Understanding Performance Implications of Distributed Data for Data-Intensive Applications

BY CHRISTOPHER MICELI[1], MICHAEL MICELI[1], BETY RODRIGUEZ-MILLA[1], SHANTENU JHA[1,2,*]

[1]*Center for Computation & Technology, Louisiana State University, USA*

[2]*Department of Computer Science, Louisiana State University, USA*

*CORRESPONDING AUTHOR SJHA@CCT.LSU.EDU

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 an All-Pairs based genome sequence matching application as a representative data-intensive application. This paper aims to understand the factors that influence the performance of this application, and to understand their interplay. 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 – simple application-level data placement heuristics versus distributed file systems – for executing distributed data-intensive applications.

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 and Ghemawat 2008), with trends showing continuing growth. 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 localisation of data due to security & privacy issues. The challenges in developing effective and efficient distributed data-intensive applications are a complex interplay of the challenges in developing distributed applications on the one hand, with those in developing data-intensive applications to meet a range of design & performance metrics in the other. New algorithmic, infrastructure, and data-management techniques are required to handle large data-volumes effectively. In general at such large scales data-placement, data-scheduling, as well as management need increased attention.

An important design consideration and objective for data-intensive distributed applications and systems is the ability to find and support optimal distribution of data and computational workload. Thus 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 degrees-of-freedom that determine the performance of a given application on distributed infrastructure. Thus a rigorous benchmarking process, which provides repeatable, extensible, and verifiable performance tests

on different distributed platforms is required in order to understand the interplay and trade-offs between these variables. This is analogous to the situation of developing and testing benchmarks for high performance computing (HPC).

The work in this paper is performed in the context of a SAGA-based application 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. In this paper, we investigate two ways to handle the optimal data-computation distribution problem. In the first approach, we encode a simple heuristic to determine the *workload* placement into an "intelligent framework". We contrast this approach with the use of an open-source distributed filesystem (DFS) – which natively supports effective *data* placement. A DFS controls the data placement and provides a uniform interface for accessing files on multiple hosts. DFS are becoming increasing common as part of Cloud infrastructures and available machine images. But the underlying algorithms, scheduling strategies, and implementations vary greatly between different infrastructure, 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. In Section 3 we will establish how the use of adaptors enables SAGA-based applications to make this comparison effectively. 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 redistributed the data". As we will show, the actual answer will depend significantly on the specific data-set 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 Section 2, we provide a very brief discussion of SAGA which provides the distributed programming capability to develop the All-Pairs based application (Section 3). Section 4 provides an overview of the various parameters used to understand the experimental configurations, discusses the experiments carried out to understand the performance, as well as providing a local analysis of the experimental results. Section 5 provide an overview of our the understanding gained and a look ahead.

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

The Simple API for Grid Applications (SAGA) is an API that provides a simple, POSIX-style API to the most common distributed functions at a sufficiently high-level of abstraction so as to be independent of the diverse and dynamic Grid environments. The SAGA specification defines interfaces for the most common Grid-programming functions grouped as a set of functional packages (Fig. 1). Some key packages are: (i) File package - provides methods for accessing local and remote filesystems, browsing directories, moving, copying, and deleting files, setting access permissions, as well as zero-copy reading and writing. (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.

In the absence of a formal theoretical taxonomy of distributed applications, Fig. 1 can

