Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications

Thilina Gunarathne, Tak-Lon Wu, Jong Youl Choi, Seung-Hee Bae, Judy Qiu
School of Informatics and Computing / Pervasive Technology Institute
Indiana University, Bloomington.
{tgunarat, taklwu, jychoi, sebae, xqiu}@indiana.edu

Abstract

Cloud computing offers exciting new approaches for scientific computing that leverages the hardware and software investments on large scale data centers by major commercial players. Loosely coupled problems are very important in many scientific fields and are on the rise with the ongoing move towards data intensive computing. There exist several approaches to leverage clouds & cloud oriented data processing frameworks to perform pleasingly parallel computations. In this paper we present three pleasingly parallel biomedical applications, 1) assembly of genome fragments 2) sequence alignment and similarity search 3) dimension reduction in the analysis of chemical structures, implemented utilizing cloud infrastructure service based utility computing models of Amazon Web Services and Microsoft Windows Azure as well as utilizing MapReduce based data processing frameworks, Apache Hadoop and Microsoft DryadLINQ. We review and compare each of the frameworks and perform a comparative study among them based on performance, efficiency, cost and the usability. Cloud service based utility computing model and the managed parallelism (MapReduce) exhibited comparable performance and efficiencies for the applications we considered. We analyze the variations in cost between the different platform choices (eg: EC2 instance types), highlighting the need to select the appropriate platform based on the nature of the computation.

1. Introduction

Scientists are overwhelmed with the increasing amount of data processing needs arising from the storm of data that is flowing through virtually every field of science. One example is the production of DNA fragments at massive rates by the now widely available automated DNA Sequencer machines. Another example would be the data generated by the Large Hadron Collider. Preprocessing, processing and analyzing these large amounts of data is a unique very challenging problem, yet opens up many opportunities for computational as well as computer scientists. According to Jim Gray, increasingly the scientific breakthroughs will be powered by computing capabilities that support researchers to analyze massive data sets. He aptly named data intensive scientific discovery as the forth science 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 simply by use of a credit card. The utility computing model offered through those cloud computing offerings opens up exciting new opportunities for the computational scientists to perform their computations since such a model suits well for the occasional resource intensive staccato compute needs of the scientists. Another interesting feature for scientists is the ability to increase the throughput of their computations by horizontally scaling the compute resources without incurring additional cost overhead. For an example in a utility computing model, 100 hours of 10 compute nodes cost same as 10 hours in 100 compute nodes. This is facilitated by the virtually unlimited resource availability of cloud computing infrastructures backed by the world's largest data centers owned by the major commercial players

such as Amazon, Google & Microsoft. We expect the economies of scale enjoyed by the cloud providers scale would translate to cost efficiencies for the users.

In addition to the leasing of virtualized compute nodes, 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 notice the emergence of different cloud oriented data processing technologies and frameworks. One example would be the Map Reduce [2] framework, which 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 parallel data processing frameworks which supports Map Reduce type computations.

A pleasingly parallel (also called embarrassingly parallel) application is an application which can be parallelized requiring minimal effort to divide the application in to independent parallel parts, each of which have no or very minimal data, synchronization or ordering dependencies among each other. These applications are good candidates for commodity compute clusters with no specialized interconnects. There are many scientific applications that fall in to this category. Few examples of pleasingly parallel applications would be Monte Carlo simulations, BLAST searches, many image processing applications such as ray tracing, parametric studies. Most of the data cleansing and pre-processing applications can also be classified as pleasingly parallel applications. The relative number of pleasingly parallel scientific workloads has been growing recently due to the emerging data intensive computational fields such as bioinformatics.

In this paper we introduce a set of abstract frameworks constructed using the cloud oriented programming models to perform pleasingly parallel computations. We present implementations of bio medical applications such as Cap3 [5] sequence assembly, BLAST sequence search and GTM interpolation using these frameworks. We analyze the performance and the 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 use Apache Hadoop [3] Map Reduce and Microsoft DryaLINQ [4] as the distributed parallel computing frameworks.

2. Cloud technologies and application architecture

Processing of large data sets using existing sequential executables is a common use case we encounter in many scientific applications. Some of these applications exhibit pleasingly parallel characteristics where the data can be independently processed in parts allowing the applications to be easily parallelized. In the following sections we explore cloud programming models and the application frameworks we developed using them to perform pleasingly parallel computations. These frameworks have been used to implement the applications mentioned in section 3.

