workloads, but does not have data stored, while computational resource X_2 does not take part in calculations, but stores the data; the file system used is CloudStore, therefore it directly accesses the files, and we have a replication factor of 1 for the data. The machines X_i we use for the experiments are part of LONI (Louisiana Optical Network Initiative). In the experiments, we find t_c as we vary the number of workers N_w, keeping $N_c = 8$, unless otherwise specified. As described above, the All-Pairs implementation used for the experiments has a fixed distribution of data, fixed available computational resources and fixed sets of pairs to operate with. It is also notable that the experiments were reproducible and consistent over a given time of a few hours, but could vary if run more than a few hours apart. This variance is attributable to the load-factor of the computing

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

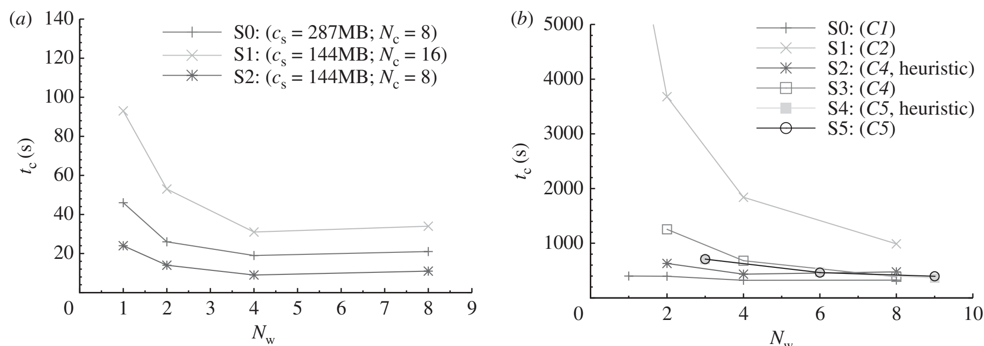

Figure 3. GRIDFTP versus local run. We plot the time to completion t_c versus the number of workers N_w. Note the scale. In general for similar configurations, the local case took less t_c than the GRIDFTP case. In (*a*), we note that on doubling the dataset size t_c also doubled (S2 versus S0). As a result of having more files, we created an overhead in S1 (versus S2) by increasing t_x. In (*b*), the two-machine configurations took more time than using a single machine. As expected, computing on one machine, while having the data stored on another ($t_c = 7360$ s for $N_w = 1$ in S1—not shown), took longer than having some data stored on the resource performing computations (S3). In general, t_c decreased as N_w increased, for up to eight workers (with the possible exception of a single machine where we reached an I/O bound). Here, we also compare the GRIDFTP and heuristic approaches for two and three machines (§4.3). For the two-machine case, we observed a performance improvement by using a data-placement heuristic approach (S2 versus S3). However, with more resources involved, the heuristic did not improve t_c (S4 versus S5). (*a*) One machine (*C1*), local case; (*b*) $c_s = 287$ MB, GRIDFTP case.

environment (LONI) experienced over the course of the experiments. However, it is beyond the scope of this paper to quantify the variance with the load of compute systems.

(*b*) *Experiment I: baseline performance*

In the first experiment, we assign a null computational workload applied on the pairs, giving us $t_{\text{compute}} = 0$. We evaluate data dependencies without the added variable of computation. We use this to examine the I/O, transfer and coordination costs. We run the SAGA-based All-Pairs application on one, two and three unique machines on the LONI grid of Linux Clusters, without any specific data-placement strategy; also, no replication or fault tolerance is invoked. The application sequentially assigns sets of pairs to the first available computational resource. All data are accessed via the GRIDFTP protocol. An important fact to notice is the essentially random mapping of datasets to computational resources based on availability. This is to mimic a naive data-intensive application.

In figure 3, we show the results for t_c, for data accessed locally, and data accessed via GRIDFTP. Using the All-Pairs framework accessing the data using the GRIDFTP protocol had an overhead which can be noted by looking at the y-axes of both graphs. In figure 3*a*, the local cases, we see that working with a smaller dataset ($c_s \sim 144$ MB, $N_c = 8$, for 1.15 GB total) took about half the time that working with a dataset double the size took ($c_s = 287$ MB, $N_c = 8$, for 2.3 GB total). We also see that when working with the same dataset size (2.3 GB),

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

but partitioned differently, t_c can differ; for example, in S0 ($c_s = 287$ MB, $N_c = 8$) versus S1 ($c_s = 144$ MB, $N_c = 16$), t_c increased for S1, owing to added transfer time by doubling the number of files, although we decreased the file size by half.