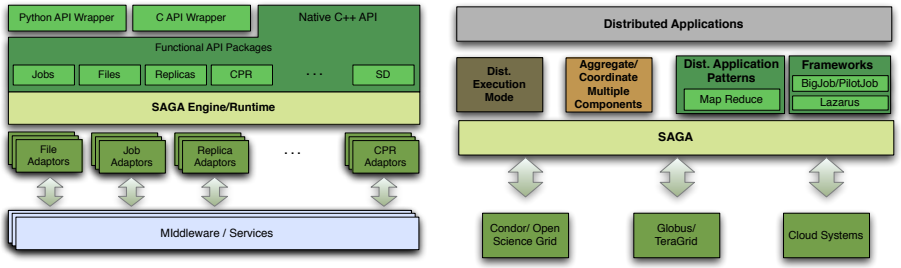


Figure 1: [Left] Layered schematic of the different components of the SAGA landscape. At the topmost level is the simple integrated API which provides the basic functionality for distributed computing. Our BigJob abstraction is built upon this SAGA layer using Python API bindings. [Right] Showing the ways in which SAGA can be used to develop distributed applications. The different shaded box represent the three different types; frameworks in turn can capture either common patterns or common application requirements/characteristics.

act a guide. Using this classification system, there are three types of distributed applications: (i) Applications where local functionality is swapped for distributed functionality, or where distributed execution modes are provided. (ii) Applications that are naturally decomposable or have multiple components are then aggregated or coordinated by some unifying or explicit mechanism. Finally, (iii) applications that are developed using frameworks, where a framework is a generic name for a development tool that supports specific application characteristics (e.g., hierarchical job submission), and/or recurring patterns (e.g., MapReduce, All-Pairs). SAGA provides the basic API to implement distributed functionality required by applications (typically used directly by the first category of applications), and is also used to implement higher-level APIs, abstractions, and frameworks that, in turn, support the development, deployment, and execution of distributed applications (El-Khamra and Jha 2009). 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 MapReduce could be developed.

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 our 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 data-sets 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, respectively, which creates a matrix M. Each element $M_{i,j}$ is the result of the operation f applied to the elements A_i and B_j.

$$AllPairs(A, B, f) \rightarrow M_{m \times n}, \tag{3.1}$$

$$\text{where} \quad M_{i,j} = f(A_i, B_j) \tag{3.2}$$

The result of this application is stored in a matrix similar to Fig. 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 from the ability to easily change the function. We use a SAGA-based All-Pairs framework and simply implemented the comparison function. Our comparison function compares genome 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(KB), our genome comparison application can be classified as having a large data throughput as it has large input O(GB) with many data reads.

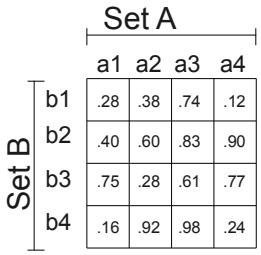


Figure 2: An example result from an All-Pairs enabled application. Each matrix element describes the similarity between the corresponding sets. (Larger values indicate more similarity.)

The problem becomes determining which pairs to put into an assignment set and with which distributed resource to run that set. If transferring data to the job takes too long, we spend more time on data transfer than computation. There may be a resource capable of the work that may be slower than others, but able to be accessed in a relatively quick manner via the network, correcting for this lack of computational ability. An intelligent application attempts to predict and determine a specific data set's affinity to a specific network resource.

Infrastructure Used: In our experiments, to find the performance of a DFS, we use the stable, open-source DFS CloudStore (formerly KFS), which is written in C++ and released under the Apache License Version 2.0. It is inspired by the highly successful Google Filesystem, GFS (CloudStore 2009), which is closed source and unavailable for research. 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, DFS are useful and effective tools to consider for data-intensive scientific applications which have come of age, with multiple open-source, reliant file-systems now available. The most common parameters in determining the performance of using a DFS are the performance overhead compared to a normal local filesystem, 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 is being stored.

Our intelligent framework method differs from a DFS in that it determines where data is and where the work should go. Determining data location can be as simple as looking at the

IP address of the worker and seeing geographically where it is located, or as complicated as using network analyses tools to determine the optimal data transfer minimization time. For the intelligent method, we use gridFTP – a tool that can support high-performance transfer.

