SPECIAL ISSUE PAPER

Cloud computing paradigms for pleasingly parallel biomedical applications

Thilina Gunarathne*,†, Tak-Lon Wu, Jong Youl Choi, Seung-Hee Bae and Judy Qiu

School of Informatics and Computing / Pervasive Technology Institute, Indiana University, Bloomington, IN, USA

SUMMARY

Cloud computing offers exciting new approaches for scientific computing that leverage major commercial players' hardware and software investments in large-scale data centers. Loosely coupled problems are very important in many scientific fields, and with the ongoing move towards data-intensive computing, they are on the rise. There exist several different approaches to leveraging clouds and cloud-oriented data processing frameworks to perform pleasingly parallel (also called embarrassingly parallel) computations. In this paper, we present three pleasingly parallel biomedical applications: (i) assembly of genome fragments; (ii) sequence alignment and similarity search; and (iii) dimension reduction in the analysis of chemical structures, which are implemented utilizing a cloud infrastructure service-based utility computing models of Amazon Web Services (Amazon.com Inc., Seattle, WA, USA) and Microsoft Windows Azure (Microsoft Corp., Redmond, WA, USA) as well as utilizing MapReduce-based data processing frameworks Apache Hadoop (Apache Software Foundation, Los Angeles, CA, USA) and Microsoft DryadLINQ. We review and compare each of these frameworks, performing a comparative study among them based on performance, cost, and usability. High latency, eventually consistent cloud infrastructure service-based frameworks that rely on off-the-node cloud storage were able to exhibit performance efficiencies and scalability comparable to the MapReduce-based frameworks with local disk-based storage for the applications considered. In this paper, we also analyze variations in cost among the different platform choices (e.g., Elastic Compute Cloud instance types), highlighting the importance of selecting an appropriate platform based on the nature of the computation.

Received 20 October 2010; Revised 4 April 2011; Accepted 17 April 2011

KEY WORDS: cloud technology; map reduce; bioinformatics

1. INTRODUCTION

Scientists are overwhelmed by the increasing amount of data processing needs that have arisen from the massive amount of data now flowing through virtually every field of science. Preprocessing, processing, and analyzing these large amounts of data are a unique and challenging problem; however, it also opens up many opportunities for both computational and computer scientists. Jim Gray has noted that increasingly, scientific breakthroughs will be powered by computing capabilities that support the ability of researchers to analyze massive data sets. Aptly, he dubbed data-intensive scientific discovery 'the fourth scientific paradigm of discovery' [1].

Cloud computing offerings by major commercial players provide on-demand computational services over the web, which can be purchased within a matter of minutes by simply using a credit card. The utility computing model of these cloud computing offerings opens up exciting new opportunities for computational scientists to perform their computations because this type of model is well-suited

to the scientists' occasional needs for resource-intensive computing. Another interesting feature is the ability to increase the throughput of their computations by horizontally scaling computing resources without incurring any additional overhead costs. For example, 100 hours in 10 cloud compute nodes cost the same as 10 hours in 100 cloud compute nodes. This is facilitated by the virtually unlimited resource availability of cloud computing infrastructures, which are backed by the world's largest data centers owned by major commercial players such as Amazon, Google, and Microsoft. We expect that the economies of scale enjoyed by cloud providers would translate into lower costs for users. Cloud computing platforms also offer a rich set of distributed cloud infrastructure services including storage, messaging, and database services with cloud-specific service guarantees. These services can be leveraged to build and deploy scalable-distributed applications on cloud environments. At the same time, we can notice the emergence of cloud-oriented data processing technologies and frameworks such as MapReduce [2] framework. MapReduce frameworks allow users to effectively perform distributed computations in increasingly brittle environments, such as commodity clusters and computational clouds. Apache Hadoop [3] and Microsoft DryadLINQ [4] are two such distributed parallel data processing frameworks that support MapReduce type computations.

A pleasingly parallel application is an application that can be parallelized, thus requiring minimal effort to divide the application into independent parallel parts. Each independent parallel part has very minimal or no data, synchronization or ordering dependencies with the others. These applications are good candidates for computing clouds and compute clusters with no specialized interconnections. There are many scientific applications that fall under this category. Examples of pleasingly parallel applications include Monte Carlo simulations, BLAST searches, parametric studies, and image processing applications such as ray tracing. Most of the data cleansing and preprocessing applications can also be classified as pleasingly parallel applications. Recently, the relative number of pleasingly parallel scientific workloads has grown because of the emergence of data-intensive computational fields such as bioinformatics.

In this paper, we introduce a set of frameworks that have been constructed using cloud-oriented programming models to perform pleasingly parallel computations. Using these frameworks, we present implementations of biomedical applications such as the Cap3 [5] sequence assembly, BLAST sequence search and GTM Interpolation. We analyze the performance, cost, and usability of different cloud-oriented programming models using the above-mentioned implementations. We use Amazon Web Services [6] and Microsoft Windows Azure [7] cloud computing platforms and Apache Hadoop [3] MapReduce and Microsoft DryaLINQ [4] as the distributed parallel computing frameworks.

2. CLOUD TECHNOLOGIES AND APPLICATION ARCHITECTURE

Processing large data sets using existing sequential executables is a common use case encountered in many scientific applications. Many of these applications exhibit pleasingly parallel characteristics in which the data can be independently processed in parts. In the following sections, we explore cloud programming models and the frameworks that we developed to perform pleasingly parallel computations.

2.1. Classic Cloud architecture

2.1.1. Amazon Web Services. Amazon Web Services (AWS) [6] are a set of cloud computing services by Amazon, offering on-demand computing and storage services including, but not limited to, Elastic Compute Cloud (EC2), Simple Storage Service (S3), and Simple Queue Service (SQS).

EC2 provides users the option to lease virtual machine instances that are billed hourly and that allow users to dynamically provision resizable virtual clusters in a matter of minutes through a web service interface. EC2 supports both Linux and Windows virtual instances. EC2 follows an approach that uses infrastructure as a service; it provides users with 'root' access to the virtual machines, thus providing the most flexibility possible. Users can store virtual machine snapshots as Amazon Machine Images, which can then be used as templates for creating new instances. Amazon EC2 offers a variety of hourly billed instance sizes with different price points, giving users a richer

set of options to choose from, depending on their requirements. One particular instance type of interest is the High-CPU Extra Large (HCXL) instances, which cost the same as the Extra Large (XL) instances but offer greater CPU power and less memory than XL instances. Table I provides a summary of the EC2 instance types used in this paper. The clock speed of a single EC2 compute unit is approximately 1 to 1.2 GHz. The Small instance type with a single EC2 compute unit is only available in a 32-bit environment, whereas the larger instance types also support a 64-bit environment.

Simple Queue Service is a reliable, scalable, distributed web-scale message queue service that is eventually consistent and ideal for small, short-lived transient messages. SQS provides a Representational State Transfer-based web service interface that enables any Hypertext Transfer Protocol (HTTP) capable client to use it. Users can create an unlimited number of queues and send an unlimited number of messages. SQS does not guarantee the order of the messages, the deletion of messages or the availability of all the messages for a request, though it does guarantee eventual availability over multiple requests. Each message has a configurable visibility timeout. Once it is read by a client, the message will be hidden from other clients until the visibility time expires. The message reappears upon expiration of the timeout as long as it is not deleted. The service is priced based on the number of Application Programming Interface (API) requests and the amount of data transfer.