In figure 3*b*, in which we used GridFTP to access the files, we see that a single machine took less time than the configurations that involved two machines. Also, having to access data remotely was a disadvantage, $t_c(\mathrm{S0}) > t_c(\mathrm{S1})$. For the one-machine configuration, t_c was approximately constant, probably caused by an I/O bound. For all the cases, it is expected that t_c decreases with increasing number of workers; however, after a critical value of N_w (defined as N_w^c), t_c will increase because it will take more time to coordinate the workers.

(*c*) *Experiment II: heuristic-based data placement*

The second experiment is similar to the first, except that the application is aware of the location of data before determining where to assign a set of data-dependent computation to an idle worker. Inspired by earlier work (Jha *et al.* 2007), this version of the application performs an extra step during application start-up that approximates the performance of the network by pinging the hosts that may be either a computational resource or a data store. This information is then assembled into a graph data structure. This graph is used at run-time when the application needs to map an idle worker to an unprocessed workload (set of pairs) defined in the XML file. This changes the first available computational resource assignment mechanism described in the first experiment to a heuristic-based approach. Although ping is not very sophisticated in terms of describing a network's behaviour, it is a first approximation to a performance aware data-placement strategy. We also experimented with the Netperf (Netperf 2009) to capture the network's behaviour. Even though Netperf has the advantage of being able to determine throughput and bandwidth over multiple protocols, it did not yield different results. We attribute this to the fact that we were working with a static set of resources (LONI), the same data graph was generated as with ping. The Netperf-based heuristic system added approximately 8 s per resource of overhead; hence, we used a ping-based approach.

These approaches know where the files are located and their distance to available computational resources, thus allowing more intelligent decisions when mapping a set of pairs to a computational resource. An even more involved approach would be to manage locations of files dynamically at run-time depending on usage patterns. We leave this approach to future research.

The overhead of heuristics includes the time spent pinging hosts and building the graph data structure. The total time spent for this overhead was negligible at approximately 2 s per application run. In figure 3*b*, we see that we achieved a reduction in time to completion owing to the use of heuristics. However, when the same tests were performed using three resources instead of two, the heuristic seemed to offer no significant reduction. The explanation that we propose for this is that all datasets comprised non-collocated files. Because of this, the data dependencies for any given dataset were similar to all other datasets. Each set of pairs would take the same time to transfer using any computational resource.

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

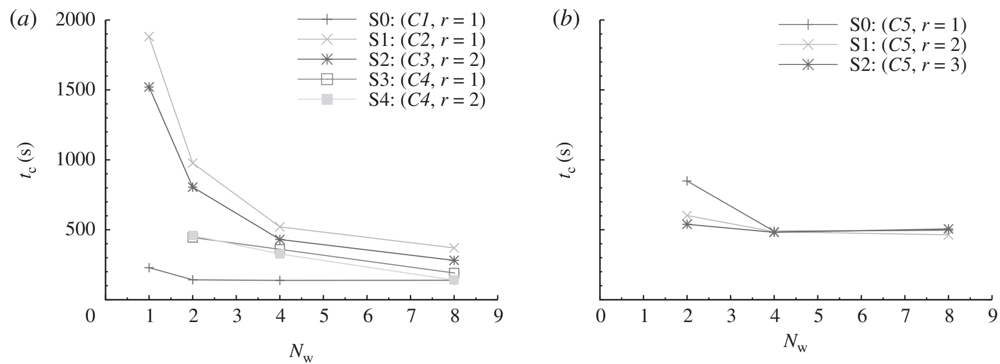

Figure 4. (287 MB, CloudStore). (a) All-Pairs' performance with CloudStore locally and on two different machines. (b) A demonstration of how this scales to three machines, for degree of replication $r = 1, 2$ and 3. CloudStore performed better than the local and heuristic-based approaches (see figure 3). Again, having data in the resource with the workload decreased t_c. When data were spread across all the computational resources, having a degree of replication $r = 1, 2$ or 3 did not significantly decrease t_c, except to the case of three machines and two workers. (a) One and two machines; (b) three machines.

(d) *Experiment III: CloudStore*

The third experiment provides information on CloudStore's performance in handling data locality. The same All-Pairs application as in experiments I and II is used, except all data are stored on the DFS CloudStore under various configurations. Some variables of importance include number of data servers that store data, replication value for data in these data servers, and, as above, placement and number of computational resources. All read and writes also use the DFS. Again, for the first set of results, we do not add the comparison function, giving us $t_{\text{compute}} = 0$.