To test CloudStore and gridFTP, we wrote adaptors for SAGA that implement the filesystem package. This enables us to compare CloudStore with gridFTP, while still using the same application. Doing this with SAGA is really simple. For accurate comparisons, we must consider the overhead introduced by SAGA. We have measured a slight difference in times; however, the adaptor implementations grew at the same rate as gridFTP and CloudStore usage without SAGA. SAGA allows our application to handle seamlessly the DFS and gridFTP based data stores on clouds and grids. This allows the same exact application to be used for all of our experiments.

4. Experiments and Performance Analysis

We developed three types of experiments in order to understand the interplay of different determinants of performance and make a meaningful comparision of CloudStore's behaviour to manual data placement and file management. We try to answer questions such as: Is a DFS slower than manual placement of data? When manually handling data, what are the advantages of being able to move work to data or data the work? For this, we define the time to completion t_c:

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

where t_x is the pre-processing time, whose the dominant component 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 our comparison function to run. We focus on three variables to measure t_c: degree of distribution, data dependency, and workload. The degree of distribution (D_d) is defined as the number of resources that are utilized for a given computation/problem. For example, if data is distributed over 3 machines, $D_d = 3$; if data is distributed over three machines but the computational tasks over 4 machines, $D_d = 4$.

(*a*) *Experimental Configuration*

As explained before, for our experiments, we use an All-Pairs implementation that utilises SAGA. An XML configuration file defines various initial parameters which alter the behavior of our All-Pairs implementation. The configuration file defines the location of data that comprises 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 following experiments can be completely described by a tuple of the following form

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

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 work-loads that the total work is decomposed into, i.e., number of work assignments generated. M_c will be represented by configurations $C1, C2, C3, C4$, or $C5$, which are 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

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, X_2 store data
$C4$	$X_1(Y,Y), X_2(Y,Y)$	X_1, X_2 compute and store data
$C5$	$X_1(Y,Y), X_2(Y,Y), X_3(Y,Y)$	X_1, X_2, X_3 compute and store data

Table 1: Here we show the machine configurations M_c (tuple 4.2) that we use in our 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.

Table 1); fs is the type of filesystem used; m is the method used to access that filesystem; and r is the degree of replication utilized in the experiment (with a default value of 1). However, for CloudStore, we investigate the performance with $r = 1, 2$ and 3.

In our experiments, we have three (fs, m) configurations, and five X configurations. Our (fs, m) configurations are (local, local), (local, gridFTP), and (CloudStore, direct). By direct we mean CloudStore controls the data access. Our X configurations are enumerated in Table 1. For one machine, $C1 = X_1(Y,Y)$, where resource X_1 has both the data and the computing; 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. By granularity, we mean the ability to be distributed to many resources. If N_c is too small, there may exist idle resources unable to assist the workload while the ones that are participating are overloaded.

A sample description of an experiment will now be explained:

$$(287\text{MB}, 8, C2, CloudStore, Direct, 1), \tag{4.3}$$

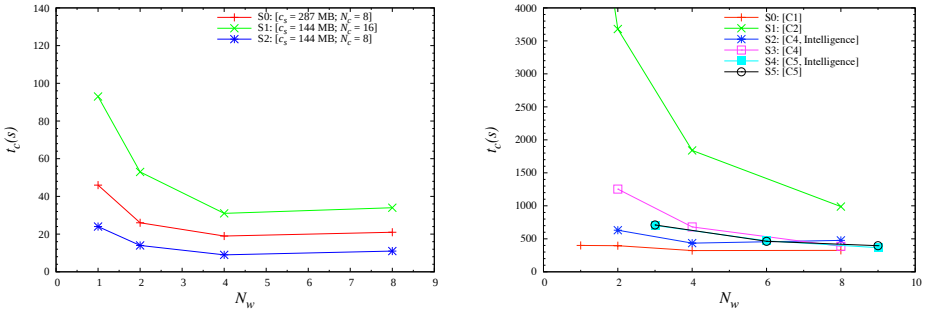
shows that each element of a set is 287MB in size; we have 8 assignments; the computational resource X_1 does calculations, but does not have data stored, while computational resource X_2 does not calculate, but stores the data; the filesystem used is CloudStore, it directly access the files, and we have a replication factor of 1 for our data. The machines X_i we use for our experiments are part of LONI (Louisiana Optical Network Initiative). For most of our 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 our experiments has a fixed distribution of data, fixed available computational resources, and fixed sets of pairs to operate with. These variabilities listed above (tuple 4.2) are manipulated to determine causes for different I/O complexities observed in an attempt to build an understanding of issues that arise when utilising data-intensive applications. It is also notable that the following experiments were consistent and reproducible for a given time, but could vary if run more than a few hours apart. This variance is attributable to the amount of use that our computing environment (LONI) was experiencing at the time of the experiment. However, since there are no standardised mechanisms to inquire the amount of load of compute systems we could not quantize this variance.