Storage Service provides a web-scale distributed storage service where users can store and retrieve any type of data through a web services interface. S3 is accessible from anywhere in the web. Data objects in S3 are access controllable and can be organized into buckets. S3 pricing is based on the size of the stored data, amount of data transferred and the number of API requests.

2.1.2. Microsoft Azure Platform. Windows Azure Compute only supports Microsoft Windows based virtual machine instances and offers a limited variety of instance types compared to Amazon EC2. As shown in Table II, Azure instance type configurations and the cost scales up linearly from Small, Medium, Large to Extra-Large. All Azure instances are available in a 64-bit environment. It has been speculated that the clock speed of a single CPU core in Azure is approximately 1.5 GHz to 1.7 GHz. During our performance testing using the Cap3 program (Section 4), we found that 8 Azure small instances perform comparably to a single Amazon HCXL instance with 20 EC2 compute units. Azure Compute follows the platform of a service approach and offers the .net runtime as the platform. Users can easily deploy their programs through a web application as an Azure deployment package as well as directly from Visual Studio IDE. Azure Storage Blob service and Azure Queue service respectively provide functionality similar to Amazon S3 and Amazon SQS services described above.

Table I. Selected Elastic Compute Cloud (EC2) instance types.

Instance type	Memory (GB)	EC2 compute units	Actual CPU cores	Cost per hour ($)
Large (L)	7.5	4	2 × (∼ 2GHz)	0.34
Extra Large (XL)	15	8	4 × (∼ 2GHz)	0.68
High-CPU Extra Large (HCXL)	7	20	8 × (∼ 2.5GHz)	0.68
High-Memory Quadruple Extra Large (HM4XL)	68.4	26	8 × (∼ 3.25GHz)	2.00

Table II. Microsoft Windows Azure instance types.

Instance type	CPU cores	Memory (GB)	Local disk space (GB)	Cost per hour ($)
Small	1	1.7	250	0.12
Medium	2	3.5	500	0.24
Large	4	7	1000	0.48
Extra Large	8	15	2000	0.96

2.1.3. Classic Cloud processing model. Figure 1 depicts the Classic Cloud processing model. Varia [8] and Chappell [9] described similar architectures that are implemented using Amazon and Azure processing models respectively. The Classic Cloud processing model follows a task processing pipeline approach with independent workers. It uses the cloud instances (EC2/Azure Compute) for data processing and uses Amazon S3/Windows Azure Storage for the data storage. For the task scheduling pipeline, it uses an Amazon SQS or an Azure queue as a queue of tasks where every message in the queue describes a single task. The client populates the scheduling queue with tasks, while the worker-processes running in cloud instances pick tasks from the scheduling queue. The configurable visibility timeout feature of Amazon SQS and Azure Queue service is used to provide a simple fault tolerance capability to the system. The workers delete the task (message) in the queue only after the completion of the task. Hence, a task (message) will get processed by some worker if the task does not get completed with the initial reader (worker) within the given time limit. Rare occurrences of multiple instances processing the same task or another worker re-executing a failed task will not affect the result due to the idempotent nature of the independent tasks.

For the applications discussed in this paper, a single task comprises of a single input file and a single output file. The worker processes will retrieve the input files from the cloud storage through the web service interface using HTTP and will process them using an executable program before uploading the results back to the cloud storage. In this implementation, the user can configure the workers to use any executable program in the virtual machine to process the tasks, provided that it takes input in the form of a file. Our implementation uses a monitoring message queue to monitor the progress of the computation. One interesting feature of the Classic Cloud framework is the ability to extend it to use the local machines and clusters side by side with the clouds. Although it might not be the best option because of the data being stored in the cloud, one can start workers in computers outside of the cloud to augment compute capacity.

2.2. Apache Hadoop MapReduce

Apache Hadoop [3] is an open source implementation of the Google MapReduce [2] technology. Apache Hadoop MapReduce uses Hadoop Distributed File System (HDFS) for data storage, which stores the data across the local disks of the compute nodes while presenting a single file system view

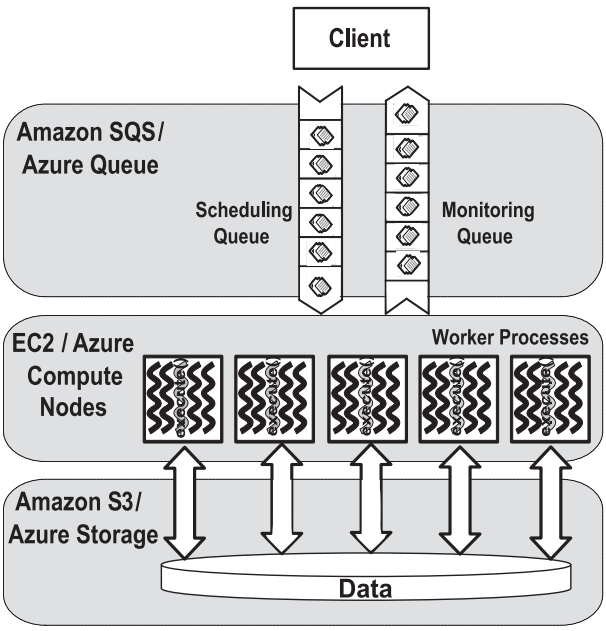

Figure 1. Classic Cloud processing model. SQS, Simple Queue Service; EC2, Elastic Compute Cloud; S3, Simple Storage Service.

through the HDFS API. HDFS is designed for deployment on commodity clusters and achieves reliability through replication of data across nodes. Hadoop optimizes the data communication of MapReduce jobs by scheduling computations near the data using the data locality information provided by HDFS. Hadoop follows a master node with many client workers approach and uses a global queue for the task scheduling, achieving natural load balancing among the tasks. Hadoop performs data distribution and automatic task partitioning based on the information provided in the master program and based on the structure of the data stored in HDFS. MapReduce architecture reduces the overheads of data transfer by overlapping data communication with the computations. Hadoop performs duplicate execution of slower executing tasks and handles tasks failures by rerunning of the failed tasks.

As shown in Figure 2, the pleasingly parallel application framework on Hadoop is developed as a set of map tasks that process the given data splits (files) using the configured executable program. Input to a map task comprises of key, value pairs, where by default Hadoop parses the contents of the data split to read them. Most of the legacy data processing applications expect a file path as the input instead of the contents of the file, which is not possible with the Hadoop built-in input formats and record readers. We implemented a custom InputFormat and a RecordReader for Hadoop to provide the file name and the HDFS path of the data split, respectively, as the key and the value for the map function, while preserving the Hadoop data locality-based scheduling.