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 compute 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 capability to lease hourly billed Xen based virtual machine instances allowing 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 infrastructure as a service approach where it provides users with ‘root’ access to the virtual machines

giving maximum possible flexibility. Users can store virtual machines snapshots as Amazon Machine Images (AMIs), which can be used as templates to create new instances. Amazon EC2 offers a variety of hourly billed instance sizes with different price points giving a richer set of options for the users to choose from depending on thier requirements. One particular instance type of interest is the High-CPU-Extra-Large instances, which costs the same as an Extra-Large instance but offers more CPU power, but with less memory. Similarly EC2 offers High-Memory instance types which offer larger memory sizes, but fewer CPU cores. Table 1 provides a summary of the EC2 instance types we used in this paper. The clock speed of a single EC2 compute unit is approximately 1 GHz to 1.2 GHz. The small instance type with a single EC2 compute unit is only available in 32-bit x86 environment, while the larger instance types support 64 bit x86_64 environment as well.

SQS is an eventual consistent, reliable, scalable and distributed web-scale message queue service ideal for small short-lived transient messages. SQS provides a REST based web service interface enabling any HTTP capable client to use it. Users can create unlimited number of queues and send unlimited number of messages. SQS does not guarantee the order of the messages, the deletion of messages and availability of all the messages for a request, though it guarantees the eventual availability over multiple requests. Each message has a configurable visibility timeout. Once it's read by a client, the message will be hidden from other clients till the visibility time expires. Message will reappear upon expiration of the timeout, as long as it is not deleted. The service is priced based on the number of API requests as well as based on the total amount of data transfer per month.

S3 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 in to buckets. S3 pricing is based on the size of the stored data, amount of data transferred and the number of API requests.

Table 1 : Selected EC2 instance types

Instance Type	Memory	EC2 compute units	Actual CPU cores	Cost per hour
Large (L)	7.5 GB	4	2 X (~2Ghz)	0.34$
Extra Large (XL)	15 GB	8	4 X (~2Ghz)	0.68$
High CPU Extra Large (HCXL)	7 GB	20	8 X (~2.5Ghz)	0.68$
High Memory 4XL (HM4XL)	68.4 GB	26	8 X (~3.25Ghz)	2.00$

Table 2 : Microsoft Windows Azure instance types

Instance Type	CPU Cores	Memory	Local Disk Space	Cost per hour
Small	1	1.7 GB	250 GB	0.12$
Medium	2	3.5 GB	500 GB	0.24$
Large	4	7 GB	1000 GB	0.48$
Extra Large	8	15 GB	2000 GB	0.96$

2.1.2. **Microsoft Azure Platform.** Microsoft Azure platform [7] is a cloud computing platform offering a set of cloud computing services similar to the Amazon Web Services. Windows Azure compute, Azure Storage Queues and Azure Storage blob services are the Azure counterparts for Amazon EC2, Amazon SQS and the Amazon S3 services. Features of the Azure services are more or less similar to the features of the AWS services we discussed above, except for the following.

Windows Azure Compute only supports Windows virtual machine instances and offers a limited variety of instance types when compared with Amazon EC2. As shown in Table 2, Azure instance type configurations and the cost scales up linearly from small, medium, large to extra large. Azure instances are available in 64 bit environment. It's been speculated that the clock speed of a single CPU core in Azure terminology 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 comparable to a single Amazon high CPU extra large instance with 20 EC2 compute units. Azure Compute follows a platform as a service approach and offers the .net runtime as the platform. Users can deploy their programs as an Azure deployment package through a web application. Users do not have the ability to interact with the Azure instances, other than through the deployed programs.