As with the first experiment, we attempted to capture how compute time scales for different configurations (figure 4). Accessing data remotely adversely affected the performance. We can see that computing in one resource while the data were on another (S1) took the greatest amount of time. Having data in the resource that was computing helped performance (S2, S3 and S4). The number of machines to which workload was assigned was also important. Placing workload on two machines also decreased t_c (S2 versus S3). We varied the degree of replication for the *C4* and *C5* configurations, i.e. for the cases of two and three machines, where all the resources had workload assigned and data stored. With a replication degree larger than 1, data were almost certainly collocated with the computational resource. For *C4*, having $r = 2$ improved t_c, but not considerably. For *C5*, different degrees of replication only made a difference in the case of two workers.

We then added the actual genome comparison function and we compared it with the base case where we did not include the function. We define Δt_c, which is the time difference between these two cases. In figure 5, we see that the time taken to do the genome comparison was relatively small compared with the set-up time, transfer and I/O time added together. The fact that Δt_c for a given N_w was not the same for most configurations shows that there was still an overhead; t_x and $t_{I/O}$ were not the same at the times we ran the All-Pairs framework for both cases.

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

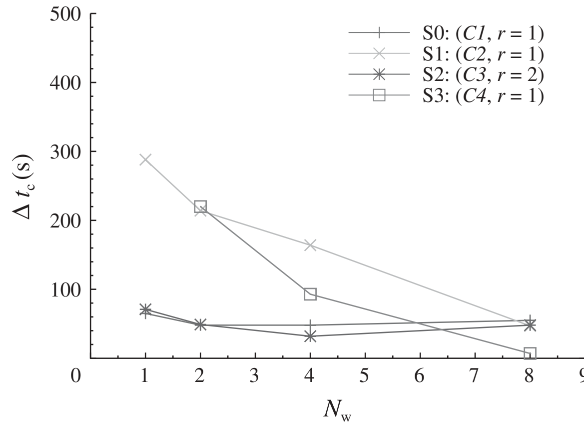

Figure 5. (144 MB, CloudStore). Comparison of CloudStore using the All-Pairs application with and without actual computation. Δt_c is defined as the time it takes to run to completion, where we include the genome comparison function, minus the time it takes to run it when we do not include the comparison function. For the case of $c_s = 144$ MB, the genome function took about two orders of magnitude less than t_x and $t_{I/O}$ combined. Δt_c at a given N_w differed for most of the configurations, showing an overhead, probably caused by different network conditions at the times the runs were performed.

We also compared the results with no comparison function ($t_{compute} = 0$) for two different dataset sizes, one with $c_s = 287$ MB and the other one with half the size, $c_s = 144$ MB (rounded value). Both used CloudStore, and had eight assignments. We defined two quantities, $\Delta t_c^d = t_c(287\,\text{MB}) - t_c(144\,\text{MB})$ and $t_{OH} = 2 \times t_c(144\,\text{MB}) - t_c(287\,\text{MB})$. Figure 6 shows that there are multiple factors that can alter t_c. Some of the factors are network conditions, I/O time, as well as transferring time that can be size dependent, disk seek time, etc. In figure 6*a*, we see that the difference did not scale linearly with the number of workers. It is worth noticing that Δt_c was almost zero (and negative) for eight workers when all the resources had workload and data stored (S3); that is, it took about 10 s less for a 2.3 GB set than for a 1.15 GB set. Figure 6*b* shows that, for most of the cases, there was an overhead which decreased with the number of workers. It also shows that the remote data configuration was the one with the most overhead. Moreover, the S3 case seemed to use a 'non-scalable' infrastructure, as the overhead increased with eight workers.

In figure 7*a*, we compared the lowest times for both the heuristic approach and CloudStore as a function of the number of resources N_r. CloudStore performed better for one and two machines, but not three resources, where the performance of CloudStore decreased. Using the heuristic approach, the lowest times were about the same as N_r was increased from one up to three. In figure 7*b* we plot the time to completion as a function of the number of workers when three resources are used. All the resources had workload assigned and data stored. CloudStore's performance did not vary significantly with the number of workers, while the heuristic approach performed better as we increased the number of resources from one to three.