2.3. DryadLINQ

Dryad [10] is a framework developed by Microsoft Research as a general-purpose distributed execution engine for coarse-grain parallel applications. Dryad applications are expressed as directed acyclic dataflow graphs (DAGs), where vertices represent computations and edges represent communication channels between the computations. DAGs can be used to represent MapReduce type computations and can be extended to represent many other parallel abstractions too. Similar to the MapReduce frameworks, the Dryad scheduler optimizes the data transfer overheads by scheduling the computations near data and handles failures through rerunning of tasks and duplicate task execution. In the Dryad version we used for this paper, data for the computations need to be partitioned manually and stored beforehand in the local disks of the computational nodes via Windows-shared directories. Dryad is available for academic usage through the DryadLINQ [4] API, which is a

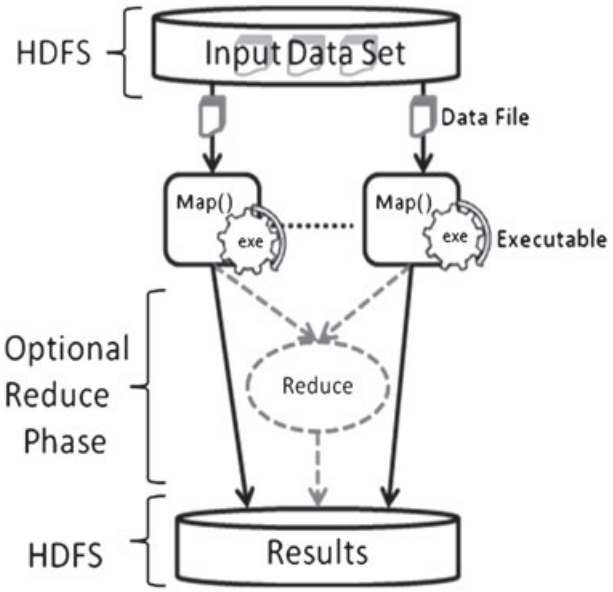

Figure 2. Hadoop MapReduce processing model. HDFS, Hadoop Distributed File System.

high-level declarative language layer on top of Dryad. DryadLINQ queries get translated into distributed Dryad computational graphs in the run time. DryadLINQ can be used only with Microsoft Windows High Performance Computing (HPC) clusters. The DryadLINQ implementation of the framework uses the DryadLINQ 'select' operator on the data partitions to perform the distributed computations. The resulting computation graph looks much similar to Figure 2, where instead of using HDFS, Dryad will use the Windows-shared local directories for data storage. Data partitioning, data distribution, and the generation of metadata files for the data partitions are implemented as part of our pleasingly parallel application framework.

2.4. Usability of the technologies

Implementing the above-mentioned application framework using the Hadoop and DryadLINQ data processing frameworks was easier than implementing them from the scratch using cloud infrastructure services as the building blocks (Table III). Hadoop and DryadLINQ take care of scheduling, monitoring, and fault tolerance. With Hadoop, we had to implement a Map function, which copy the input file from HDFS to the working directory, execute the external program as a process, and finally upload the result file to the HDFS. It was also necessary to implement a custom InputFormat and a RecordReader to support file inputs to the map tasks. With DryadLINQ, we had to implement a side effect-free function to execute the program on the given data and copy the result to the output-shared directory. But significant effort had to be spent on implementing the data partition and the distribution programs to support DryadLINQ.

Elastic Compute Cloud and Azure Classic Cloud implementations involved more effort than the Hadoop and DryadLINQ implementations, as all the scheduling, monitoring, and fault tolerance had to be implemented from scratch using the cloud infrastructure services' features. The deployment process was easier with Azure as opposed to EC2, in which we had to manually create instances, install software, and start the worker instances. On the other hand, the EC2 infrastructure gives developers more flexibility and control. Azure software development kit (SDK) provides better development, testing, and deployment support through Visual Studio integration. The local development compute fabric and the local development storage of the Azure SDK make it much easier to test and debug Azure applications. Although the Azure platform is heading towards providing a more developer-friendly environment, it still lags behind in terms of the infrastructure maturity Amazon AWS has accrued over the years.

Table III. Summary of cloud technology features.

	AWS/Azure	Hadoop	DryadLINQ
Programming patterns	Independent job execution, more structure possible using client side driver program.	MapReduce	DAG execution, extensible to MapReduce and other patterns
Fault tolerance	Task re-execution based on a configurable time out	Re-execution of failed and slow tasks.	Re-execution of failed and slow tasks.
Data storage and communication	S3/Azure Storage. Data retrieved through HTTP.	HDFS parallel file system. TCP based Communication	Local files
Environments	EC2/Azure virtual instances, local compute resources	Linux cluster, Amazon Elastic MapReduce	Windows HPCS cluster
Scheduling and load balancing	Dynamic scheduling through a global queue, providing natural load balancing	Data locality, rack aware dynamic task scheduling through a global queue, providing natural load balancing	Data locality, network topology aware scheduling. Static task partitions at the node level, suboptimal load balancing

AWS, Amazon Web Service; S3, Storage Service; HTTP, Hypertext Transfer Protocol; EC2, Elastic Compute Cloud; HDFS, Hadoop Distributed File System; TCP, Transmission Control Protocol; DAG, directed acyclic graph; HPCS, High Performance Computing Server.

3. EVALUATION METHODOLOGY

In the performance studies, we use parallel efficiency as the measure by which to evaluate the different frameworks. Parallel efficiency is a relatively good measure for evaluating the different approaches we use in our studies, as we do not have the option of using identical configurations across the different environments. At the same time, we cannot use efficiency to directly compare the different technologies. Even though parallel efficiency accounts for the system dissimilarities that affect the sequential and the parallel run time, it does not reflect other dissimilarities, such as memory size, memory bandwidth, and network bandwidth. Parallel efficiency for a parallel application on p number of cores can be calculated using the following formula [11]:

$$\text{Parallel Efficiency} = \frac{T_1}{pT_p} \tag{1}$$

In this equation, T_p is the parallel run time for the application. T_1 is the best sequential run time for the application using the same data set or a representative subset. In this paper, the sequential run time for the applications was measured in each of the different environments, having the input files present in the local disks, avoiding the data transfers.

The average run time for a single computation in a single core is calculated for each of the performance tests using the following formula. The objective of this calculation is to give readers an idea of the actual performance they can obtain from a given environment for the applications considered in this paper.

$$\text{Avg. run time for a single computation in a single core} = \frac{pT(p)}{\text{No. of computations}} \tag{2}$$

Because of the richness of the instance type choices Amazon EC2 provides, it is important to select an instance type that optimizes the balance between performance and cost. We present instance type studies for each of our applications for the EC2 instance types mentioned in Table I using 16 CPU cores for each study. EC2 Small instances were not included in our study because they do not support 64-bit operating systems. We do not present results for Azure Cap3 and GTM Interpolation applications, as the performance of the Azure instance types for those applications scaled linearly with the price. However, the total size of memory affected the performance of BLAST application across Azure instance types; hence, we perform an instance type study for BLAST on Azure.

Cloud virtual machine instances are billed hourly. When presenting the results, the 'Compute Cost (hour units)' assumes that particular instances are used only for the particular computation and that no useful work is carried out for the remainder of the hour, effectively making the computation responsible for the entire hourly charge. The 'Amortized Cost' assumes that the instance will be used for useful work for the remainder of the hour, making the computation responsible only for the actual fraction of time during which it was executed. The vertical axes of the EC2 cost figures (Figures 3, 4 and 12) and the horizontal axis labeling of the EC2 compute time figures (Figures 5, 6 and 13) are labeled in the format 'Instance Type' − 'Number of Instances' × 'Number of Workers per Instance'. For an example, HCXL − 2 × 8 means two HCXL instances were used with eight workers per instance.