2.1.3. Classic cloud processing model

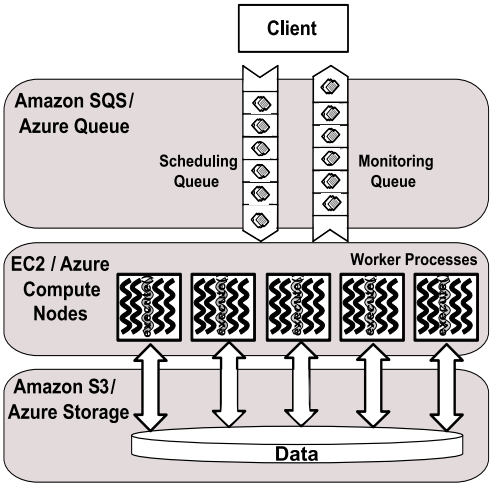

Figure 1: Classic cloud processing model

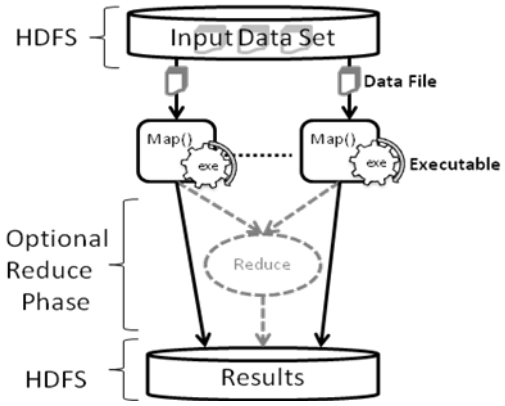

Figure 2: Hadoop MapReduce processing model

Figure 1 depicts the architecture of the classic cloud processing model. Varia [8] and Chappell [9] describe 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 SQS and Azure queue, where a message will not be visible to other workers for the given amount of time once a worker reads it and reappears after the timeout, 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. 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 a user can configure the workers to use any executable program installed 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, but for more sophistication one can use cloud data stores like Amazon SimpleDB to store the monitoring and state data. 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. Even though it might not be the best option due to the data being stored in the cloud, one can start workers in computers outside of the cloud to help perform the computations.

2.2. Apache Hadoop MapReduce

Apache Hadoop [3] is an open source implementation of the Google MapReduce [2] technology and shares many characteristics with the Google MapReduce implementation. Apache Hadoop MapReduce uses HDFS distributed parallel file system for data storage, which stores the data across

the local disks of the compute nodes while presenting a single file system view through the HDFS API. HDFS is targeted for deployment on commodity clusters and achieves reliability through replication of file data. When executing using the stored in HDFS, Hadoop optimizes the data communication by scheduling computations near the data using the data locality information provided by the HDFS file system. 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. The Map Reduce model reduces the data transfer overheads by overlapping data communication with computation when reduce steps are involved. Hadoop performs duplicate execution of slower tasks and handles failures by rerunning of the failed tasks using different workers.

As shown in figure Figure 2, the pleasingly parallel application framework on Hadoop is developed as a set of map tasks which process the given data splits (files) using the configured executable program. Input to a Hadoop 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 which will provice 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 data-flow graphs (DAG), 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 Map Reduce frameworks, the Dryad scheduler optimizes the data transfer overheads by scheduling the computations near data and handles failures through rerunning of tasks and duplicate instance execution. 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 API. DryadLINQ [4] is a high level declarative language layer on top of Dryad. DryadLINQ queries get translated in to distributed Dryad computational graphs in the run time. Latest version of DryadLINQ operates only on Window HPC clusters.

The DryadLINQ implementation of the framework uses the DryadLINQ “select” operator on the data partitions to perform the distributed computation. The resulting computation graph looks much similar to the figure 2, where instead of using HDFS, Dryad will use the windows shared local directories for data storage. Data partitioning, distribution and the generation of metadata files for the data partitions is implemented as part of our pleasingly parallel framework.

2.4. Usability of the technologies

As expected, implementing the above mentioned application framework using already existing Hadoop and DryadLINQ data processing frameworks was easier than implementing them using cloud services as building blocks. Hadoop and DryadLINQ take care of scheduling, monitoring and fault tolerance. With Hadoop we had to implement a Map function, which contained the logic to copy the input file from HDFS to the working directory, execute the external program as a process and to upload the results files to the HDFS. In addition to this, we had to implement a custom InputFormat and a RecordReader to support file inputs to the map tasks. With DryadLINQ we had implement a

side effect free function to execute the program on the given data and to copy the result to the output shared directory. But significant effort had to be spent on implementing the data partitioning and the distribution programs to support DryadLINQ.