(*b*) *Experiment I: Baseline Performance*

In the first experiment, we do not have an actual operation being applied on the pairs, giving us $t_{compute} = 0$, that is, 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 a grid (LONI), without any specific data placement strategy; also, no replication or fault-tolerance takes place. The application sequentially assigns sets of pairs to the first available computational resource. All data is accessed via the gridFTP protocol. An important fact to notice is the essentially random mapping of data sets to computational resources based on availability. This is to mimic a naive data-intensive application.

In figure 3, we show our results for data accessed without any protocol (local case) and data accessed via gridFTP. Using our All-Pairs framework accessing the data using gridFTP protocol had an overhead which can be noted by looking at the $y-$scales of both graphs. In figure 3(a), our local cases, we see that working with a smaller data set ($c_s \sim$ 144MB, $N_c = 8$, 1.15GB total) took about half the time than working with a data set double the size ($c_s = 287$MB, $N_c = 8$, 2.3GB total). We also see that when working with the same data set size (2.3GB), but partitioned differently, i.e., S0 ($c_s = 287$MB, $N_c = 8$) vs. S1 ($c_s = 144$MB, $N_c = 16$), t_c increased for S1, due to added transfer time by doubling the number of files, although we decreased the file size by half. In figure 3(b), where we used gridFTP to access the files, we see that a single machine took less time compared to the configurations that involved two machines. Also, having to access data remotely was a disadvantage, $t_c(\text{S1}) > t_c(\text{S2})$. For the one machine configuration, t_c was approximately constant, probably caused by an I/O bound. For all our 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.



(a) $(M_c, fs, m) = (C1, \text{local}, \text{local})$ (b) $(c_s, N_c, fs, m) = (287\text{MB}, 8, \text{local}, \text{gridFTP})$

Figure 3: GridFTP vs. local run. We plot the time to completion t_c vs. the number of workers N_w. Note the scale, the local case took less t_c than the gridFTP case. In Fig. 3(a), we note that by doubling the data set size we doubled t_c, and that we created an overhead in S1 (vs. S0) by increasing t_x as a result of having more files. In Fig. 3(b), the two-machine configurations took more time than using a single machine. As expected, computing in one machine, while having the data stored in another (S1– not pictured $t_c(N_w = 1) = 7360$), took longer compared to having some data stored in the resource performing computations (S3). For up to 8 workers, t_c decreased as N_w increased, with the exception of a single machine where we reached an I/O bound. Here, we also compare the gridFTP and the intelligent approaches for two and three machines (Sec. c). For the two-machine case, we observed a performance improvement by using our intelligent approach (S2 vs. S3). However, with more resources involved, our intelligent system did not improve t_c (S4 vs. S5).