When presenting the results used in this paper, we considered a single EC2 XL instance, with 20 EC2 compute units as eight actual CPU cores, whereas an Azure Small instance was considered as a single CPU core. In all of the test cases, it is assumed that the data was already present in the framework's preferred storage location. We used Apache Hadoop version 0.20.2 and DryadLINQ version 1.0.1411.2 (November 2009) for our studies.

In Gunarathne *et al.* [12], we investigated the sustained performance of Amazon AWS and Windows Azure cloud infrastructures for MapReduce type applications over a week's time during different times of the day. The performance variations we observed were very minor, with standard deviations of 1.56% for Amazon AWS and 2.25% for Windows Azure, with no noticeable correlations with the day of the week or the time of the day. Hence, we assume that the performance results we obtained for this paper would not depend on such variables.

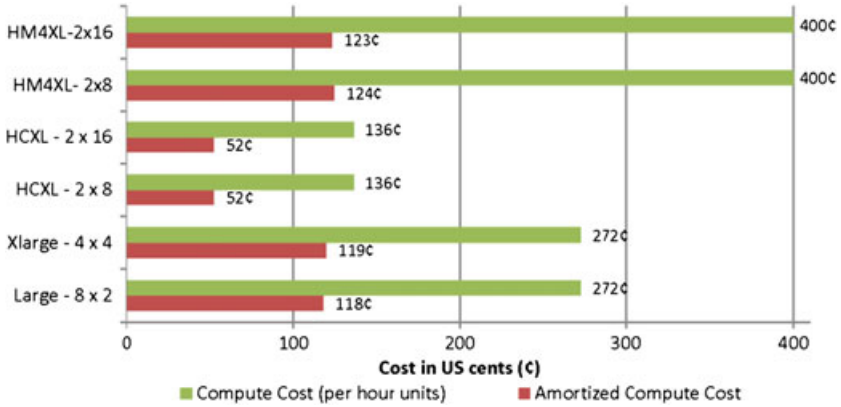

Figure 3. Cap3 cost with different Elastic Compute Cloud instance types. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large; Xlarge, Extra Large.

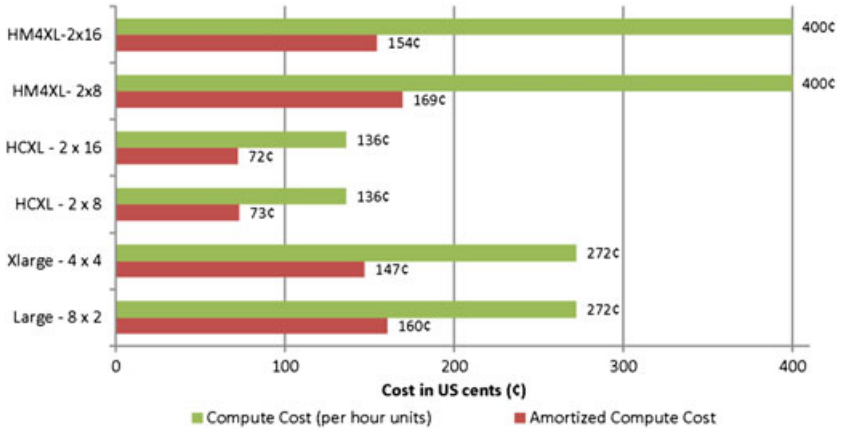

Figure 4. Cost to process 64 query files using BLAST in EC2. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large; Xlarge, Extra Large.

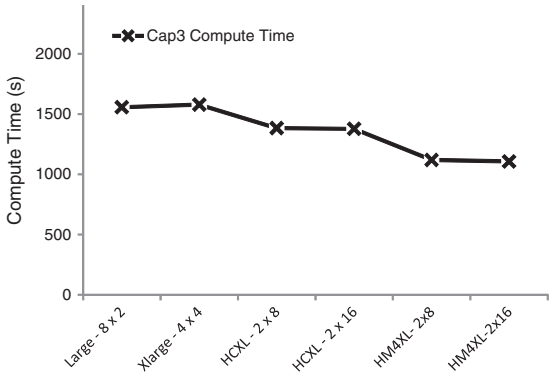

Figure 5. Cap3 compute time with different instance types. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large; Xlarge, Extra Large.

4. CAP3

Cap3 [5] is a sequence assembly program that assembles DNA sequences by aligning and merging sequence fragments to construct whole genome sequences. Sequence assembly is an integral part of genomics as the current DNA sequencing technology, such as shotgun sequencing, is capable

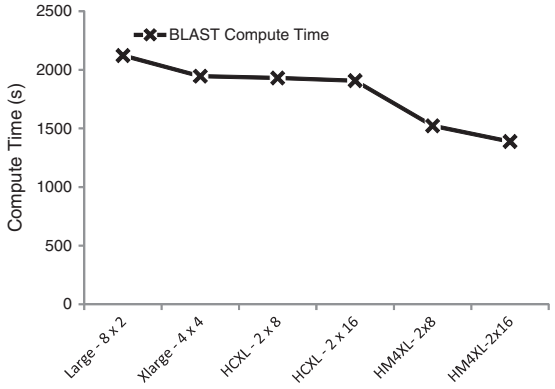

Figure 6. Time to process 64 query files using BLAST in Azure. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large; Xlarge, Extra Large.

of reading only parts of genomes at once. The Cap3 algorithm operates on a collection of gene sequence fragments presented as FASTA-formatted files. It removes the poor regions of the DNA fragments, calculates the overlaps between the fragments, identifies and removes the false overlaps, joins the fragments to form contigs of one or more overlapping DNA segments, and finally, through multiple sequence alignment, generates consensus sequences.

The increased availability of DNA sequencers are generating massive amounts of sequencing data that needs to be assembled. Cap3 program is often used in parallel with lots of input files because of the pleasingly parallel nature of the application. The run time of the Cap3 application depends on the contents of the input file. Cap3 is less memory intensive than the GTM Interpolation and BLAST applications discussed in Sections 5 and 6. The size of a typical data input file for the Cap3 program and the result data file range from hundreds of kilobytes to few megabytes. Output files resulting from the input data files can be collected independently and do not need any combining steps.

4.1. Performance with different Elastic Compute Cloud instance types

Figures 3 and 5 present benchmark results for the Cap3 application on different EC2 instance types. These experiments processed 200 FASTA files, each containing 200 reads using 16 compute cores. According to these results, we can infer that memory is not a bottleneck for the Cap3 program and that performance depends primarily on computational power. Although the EC2 High-Memory Quadruple Extra Large (HM4XL) instances show the best performance because of the higher clock-rated processors, the most cost-effective performance for the Cap3 EC2 Classic Cloud application is gained using the EC2 HCXL instances.