Table 3: Summary of cloud technology features

	AWS/ Azure	Hadoop	DryadLINQ
Programming patterns	Independent job execution, More structure can be imposed using client side driver program.	Map Reduce	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 & Communication	S3/Azure Storage. Data retrieved through HTTP.	HDFS parallel file system. TCP/IP based Communication	Local files
Environment	EC2/Azure virtual instances, local compute resources	Linux cluster, Amazon Elastic MapReduce	Windows HPCS cluster
Scheduling & 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

EC2 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 the scratch using the features of the cloud services. Amazon EC2 provides infrastructure as a service by allowing users to access the raw virtual machine instances while windows Azure provides the .net platform as a service allowing users to deploy .net applications in the virtual machines through a web interface. Hence the deployment process was easier with Azure as oppose to the EC2 where we had to manually create instances, install software and start the worker instances. On the other hand the EC2 infrastructure as a service gives more flexibility and control to the developers. Azure SDK provides better development and testing support through the 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 the Azure applications. Azure platform is heading towards providing a more developer friendly environment, but as of today (Oct 2010) the Azure platform is less matured compared to the AWS, with deployment glitches and with the non-deterministic times taken for the deployment process.

3. Performance Methodology

In the performance studies we use parallel efficiency as the measure to evaluate the different frameworks. Parallel efficiency is a relatively good measure to evaluate the different approaches we use in our studies as we don't have the possibility to use identical configurations across the different environments. At the same time we cannot use efficiency to directly compare the different technologies. Even though efficiency accounts the system dissimilarities which affect the sequential running time as well as the parallel running time, it does not reflect other dissimilarities such as memory size, memory bandwidth and network bandwidth that can affect when running parallel computations. Parallel efficiency is calculated using the following formula.

$$Efficiency\ (Ep) = \frac{\mathrm{T}(1)}{\mathrm{pT}(\rho)}$$

T(1) is the best sequential execution time for the application in a particular environment using the same data set or a representative subset. In all the cases the sequential time was measured with no data transfers with input files already present in the local disks. T(ρ) is the parallel run time for the application while "p" is the number of processor cores used.

Per core per computation time is calculated in each test to give an idea about the actual execution times in the different environments.

$$Per\ Computation\ Per\ Core\ time = \frac{pT(\rho)}{No.of\ computations}$$

When composing results for this paper, we considered a single EC2 Extra-Large instance, with 20 EC2 compute units, as 8 actual CPU cores while an Azure small instance was considered as a single CPU core. In all the test cases, it is assumed that the data was already present in the frameworks preferred storage location. We used Apache Hadoop version 0.20.2 and DryadLINQ version 1.0.1411.2 (November 2009) for our studies.

3.1. Performance of different EC2 instance types

Due to the richness of the instance type choices Amazon EC2 provides, it is important to select an instance type which optimizes the balance between performance and cost. We perform an instance type study for each of our applications using the EC2 instance types mentioned in Table 1 using 16 CPU cores for each study. We do not perform such studies for Azure as features of the Azure instance types scale linearly with the price as shown in table 2. EC2 small instances were not included in our study as they do not support 64 bit operating systems.

Cloud virtual machine instances are hourly billed. When presenting the results, compute cost (hour units) assumes that particular instances are used only for that particular computation and no useful work is done for the remainder of the hour, effectively making the computation responsible for the whole 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 it got executed. The horizontal axis labeling of the graphs are in the format 'Instance Type' – 'Number of Instances' X 'Number of Workers per Instance'. For an example, HCXL – 2 X 8 means two High-CPU-Extra-Large instances are used with 8 workers per instance.

4. Cap3

Cap3 [5] is a sequence assembly program which 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 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 is generating massive amounts of sequencing data that needs to be assembled. Cap3 program is often used in parallel with lots of input files due to the pleasingly parallel nature of the application. The run time of the Cap3 application depends on the contents of the input file. Cap3 is relatively not memory intensive compared to the interpolation algorithms we discuss below. Size of a typical data input file for 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 EC2 cloud instance types