(*c*) *Experiment II: Intelligence Based System*

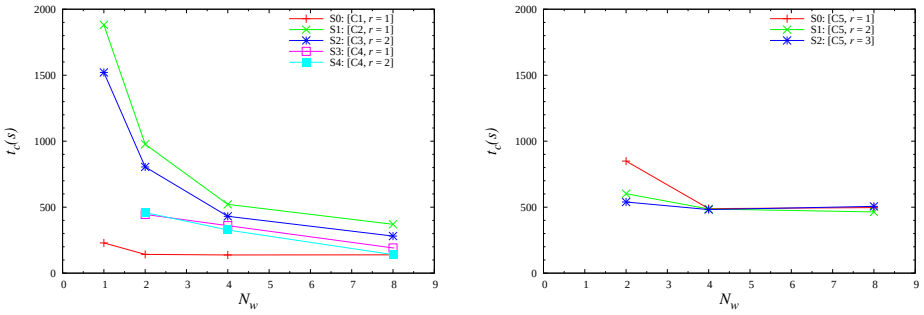
The second experiment is similar to the first, except the All-Pairs application is aware of the data location before determining whether or not to assign a certain set of data-dependent computation to an idle worker. Inspired by earlier work (Jha et al. 2007), this version of the application performs an extra step during application startup 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 utilised at runtime when the application needs to map an idle worker to an unprocessed set of pairs defined in the XML file. This changes the first-available computational resource assignment mechanism described in the first experiment to an intelligent based system. Though ping is not very sophisticated in terms of describing a network's behavior, it is a first-approximation to a performance aware data-placement strategy. We also experimented with the netperf application (Netperf 2009) as a method to describe a network's behavior, but since we were working with such a static set of resources (LONI), the same data graph was generated as with ping. The netperf-based intelligent system added approximately 8 seconds per resource of overhead due to the nature of the network evaluation, but yielded no benefit. Netperf has the advantage of being able to determine throughput and bandwidth over multiple protocols. 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 intelligence includes the time spent pinging hosts and building the graph data structure. The total time spent for this overhead was negligible at approximately two seconds per application run. In figure 3(b), we see we achieved a reduction in time to completion due to the use of intelligence. However, when the same tests were performed utilising three resources instead of two, the intelligence seemed to offer no significant reduction. The explanation that we propose for this, is that the sets of pairs defined in the configuration file at application start were geographically equally dispersed throughout the network. Essentially, each set of pairs had approximately the same cost calculated by using the graph data structure. Each set of pairs would take the same time to compute using any computational resource.

(*d*) *Experiment III: CloudStore*

The third experiment provides information into CloudStore's performance in handling data locality issues. The same All-Pairs application as in Experiments I and II is used, except all data is stored on the distributed filesystem 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 utilise the distributed filesystem. Again, for our first set of results, we don't add our comparison function, giving us $t_{compute} = 0$.

As done above for the first experiment, we attempted to capture how compute time scales under these configurations, see figure 4. Having data remotely affected the performance. We can see that computing in one resource while the data was on another (S1) took the most amount of time. Having data in the resource that was computing helped performance (S2, S3, S4). The number of machines to which workload was assigned was also important. Placing workload on two machines also decreased t_c (S2 vs. S3). We varied



(a) One and two machines (b) Three machines

Figure 4: $(c_s, N_c, fs, m) = (287\text{MB}, 8, \text{CloudStore}, \text{Direct})$. The figure on the left demonstrates All-Pairs' performance with CloudStore locally and on two different machines. The figure on the right demonstrates how this scales to three machines, for degree of replication $r = 1, 2,$ and 3. CloudStore performed better than our local and intelligent approaches (see Fig. 3). Again, having data in the resource with the workload decreased t_c. When data was 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 2 workers.

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 one, data was almost certainly co-located 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 to 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 to 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 of configurations, shows that there was still an overhead; t_x and $t_{I/O}$ were not the same at the times we run our All-Pairs framework for both cases.

We also compared the results with no comparison function ($t_{compute} = 0$) for two different data set 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 have eight assignments. We defined two quantities, $\Delta t_c^d = t_c(c_s = 287\text{MB}) - t_c(c_s = 144\text{MB})$, and $t_{OH} = 2 \times t_c(c_s = 144\text{MB}) - t_c(c_s = 287\text{MB})$. 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); this is, it took about 10 seconds less for a 2.3GB set vs. a 1.15GB set. Figure 6(b) shows that for most of our cases, there was an overhead which decreased with the number of workers. It also show us that the remote data configuration was the one with the most overhead. Moreover, our S3 case seemed to use a "non scalable" infrastructure, as the overhead increased with eight workers.