4.2. Scalability study

We benchmarked the Cap3 Classic Cloud implementation performance using a replicated set of FASTA-formatted data files, each file containing 458 reads, and compared this to our previous performance results [13] for Cap3 DryadLINQ and Cap3 Hadoop. Sixteen HCXL instances were used for the EC2 testing, and 128 small Azure instances were used for the Azure Cap3 testing. The DryadLINQ and Hadoop bare metal results were obtained using a 32 node$\times$8 core (2.5 GHz) cluster with 16 GB of memory on each node.

Load balancing across the different subtasks does not pose a significant overhead in the Cap3 performance studies, as we used a replicated set of input data files making each subtask identical. We performed a detailed study of the performance of Hadoop and DryadLINQ in the face of inhomogeneous data in one of our previous studies [13]. In this study, we noticed better natural load balancing in Hadoop than in DryadLINQ because of Hadoop's dynamic global level scheduling as opposed to DryadLINQ's static task partitioning. We assume that cloud frameworks will be able to

perform better load balancing similar to Hadoop because they share the same dynamic scheduling global queue-based architecture.

Based on Figures 7 and 8, we can conclude that all four implementations exhibit comparable parallel efficiency (within 20%) with low parallelization overheads. When interpreting Figure 8, it should be noted that the Cap3 program performs $\sim$ 12.5% faster on Windows environment than on the Linux environment. As mentioned earlier, we cannot use these results to claim that a given framework performs better than another, as only approximations are possible, given that the underlying infrastructure configurations of the cloud environments are unknown.

4.3. Cost comparison

We estimate the cost of assembling 4096 FASTA files using Classic Cloud frameworks on EC2 and on Azure (Table IV). For the sake of comparison, we also approximate the cost of the computation using one of our internal compute clusters (32 node 24 core, 48 GB memory per node with Infiniband interconnects), with the cluster purchase cost ($\sim$ \$500, 000) depreciated over the course of 3 years plus the yearly maintenance cost ($\sim$ \$150, 000), which include power, cooling, and administration costs. We executed the Hadoop-Cap3 application in our internal cluster for this purpose. The cost for computation using the internal cluster was approximated to \$8.25 for 80% utilization, \$9.43 for 70% utilization, and \$11.01 for 60% utilization. For the sake of simplicity, we did not consider other factors such as the opportunity costs of the upfront investment, equipment failures, and upgradability. There would also be additional costs in the cloud environments for the instance time required for environment preparation and minor miscellaneous platform-specific charges, such as the number of storage requests.

5. BLAST

National Center for Biotechnology Information (NCBI) BLAST+ [14] is a very popular bioinformatics application that is used to handle sequence similarity searching. It is the latest version of BLAST [15], a multiletter command line tool developed using the NCBI C++ toolkit, to translate a FASTA-formatted nucleotide query and to compare it to a protein database. Queries are processed independently and have no dependencies between them. This makes it possible to use multiple BLAST instances to process queries in a pleasingly parallel manner. We used a subset of a real-world protein sequence data set as the input BLAST queries and used NCBI's non-redundant protein sequence database (8.7 GB), updated on 23 June 2010, as the BLAST database. In order to make the tasks coarser granular, we bundled 100 queries into each data input file resulting in files with sizes in the range of 7–8 KB. The output files for these input data range from few bytes to few megabytes.

We implemented distributed BLAST applications for Amazon EC2, Microsoft Azure, DryadLINQ, and Apache Hadoop using the frameworks that were presented in Section 2. All of the implementations download and extract the compressed BLAST database (2.9 GB compressed) to a local disk partition of each worker prior to beginning the processing of the tasks. Hadoop-BLAST uses the Hadoop-distributed cache feature to distribute the database. We added a similar data preloading feature to the Classic Cloud frameworks, in which each worker will download the specified file from the cloud storage at the time of startup. In the case of DryadLINQ, we manually distributed the database to each node using Windows-shared directories. The performance results presented in this paper do not include the database distribution times.

Table IV. Cost comparison.

	Amazon Web Services (\$)	Azure (\$)
Compute cost	10.88 (0.68 $\times$ 16 HCXL)	15.36 (0.12 $\times$ 128 Azure Small)
Queue messages ($\sim$ 10, 000)	0.01	0.01
Storage (1 GB, 1 month)	0.14	0.15
Data transfer in/out (1 GB)	0.10 (in)	0.10 (in) + 0.15 (out)
Total cost	11.13	15.77

5.1. Performance with different cloud instance types

Figure 4 and 6 present the benchmark results for BLAST Classic Cloud application on different EC2 instance types. These experiments processed 64 query files, each containing 100 sequences using 16 compute cores. Whereas we expected the memory size to have a strong correlation to the BLAST performance, because of the querying of a large database, the performance results do not show a significant effect related to the memory size, as HCXL instances with less than 1 GB of memory per CPU core were able to perform comparatively to Large and XL instances with 3.75 GB per CPU core. However, it should be noted that there exists a slight correlation with memory size, as the lower clock-rated XL (∼ 2.0 GHz) instances with more memory per core performed similarly to the HCXL (∼ 2.5 GHz) instances. The HM4XL instances (∼ 3.25 GHz) have a higher clock rate, which partially explains the faster processing time. Once again, the EC2 HCXL instances gave the most cost-effective performance, thus offsetting the performance advantages demonstrated by other instance types.

Figure 9 presents the benchmark results for BLAST Classic Cloud application on different Azure instance types. These experiments processed eight query files, each containing 100 sequences using eight small, four medium, two large, and one XL instances, respectively.

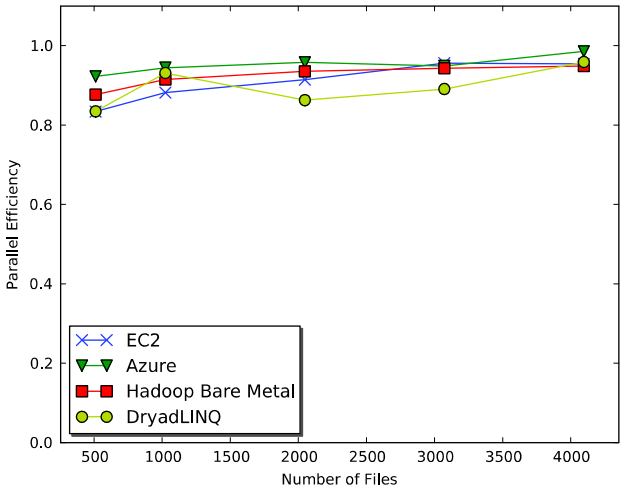

Figure 7. Cap3 parallel efficiency.

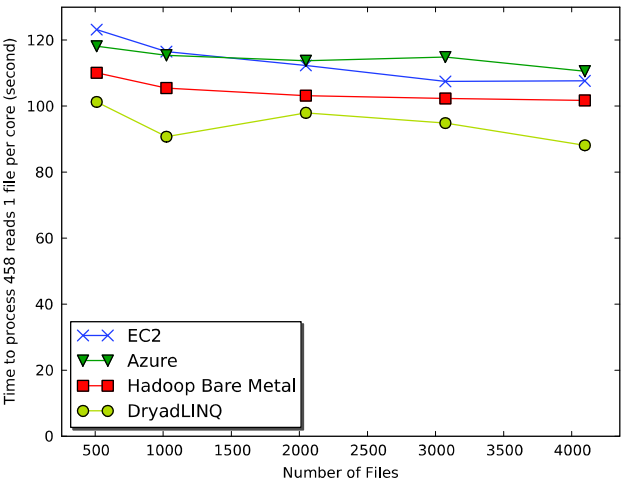

Figure 8. Cap3 execution time for single file per core.