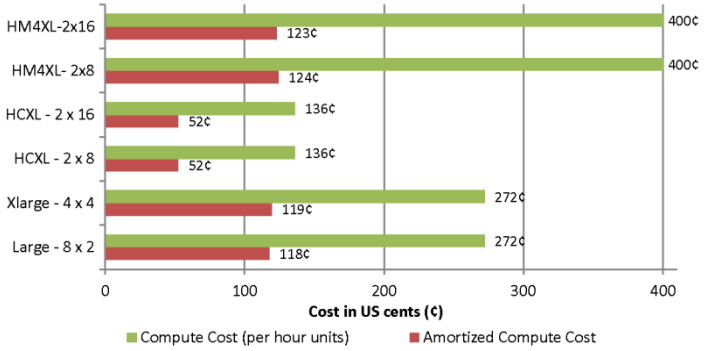

Figure 3 : Cap3 cost with different EC2 instance types

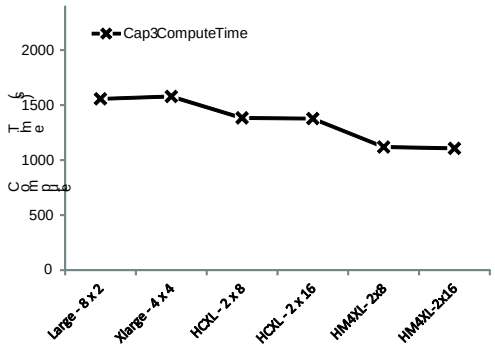

Figure 4 : Cap3 compute time with different instance types

Figure 3 and Figure 4 present the benchmark results for 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 the performance depends primarily on the computational power. While EC2 High-Memory-Quadruple-Extra-Large instances show the best performance due to the higher clock rated processors, the most cost effective performance for the Cap3 EC2 classic cloud application is gained using the EC2 High-CPU-Extra-Large instances.

4.2. Scalability study

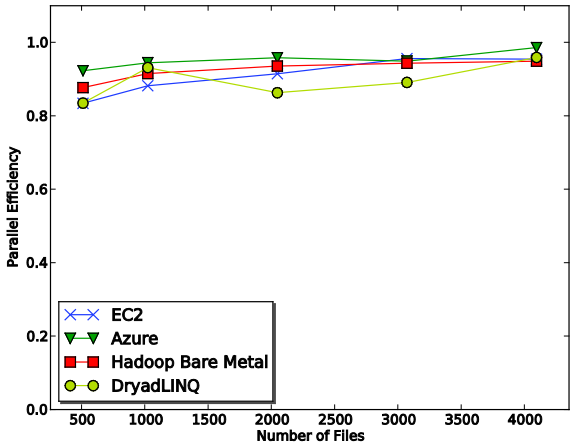

Figure 5 : Cap3 parallel efficiency

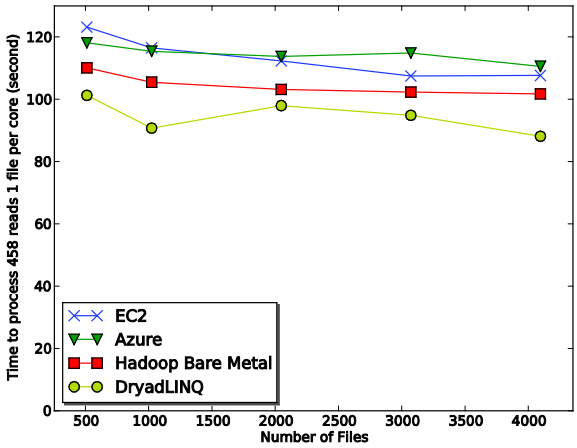

Figure 6 : Cap3 execution time for single file per core

We benchmarked the Cap3 classic cloud implementation performance using a replicated set of FASTA formatted data files, each file containing 458 reads, and compared with our previous performance results [11] for Cap3 DryadLINQ and Cap3 Hadoop. 16 High-CPU-Extra-Large instances were used for the EC2 testing and 128 small Azure instances were used for the Azure Cap3 testing. DryadLINQ and Hadoop bare metal results were obtained using a 32 node X 8 core (2.5 Ghz) cluster with 16 GB memory on each node.

Load balancing across the different sub tasks do not pose a significant overhead in the Cap3 performance studies, as we used a replicated set of input data files making each sub task identical. We performed a detailed study of the performance of Hadoop and DryadLINQ in the face of inhomogeneous data in one of our previous works [11], where we noticed better natural load balancing in Hadoop due to its dynamic global level scheduling than in DryadLINQ, which uses static

task partitioning. We assume cloud frameworks will be able perform load balancing similar to Hadoop as they share the same dynamic scheduling global queue architecture.