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

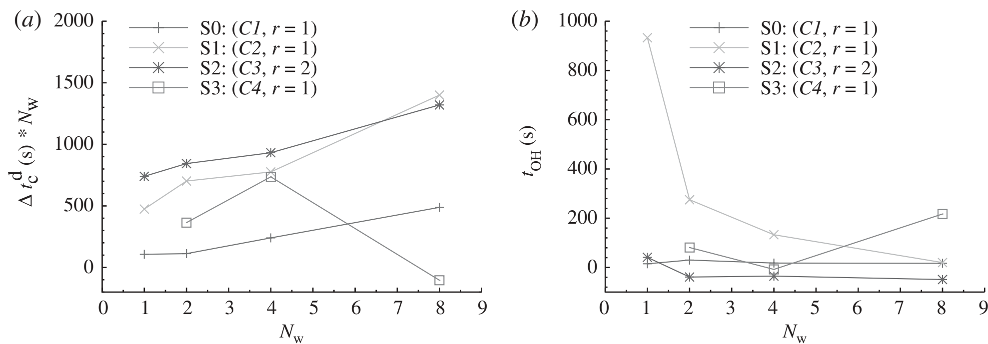

Figure 6. (CloudStore). Here we define two quantities, Δt_c^d, and t_{OH}. Δt_c^d is the difference between t_c found for dataset sizes 1.15 and 2.3 GB, that is, for chunk sizes $c_s = 144$ and 287 MB. t_{OH} is the overhead time of working with chunks of 287 MB in size versus 144 MB chunks twice. (*a*) We see that the difference Δt_c^d decreased with increasing number of workers, but did not scale inversely proportional with N_w; this would be indicated by horizontal lines. (*b*) We see that S1 (remote data) was the one with the most overhead. (*a*) $\Delta t_c^d (= t_c(287\,\text{MB}) - t_c(144\,\text{MB})) \times N_w$; (*b*) $t_{OH} = 2 \times t_c(144\,\text{MB}) - t_c(287\,\text{MB})$.

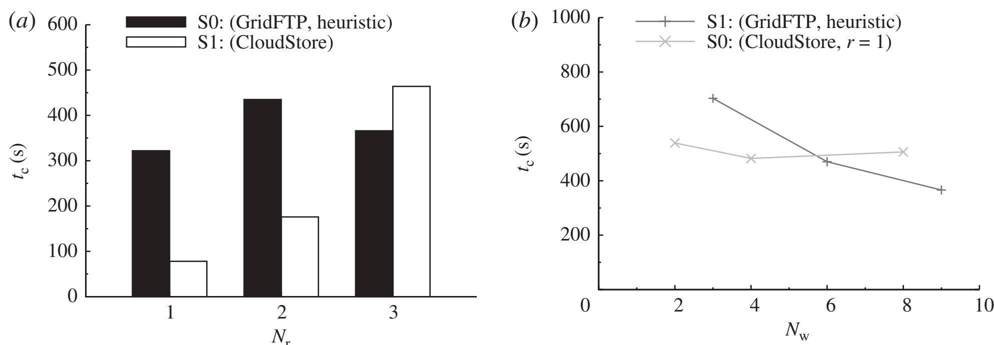

Figure 7. (287 MB). (*a*) The lowest times obtained for a given number of resources (N_r). The lowest times did not vary much for the heuristic-based method, while the performance of CloudStore decreased with increasing number of resources (up to three). (*b*) Performance with three resources for both GridFTP and CloudStore. The three resources computed and had data stored. Performance using CloudStore remained about constant for two, four and eight workers, while the performance of the heuristic approach improved with the number of workers. (*a*) Lowest times as a function of N_r; (*b*) GridFTP versus CloudStore with $N_r = 3$.

5. Conclusions and future work

We set out to understand the factors that influence the performance of a representative data-intensive application, and to understand their interplay. We also aimed to demonstrate how the SAGA approach enables data-intensive applications to be extensible and interoperable over a range of infrastructure. It should be noted that using SAGA to access CloudStore-based and GridFTP-based files introduced a small but negligible overhead; in contrast, the capability

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

enabled us to compare and contrast two different approaches for executing distributed data-intensive applications—simple application-level data placement heuristics versus distributed file systems.

For the volume of data we worked with in the first set of experiments, the local configuration (*C1*) took the least time to completion t_c as a function of the number of workers N_w. However, this will not necessarily continue to be the case as the volume of data increases (which we anticipate to be tens of GB of data). Configuration (*C2*) showed the highest time to completion, while t_c decreased for the mixed configuration (*C3*). Using configuration *C4* decreased the time even further, but not to the point of *C1*. t_c appears to be bounded by t_c(local), where we exhausted the I/O bandwidth. In general, t_c decreased as we added more workers, but it is expected that, after a critical value of N_w (N_w^c), t_c will increase owing to overhead of coordinating workers.

The second experiment implements a simple heuristic for efficient data and work resource assignments. This staging phase only required performing pings, not data transfer trials or reliability tests. The staging phase is worth the time required to build a network graph, as it improved upon the results of the naive baseline performance experiment. The experiments scaled to three distinct resources not because of any fundamental scalability limitation in the approach, but because of the inability to find more than three similar resources.