We compared the lowest times for both our intelligent approach and CloudStore as a function of the number of resources N_r in figure 7(a). CloudStore performed better for one

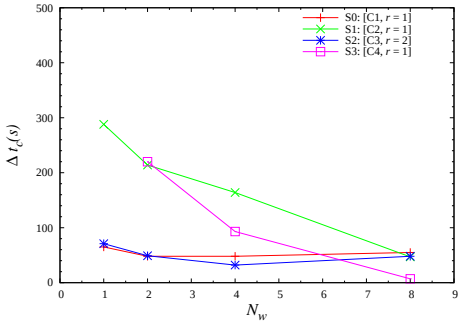
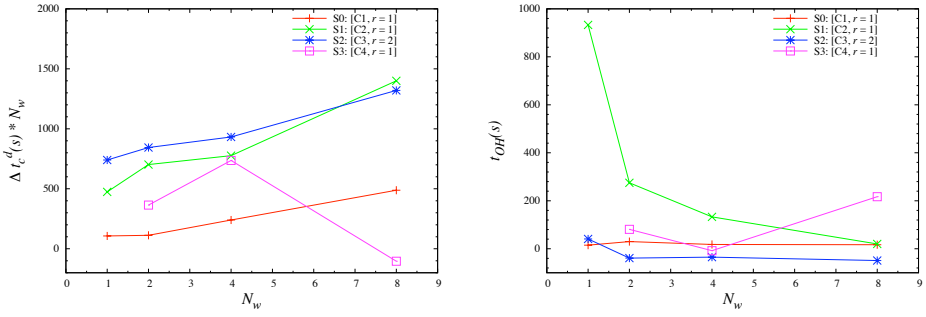


Figure 5: $(c_s, N_c, fs, m) = (144\text{MB}, 8, \text{CloudStore}, \text{Direct})$. Comparison of CloudStore using the All-Pairs application with and without actual computation. Δt_c is defined as the time it takes to run our framework where we include our genome comparison function, minus the time it takes to run it when we don't include the comparison function. For the case of $c_s = 144$MB, our 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 our configurations, showing an overhead, probably caused by different network conditions at the times our runs were performed.



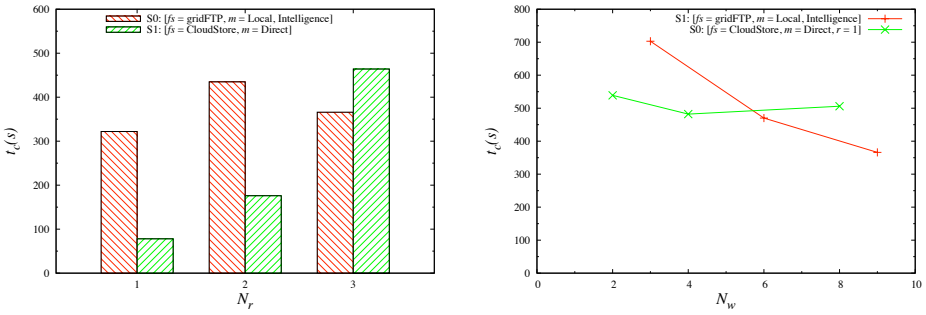
(a) $\Delta t_c^d (= t_c(287\text{MB}) - t_c(144\text{MB})) \times N_w$ (b) $t_{OH} = 2 \times t_c(144\text{MB}) - t_c(287\text{MB})$