Based on figure 5 & 6 we can conclude that all four implementations exhibit similar (within 20%) reasonable efficiency with low parallelization overheads. When interpreting figure 6, it should be noted that the Cap3 program performs ~12.5% faster on windows environment than on the Linux environment. As we 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

Table 4 : Cost Comparison

	Amazon Web Services	Azure
Compute Cost	10.88 $ (0.68$ X 16 HCXL)	15.36$ (0.12$ X 128 Azure Small)
Queue messages (~10,000)	0.01 $	0.01 $
Storage (1GB, 1 month)	0.15 $	0.15 $
Data transfer in/out (1 GB)	0.15 $	0.25 $ (0.10$ + 0.15$)
Total Cost	11.19 $	15.77 $

Below we estimate the cost to assemble 4096 FASTA files using classic computing implementations of EC2 and on Azure. For the sake of comparison, we also approximate the cost for 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 (~500,000$) depreciated over 3 years in addition to the yearly maintenance fee (~150,000$), which includes power and administration costs. Application executed in 58 minutes on EC2, in 59 minutes on Azure and in 10.9 minutes on the internal cluster. 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 simplicity, we did not consider other factors such as the opportunity costs of the upfront investment, equipment failures and the upgradability. Also there will be additional costs in the cloud environments for the instance time required for environment preparation and minor miscellaneous platform specific charges such as number of storage requests.

5. BLAST

NCBI BLAST+ [12] is a very popular bioinformatics application that is used to handle sequence similarity searching. It is the latest version of BLAST [13], a multi-letter 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 sub-set of a real-world protein sequence data set as the input BLAST queries and used NCBI's non-redundent (NR) protein sequence database (8.7 GB), updated on 6/23/2010, as the BLAST database. In order to make the tasks coarser granular, we bundled 100 queries in to 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 for Apache Hadoop using the framewroks we presented in section 2. All the implementations download the BLAST database to a local disk partition of each worker prior to start 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, where 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.

5.1. Performance with different EC2 cloud instance types

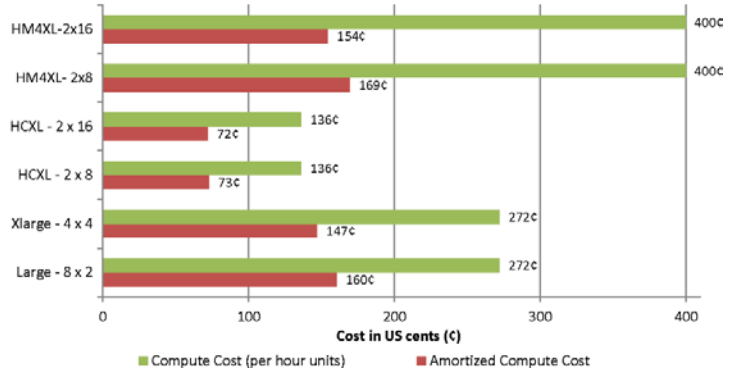

Figure 7 : Cost to process 64 query files using Blast in EC2

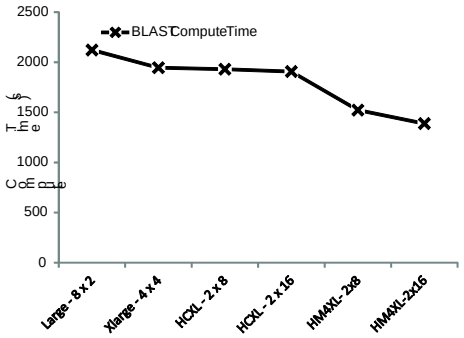

Figure 8 : Time to process 64 query files using Blast in EC2

Figure 7 and Figure 8 present the benchmark results for Blast classic cloud application on different EC2 instance types. These experiments processed 64 query files, each containing 100 seqeunces using 16 compute cores. While we expected the memory size to have a strong correlation to the BLAST performance, due to querying of a large database, the performance results do not show a significant effect with the memory size, as High-CPU-Extra-Large (HCXL) instances with less than 1GB memory per CPU core was able to perform comparatively to Large and Extra-Large instances with 3.75GB per CPU core. However it should be noted that there exist a slight correlation to the memory size, as the lower clock rated Extra-Large (~2.0Ghz) instances, but with more memory per core, performed similar to the HCXL (~2.5Ghz) instances. The High-Memory-Quadruple-Large (HM4XL) instances (~3.25Ghz) have a higher clock rate, which partially explains the faster processing time. Once again EC2 HCXL instances gave the most cost-effective performance offsetting the performance advantages by other instance types.