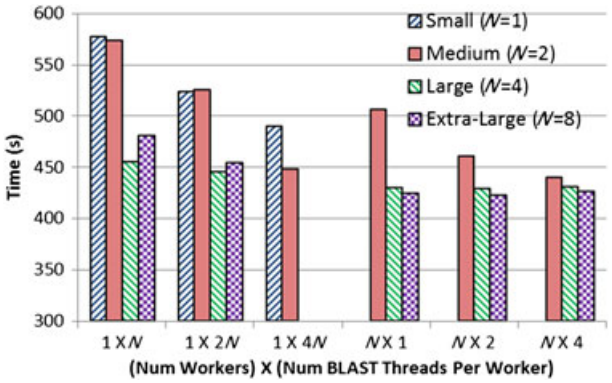

Figure 9. Time to process eight query files using BLAST in Azure.

Although the features of Azure instance types scale linearly, the BLAST application performed better with larger total memory sizes. When sufficient memory is available, BLAST can load and reuse the whole BLAST database ($\sim$ 8 GB) into the memory. BLAST application has the ability to parallelize the computations using threads. The horizontal axis of Figure 9 depicts 'Number of workers (processes) per instance'$\times$'Number of BLAST threads per worker'. 'N' stands for the number of cores per instance in that particular instance type. According to the results, Azure Large and XL instances deliver the best performance for BLAST. Using pure BLAST threads to parallelize inside the instances delivered slightly lesser performance than using multiple workers (processes). The costs to process eight query files are directly proportional to the run time, because of the linear pricing of Azure instance types.

5.2. Scalability

For the scalability experiment, we replicated a query data set of 128 files (with 100 sequences in each file), one to six times to create input data sets for the experiments, ensuring the linear scalability of the workload across data sets. Even though the larger data sets are replicated, the base 128-file data set is inhomogeneous. The Hadoop-BLAST tests were performed on an iDataplex cluster, in which each node had two four-core CPUs (Intel Xeon CPU E5410 2.33 GHz; Intel Corp., Santa Clara, CA, USA) and 16 GB memory and was inter-connected using Gigabit Ethernet. DryadLINQ tests were performed on a Windows HPC cluster with 16 cores (AMD Opteron 2.3 GHz; AMD Inc., Sunnywale, CA, USA) and 16 GB of memory per node. Sixteen HCXL instances were used for the EC2 testing, and 16 Large instances were used for the Azure testing.

Figure 10 depicts the absolute parallel efficiency of the distributed BLAST implementations, whereas Figure 11 depicts the average time to process a single query file in a single core. From those figures, we can conclude that all four implementations exhibit near-linear scalability with comparable performance (within 20% efficiency), whereas BLAST on Windows environments (Azure and DryadLINQ) exhibit the better overall efficiency. The limited memory of the HCXL instances shared across eight workers performing different BLAST computations may have contributed to the relatively low efficiency of EC2 BLAST implementation. According to Figure 6, the use of EC2 HM4XL instances would have given better performance than HCXL instances but at a much higher cost. The amortized cost to process 768 files $\times$ 100 queries in each file using Classic Cloud BLAST was $\sim$ \$10 using EC2 and $\sim$ \$12.50 using Azure.

6. GENERATIVE TOPOGRAPHIC MAPPING INTERPOLATION

Generative Topographic Mapping (GTM) [16] is an algorithm for finding an optimal user-defined low-dimensional representation of high-dimensional data. This process is known as dimension reduction, which plays a key role in scientific data visualization. In a nutshell, GTM is an unsupervised learning method for modeling the density of data and finding a non-linear mapping of

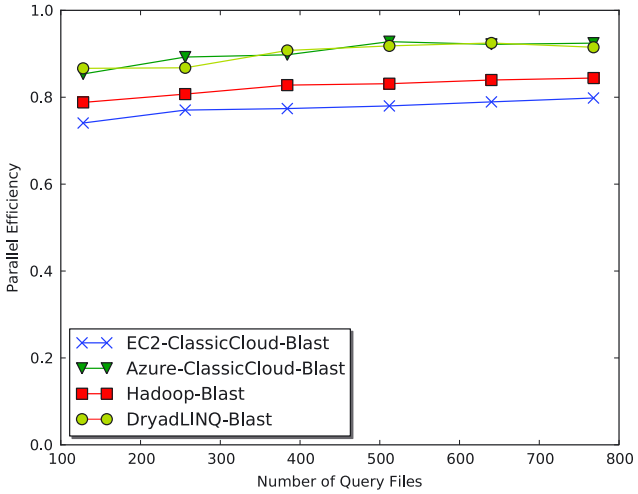

Figure 10. BLAST parallel efficiency.

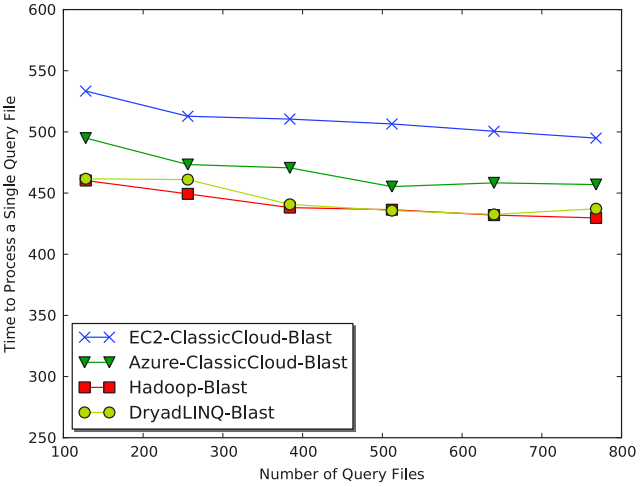

Figure 11. BLAST average time to process a single query file.

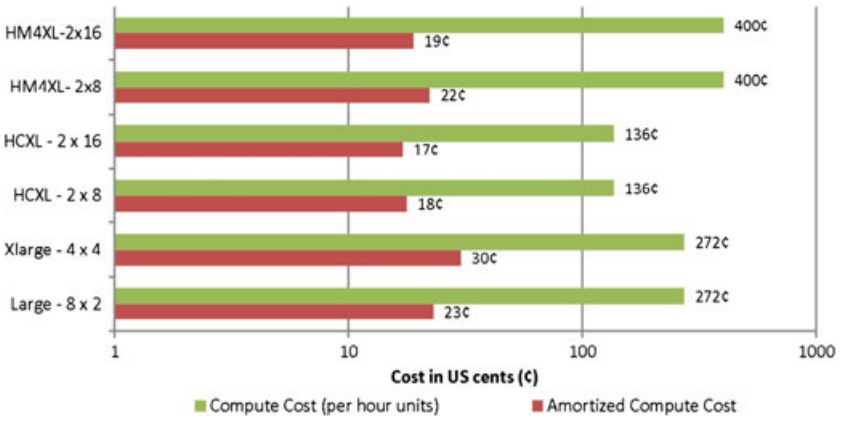

Figure 12. Generative Topographic Mapping cost with different instance types. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large; Xlarge, Extra Large.

high-dimensional data in a low-dimensional space. To reduce the high computational costs and memory requirements in the conventional GTM process for large and high-dimensional data sets, GTM Interpolation [17] has been developed as an out-of-sample extension to process much larger data points with minor trade-off of approximation. GTM Interpolation takes only a part of the full data set, known as samples, for a compute-intensive training process and applies the trained result to the rest of the data set, known as out-of-samples. With this interpolation approach in GTM, one can visualize millions of data points with modest amount of computations and memory requirement.