Overall, the use of CloudStore lowers t_c compared with the heuristic and the GridFTP approaches. The simple heuristic approach did not perform as well as CloudStore for one and two machines, but performed better than a naive use of GridFTP For three machines, the relative performance of CloudStore and the heuristic approach depended upon the N_w. The results for three resources showed that a heuristics-based use of GridFTP continued to experience improvements in performance as the number of resources increased, while CloudStore levelled off.

We will extend this work not only to understand performance over a wider range of infrastructure (DFS, distributed coordination and sharing infrastructure such as BigTable, etc.) and different data-access patterns, but also to explore correlations in data/file access. Such correlation in data access has been observed elsewhere (Doraimani & Iamnitchi 2008); devising specific abstractions to support such correlated access and 'aggregation of files' could enhance performance for a broad range of data-intensive applications and would be both interesting and rewarding.

Important funding for SAGA has been provided by the UK EPSRC grant number GR/D0766171/1 (via OMII-UK), HPCOPS NSF-OCI 0710874, and CyberTools NSF/LEQSF(2007-10)-CyberRII-01. S.J. acknowledges the e-Science Institute, Edinburgh, for supporting the research theme, 'Distributed Programming Abstractions' and theme members for shaping many important ideas. B.R.M. is supported by LONI Institute grant no. LEQSF(2007-12)-ENH-PKSFI-PRS-01. This work has also been made possible thanks to the internal resources of the Center for Computation & Technology at Louisiana State University and computer resources provided by LONI.

References

CloudStore 2009 See http://kosmosfs.sourceforge.net.

Dean, J. & Ghemawat, S. 2008 MapReduce: simplified data processing on large clusters. *Commun. ACM* **51**, 107–113.

Downloaded from rsta.royalsocietypublishing.org on August 11, 2010

Doraimani, S. & Iamnitchi, A. 2008 File grouping for scientific data management: lessons from experimenting with real traces. In *HPDC'08: Proc. 17th Int. Symp. on High Performance Distributed Computing, Boston, MA, 23–27 June 2008*, pp. 153–164. New York, NY: ACM.

El-Khamra, Y. & Jha, S. 2009 Developing autonomic distributed scientific applications: a case study from history matching using ensemblekalman-filters. In *GMAC'09: Proc. 6th Int. Conf. Industry Session on Grids Meets Autonomic Computing, Barcelona, Spain, 15 June 2009*, pp. 19–28. New York, NY: ACM.

Ghemawat, S., Gobioff, H. & Leung, S.-T. 2003 The google file system. In *Proc. 19th ACM Symp. on Operating Systems Principles, SOSP'03, Boston Landing, NY, 19–22 October 2003*, pp. 29–43. New York, NY: ACM. See http://doi.acm.org/10.1145/945445.945450.

GridFTP 2009 See http://dev.globus.org/wiki/GridFTP.

Hey, T. 2009 *The fourth paradigm: data-intensive scientific discovery.* Mountain View, CA: Microsoft Research.

Jha, S., Kaiser, H., Merzky, A. & Weidner, O. 2007 Grid interoperability at the application level using saga. In *E-SCIENCE'07: Proc. 3rd IEEE Int. Conf. on e-Science and Grid Computing, Bangalore, India, 10–13 December 2007*, pp. 584–591. Washington, DC: IEEE Computer Society. See http://saga.cct. lsu.edu/publications/saga_gin.pdf.

Merzky, A., Stamou, K., Jha, S. & Katz, D. S. 2009 A fresh perspective on developing and executing DAG-based distributed applications: a case-study of SAGA-based montage. In *Proc. 5th IEEE International Conference on e-Science, Oxford, UK, 9–11 December 2009*, pp. 231–238. Los Alamitos, CA: IEEE Computer Society.

Miceli, C., Miceli, M., Jha, S., Kaiser, H. & Merzky, A. 2009 Programming abstractions for data intensive computing on clouds and grids. In *CCGRID '09: Proc. 2009 9th IEEE/ACM Int. Symp. on Cluster Computing and the Grid, Shanghai, China, 18–21 May 2009*, pp. 478–483. Washington, DC: IEEE Computer Society.

Moretti, C., Bulosan, J., Thain, D. & Flynn, P. 2008 All-Pairs: an abstraction for data-intensive cloud computing. In *Proc. Int. Parallel and Distributed Processing Symp., Miami, FL, 14–18 April 2008*, pp. 1–11. Piscataway, NJ: IEEE.

Netperf 2009 See http://www.netperf.org/netperf/.