5.2. Scalability

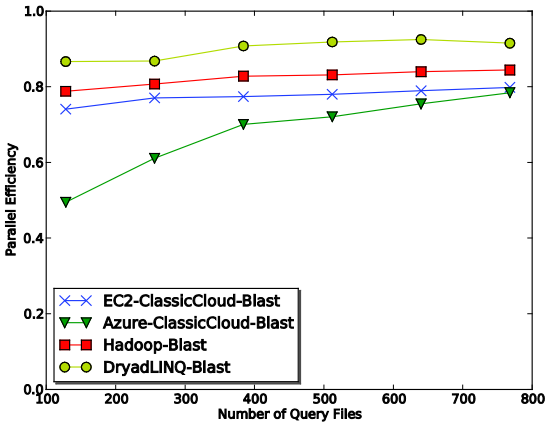

Figure 9 : Blast Parallel efficiency

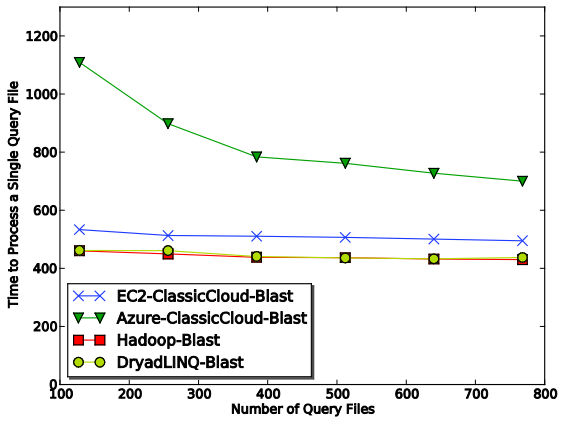

Figure 10 : Blast average time to process a single query file

For the scalability test, we replicated the query data set of 128-files (with 100 sequences in each), one to six times to create the input data sets for the experiments ensuring the linear scalability of the workload across them. 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 4-core CPUs (Intel Xeon CPU E5410 2.33GHz) 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) and 16GB memory per node. 16 High-CPU-Extra-Large instances were used for the EC2 testing and 128 small Azure instances were used for the Azure testing.

Figure 9 depicts the absolute parallel efficiency of the distributed BLAST implementations, while figure 10 depicts the average time to process a single query file in a single core. From those figures we can conclude that DryadLINQ, Hadoop and EC2 classic cloud BLAST implementations exhibit near linear scalability with comparable performance (within 20% efficiency), while DryadLINQ-BLAST exhibit the best performance. Limited memory of the High-CPU-Extra-Large instances shared across 8 workers performing different BLAST computations might have contributed to the relative low efficiency of EC2 BLAST implementation.

Azure classic cloud BLAST implementation exhibited an unusual behavior, where it showed a large performance overhead in the smallest test case and then scaled super linearly. After performing more in-depth experiments, we noticed a large variation of the execution time for individual BLAST tasks. While the variations of execution times are expected due to the inhomogeneous nature of the data, tasks executed by other frameworks exhibited variations only in the range of +/-10% of the average execution time. For the Azure the variations were as high as +/-50%, which made the execution time of the smallest test case (which only had one wave of tasks when scheduling 128 tasks on 128 CPU cores) equivalent to the execution time of the slowest task. In the subsequent test cases, which have more than one wave of tasks, the effect of in-homogeneity gets reduced due to the natural load balancing nature of the global queue based dynamic scheduling of tasks [11].

Further explorations also showed that the slow executing tasks randomly vary from one test run to another, which made us suspect that this behavior is related to an infrastructure limitation. We are planning on further exploring the reason for the slowness of random tasks on Azure. BLAST application also has the ability to parallelize queries in multi-core machines. We are planning on utilizing that feature to create a hybrid framework, where the classic cloud model will parallelize tasks across nodes and the BLAST multicore implementation will parallelize inside the node. We plan on using the resulting framework with Azure large instances to experiment whether it'll resolve the above issue.