The size of the input data for the interpolation algorithm consists of millions of data points and usually ranges in gigabytes, whereas the size of the output data in lower dimensions are orders of magnitude smaller than the input data. Input data can be partitioned arbitrarily on the data point boundaries in order to generate computational subtasks. The output data from the subtasks can be collected using a simple merging operation and do not require any special combining functions. The GTM Interpolation application is highly memory intensive and requires a large amount of memory proportional to the size of the input data.

6.1. Application performance with different cloud instance types

As can be seen in Figure 13, we can infer that memory (size and bandwidth) is a bottleneck for the GTM Interpolation application. The GTM Interpolation application performs better in the presence of more memory and a smaller number of processor cores sharing the memory. The HM4XL instances give the best performance overall, but the HCXL instances still appear to be the most economical choice (Figure 12).

6.2. Generative Topographic Mapping Interpolation speedup

We used the PubChem data set of 26 million data points with 166 dimensions to analyze the GTM Interpolation applications. PubChem is a repository of over 60 million chemical molecules, their chemical structures, and their biological activities funded by the National Institutes of Health. A pre-processed subset of 100,000 data points were used as the seed for the GTM Interpolation. We partitioned the input data into 264 files, with each file containing 100,000 data points. Figures 14 and 15 depict the performance of the GTM Interpolation implementations.

DryadLINQ tests were performed on a 16 core (AMD Opteron 2.3 GHz) per node, 16 GB memory per node cluster. Hadoop tests were performed on a 24 core (Intel Xeon 2.4 GHz) per node, 48 GB memory per node cluster that was configured to use only eight cores per node. Classic Cloud Azure tests were performed on Azure Small instances (single core). Classic Cloud EC2 tests were performed on EC2 Large, HCXL as well as on HM4XL instances separately. HM4XL and HCXL

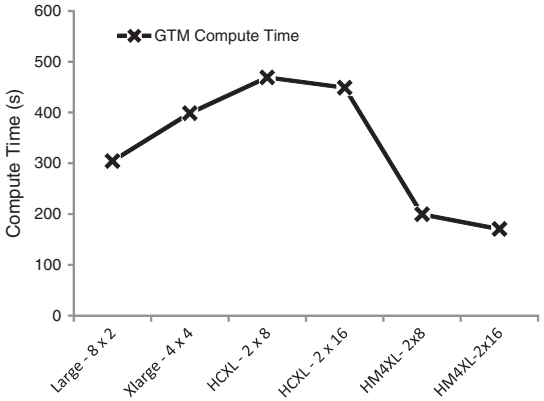

Figure 13. Generative Topographic Mapping (GTM) Interpolation compute time with different instance types. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large; Xlarge, Extra Large.

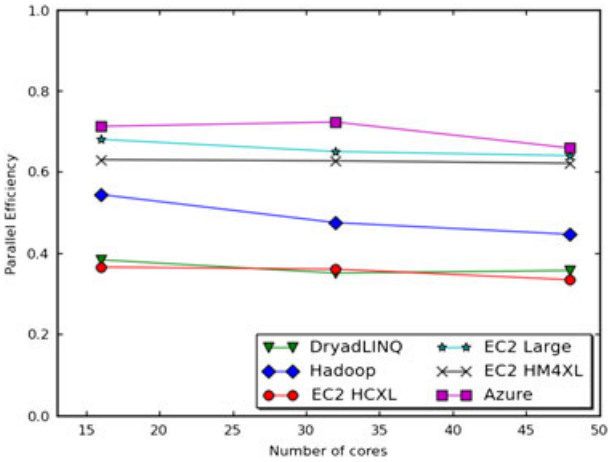

Figure 14. Generative Topographic Mapping Interpolation parallel efficiency. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large.

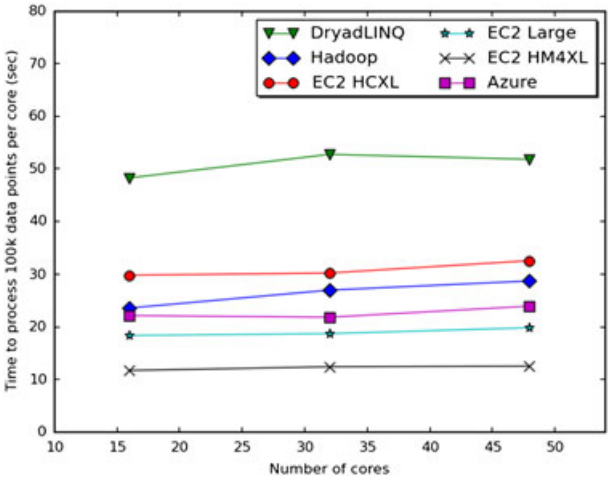

Figure 15. Generative Topographic Mapping Interpolation performance per core. HM4XL, High-Memory Quadruple Extra Large; HCXL, High-CPU Extra Large.

instances were considered eight cores per instance, whereas 'Large' instances were considered two cores per instance.

Characteristics of the GTM Interpolation application are different from the Cap3 application as GTM is more memory intensive and the memory bandwidth becomes the bottleneck, which we assume to be the cause of the lower efficiency numbers. Among the different EC2 instances, Large instances achieved the best parallel efficiency and HM4XL instances gave the best performance, whereas HCXL instances were the most economical. Azure Small instances achieved the overall best efficiency. The efficiency numbers highlight the memory-bound nature of the GTM Interpolation computation, whereas platforms with less memory contention (fewer CPU cores sharing a single memory) performed better. As noted, the DryadLINQ GTM Interpolation efficiency is lower than the others. One reason for the lower efficiency would be the usage of 16 core machines for the computation, which puts more contention on the memory.

The computational tasks of GTM Interpolation applications were much finer grain than those in the Cap3 or BLAST applications. Compressed data splits, which were unzipped before handing over to the executable, were used because of the large size of the input data. When the input data size is larger, Hadoop and DryadLINQ applications have an advantage of data locality-based scheduling over EC2. The Hadoop and DryadLINQ models bring computation to the data optimizing the input/output load, whereas the Classic Cloud model brings data to the computations.

7. RELATED RESEARCH

There exist many studies [18–20] that benchmark existing traditionally distributed scientific applications on the cloud. In contrast, we focused on implementing and analyzing the performance of biomedical applications using cloud services/technologies and cloud-oriented programming frameworks. In one of our earlier studies [13], we analyzed the overhead of virtualization and the effect of inhomogeneous data on the cloud-oriented programming frameworks. Ekanayake and Fox [20] analyzed the overhead of Message Passing Interface (MPI) running on virtual machines under different virtual machine configurations and under different MPI stacks.