Figure 6: $(N_c, fs, m) = (8, \text{CloudStore}, \text{Direct})$. Here we define two quantities, Δt_c^d, and t_{OH}. Δt_c^d is the difference between t_c found for data set sizes 1.15GB and 2.3GB, this is, for chunk sizes $c_s = 144$MB and 287MB. t_{OH} is the overhead time of working with chunks of 287MB in size vs. 144MB chunks twice. In figure 6(a) we see that the difference decreased with the number of workers, but did not scale linearly with N_w. In figure 6(b), we see that S1 (remote data) was the one with the most overhead.

and two machines, but not for three resources, when CloudStore decreased its performance. The lowest times of our intelligent approach were about the same for $N_r = 1, 2, 3$. In figure 7(b) we plotted our completion time as a function of the number of workers for three resources. All the resources had workload assigned and data stored. CloudStore's performance did not vary significantly with the number of workers, while our intelligent approach performed better as we increased the number of resources from one to three.

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



(a) $(c_s, N_c) = (287\text{MB}, 8)$ (b) $(c_s, N_c, M_c) = (287\text{MB}, 8, C5)$

Figure 7: Figure 7(a) shows the lowest times we achieved for a given number of resources (N_r). These lowest times did not vary much for our intelligent method, while CloudStore decreased its performance by adding number of resources (up to three). Figure 7(b) demonstrates 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 our intelligent approach improved its performance with the number of workers.

over a range of infrastructure. This capability enabled us to compare and contrast two different approaches – simple application-level data placement heuristics versus distributed file systems – for executing distributed data-intensive applications. It should be noted that utilising SAGA to access CloudStore-based and gridFTP-based files, introduced a small but negligible overhead.

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 be the case as the volume of data is largely increased (which we anticipate to be tens of GB of data). The remote 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 $t_c(\text{local})$, 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 due 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. Our experiments scaled to three distinct resources not due to any fundamental scalability limitation in our approach, but due to the inabilty to find more than three similar resources.

Overall, the use of CloudStore lowers t_c compared to the intelligent and the gridFTP approaches. Our simple intelligent approach did not perform as well as CloudStore for one and two machines, but performed better than gridFTP. For three machines, the relative performance of CloudStore and the intelligent approach depended upon the N_w. Our results for three resources showed that gridFTP continued to experience improvements in performance as the number of resources increased, while CloudStore leveled off.

We will extend this work 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 cor-

relation in data-access have been observed elsewhere (Doraimani and 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 be both interesting and rewarding.

Acknowledgment: 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. SJ acknowledges the e-Science Institute, Edinburgh for supporting the research theme, "Distributed Programming Abstractions" and theme members for shaping many important ideas. BRM 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.
URL: *http://kosmosfs.sourceforge.net*

Dean, J. and Ghemawat, S.: 2008, Mapreduce: simplified data processing on large clusters, *Commun. ACM* **51**(1), 107–113.

Doraimani, S. and Iamnitchi, A.: 2008, File grouping for scientific data management: lessons from experimenting with real traces, *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, ACM, New York, NY, USA, pp. 153–164.

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

Hey, T.: 2009, The fourth paradigm: Data-intensive scientific discovery.

Jha, S., Kaiser, H., Merzky, A. and Weidner, O.: 2007, Grid interoperability at the application level using saga, *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, IEEE Computer Society, Washington, DC, USA, pp. 584–591.
URL: *http://saga.cct.lsu.edu/publications/saga_gin.pdf*

Merzky, A., Stamou, K., Jha, S. and Katz, D. S.: 2009, A fresh perspective on developing and executing dag-based distributed applications: A case-study of saga-based montage.

Miceli, C., Miceli, M., Jha, S., Kaiser, H. and Merzky, A.: 2009, Programming abstractions for data intensive computing on clouds and grids, *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, Washington, DC, USA, pp. 478–483.

Moretti, C., Bulosan, J., Thain, D. and Flynn, P.: 2008, All-pairs: An abstraction for data-intensive cloud computing, *International Parallel and Distributed Processing Symposium*, pp. 1–11.

Netperf: 2009.
URL: *http://www.netperf.org/netperf/*