In addition to the biomedical applications discussed in this paper, we have also developed distributed pairwise sequence alignment applications using MapReduce programming models [13]. There are other biomedical applications developed using MapReduce programming frameworks, such as CloudBurst [21], which performs parallel genome read mappings. CloudBLAST [22] performs distributed BLAST computations using Hadoop and implements an architecture similar to the Hadoop-BLAST used in this paper. AzureBlast [23] presents a distributed BLAST implementation for Azure Cloud infrastructure developed using Azure Queues, Tables, and Blob Storage with an architecture similar to our Classic Cloud AzureBlast implementation.

CloudMapReduce is an effort to implement a MapReduce framework utilizing the Amazon cloud infrastructure services. AWS [6] also offer MapReduce as an on-demand cloud service through the Elastic MapReduce service. We are currently working to develop a MapReduce framework for Windows Azure, TwisterAzure [12], using Azure cloud infrastructure services, which will also support iterative MapReduce executions

Walker [24] presented a more detailed model for buying versus leasing decisions for CPU power based on lease-or-buy budgeting models, pure CPU hours, Moore's law, and so on. Our cost estimation in Section 4.3 is based on the pure performance of the application in different environments, the purchase cost of the cluster, and the estimate of the maintenance cost. Walker also highlights the advantages of the mobility user's gain through the ability to perform short-term leases from cloud computing environments, allowing them to adopt the latest technology. Wilkening *et al.* [25] presented a cost-based feasibility study for using BLAST in EC2 and concludes that the cost in clouds is slightly higher than the cost of using compute clusters. They benchmarked the BLAST computation directly inside the EC2 instances without using a distributed computing framework and also assumed the local cluster utilization to be 100%.

8. CONCLUSION

We have demonstrated the feasibility of Cloud infrastructures for three loosely coupled scientific computation applications by implementing them using cloud infrastructure services as well as cloud-oriented programming models, such as Hadoop MapReduce and DryadLINQ.

Cloud infrastructure services provide users with scalable, highly available alternatives to their traditional counterparts but without the burden of managing them. Although the use of high-latency, eventually consistent cloud services together with off-instance cloud storage has the potential to cause significant overheads, our work in this paper has shown that it is possible to build efficient, low overhead applications utilizing them. Given sufficiently coarser grain task decompositions, Cloud infrastructure service-based frameworks as well as the MapReduce-based frameworks offered good parallel efficiencies in almost all of the cases we considered. Computing Clouds offer different instance types at different price points. We showed that selecting an instance type that is best suited to the user's specific application can lead to significant time and monetary advantages.

Although models like Classic Cloud bring in operational and quality of services advantages, it should be noted that the simpler programming models of existing cloud-oriented frameworks like MapReduce and DryadLINQ are more convenient for the users. Motivated by the positive results presented in this paper, we are working on developing a fully fledged MapReduce framework with iterative MapReduce support for the Windows Azure Cloud infrastructure using Azure infrastructure services as building blocks, which will provide users the best of both worlds. The cost effectiveness of cloud data centers, combined with the comparable performance reported here, suggests that

loosely coupled science applications will be increasingly implemented on clouds and that using MapReduce frameworks will offer convenient user interfaces with little overhead.

ACKNOWLEDGEMENTS

We would also like extend our gratitude to our collaborators David Wild and Bin Chen. We appreciate Microsoft for their technical support on Dryad and Azure. This work is supported by the National Institutes of Health under grant 5 RC2 HG005806-02. This work was made possible using the compute use grant provided by Amazon Web Service that is titled 'Proof of concepts linking FutureGrid users to AWS'. We would like to thank Joe Rinkovsky, Jenett Tillotson, and Ryan Hartman for their technical support.

REFERENCES

1. Hey T, Tansley S, *et al. Jim Gray on eScience: a transformed scientific method*, Microsoft Research, 2009.
2. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
3. *Apache Hadoop*. Available from: http://hadoop.apache.org/core/ (accessed on 20 March 2010).
4. Yu Y, Isard M, *et al.* DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. *Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, 2008.
5. Huang X, Madan A. CAP3: a DNA sequence assembly program. *Genome Research* 1999; **9**(9):868–77.
6. *Amazon Web Services*, vol. 2010. Available from: http://aws.amazon.com/ (accessed on 20 March 2011).
7. *Windows Azure Platform*. Available from: http://www.microsoft.com/windowsazure/ (accessed on 20 March 2010).
8. Varia J. *Cloud Architectures.* Amazon Web Services. Available from: http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf (accessed on 20 April 2010).
9. Chappell D. *Introducing Windows Azure*. Available from: http://go.microsoft.com/?linkid=9682907 (accessed on December 2009).
10. Isard M, Budiu M, *et al.* Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 2007; 59–72.
11. Grama A, Karypis G, *et al. Introduction to Parallel Computing* (2nd edn). Addison Wesley: Boston, MA, 2003. ISBN 978-0201648652.
12. Gunarathne T, Wu TL, *et al.* MapReduce in the Clouds for Science. *2nd International Conference on Cloud Computing*, Indianapolis, IN, 2010.
13. Ekanayake J, Gunarathne T, *et al*. Cloud Technologies for Bioinformatics Applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2011; **22**(6):998.
14. Camacho C, Coulouris G, *et al.* BLAST+: architecture and applications. *BMC Bioinformatics 2009* 2009; **10**(1):421.
15. NCBI. BLAST. National Center for Biotechnology Information. Available from: http://blast.ncbi.nlm.nih.gov (accessed on 20 September 2010).
16. Choi JY. Deterministic Annealing for Generative Topographic Mapping GTM. *Technical Report*, Indiana University, Bloomington, IN, 2 September 2009.
17. Bae S-H, Choi JY, *et al.* Dimension reduction and visualization of large high-dimensional data via interpolation. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, Chicago, IL, 2010; 203–214.
18. Walker E. Benchmarking Amazon EC2 for high-performance scientific computing. *;login: The USENIX Magazine* 2008; **33**(5):18–23.
19. Evangelinos C, Hill CN. Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. *Cloud Computing and Its Applications (CCA-08)*, Chicago, IL, 2008.
20. Ekanayake J, Fox G. High Performance Parallel Computing with Clouds and Cloud Technologies. *First International Conference CloudComp on Cloud Computing*, Munich, Germany, 2009.
21. Schatz MC. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics* 2009; **25**(11):1363–1369.
22. Matsunaga A, Tsugawa M, *et al.* CloudBLAST: combining MapReduce and virtualization on distributed resources for bioinformatics applications. *Proceedings of the IEEE 4th International Conference on eScience*, Indianapolis, IN, 7–12 December 2008; 222–229.
23. Lu W, Jackson J. *et al.* AzureBlast: a case study of developing science applications on the Cloud. *ScienceCloud: 1st Workshop on Scientific Cloud Computing co-located with HPDC 2010 (High Performance Distributed Computing)*, Chicago, IL, 2010.
24. Walker E. The real cost of a CPU hour. *Computer* 2009; **42**(4):35–41.
25. Wilkening J, Wilke A. *et al.* Using clouds for metagenomics: a case study. *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, 31 August; 1–6.