6. Generative Topographic Mapping Interpolation

Generative Topographic Mapping **(**GTM)[14] 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 high-dimensional data in a low-dimensional space. Unlike the Kohonen's Self-Organizing Map (SOM) [15] which does not have any density model, GTM defines an explicit density model based on Gaussian distribution [16] and finds the best set of parameters associated with Gaussian mixtures by using an Expectation-Maximization (EM) optimization algorithm[17].

To reduce the high computational costs and memory requirements in the conventional GTM process for large and high-dimesional datasets, GTM Interpolation [18] has been developed as an out-of-sample extension to process much larger data points with minor trade-off of approximation. Instead of processing full dataset approach, GTM Interpolation takes only a part of the full dataset, known as samples, for a computer-intensive training process and applies the trained result to the rest of the dataset, known as out-of-samples, which is usually faster than the former process. With this interpolation approach in GTM, one can visualize millions of data points with modest amount of

computations and memory requirement. Currently we use GTM and GTM interpolation applications for DNA sequence studies and cheminformatics data mining & exploration for the analysis of large chemical compounds in the PubChem database.

The size of the input data for the interpolation algorithms consisting of millions of data points usually ranges in gigabytes, while the size of the output data in lower dimensions is orders of magnitude smaller than the input data. The input data can be partitioned arbitrarily on the data point boundaries to generate computational sub tasks. The output data from the sub tasks can be collected using a simple merging operation and does not require any special combining functions. The GTM interpolation application is high memory intensive and requires large amount of memory proportional to the size of the input data.

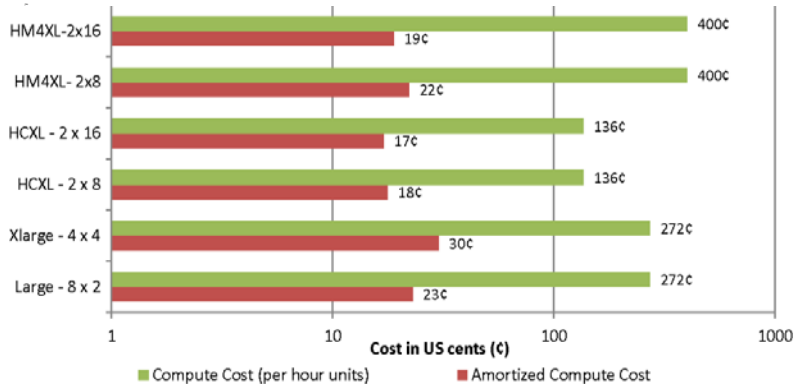

Figure 11 : GTM cost with different instance types

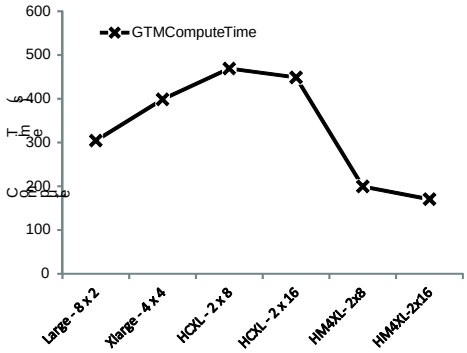

Figure 12 : GTM interpolation compute time with different instance types

6.1. Application performance with different cloud instance types

According to the figure Figure 12 : GTM interpolation compute time with different instance types, we can infer that memory (size & bandwidth) is a bottleneck for the GTM interpolation application. The GTM interpolation application performs better in the presence of more memory and less number of processor cores sharing the memory. The high memory quadruple extra large instances give the best performance, but still the high CPU extra large instances appear as the most economical choice.

6.2. GTM interpolation scalability

We used the PubChem data set of 26.4 million data points with 166 dimensions to analyze the GTM interpolation applications. PubChem is a NIH funded repository of over 60 million chemical molecules including their chemical structures and biological activities. We used a 100,000 already processed subset of the data as a seed for the GTM interpolation. We partitioned the input data in to 264 files with each file containing 100,000 data points. Figure 8 and 9 depicts the performance of the GTM interpolation implementations.

DryadLINQ Cap3 tests were performed on a 16 core (AMD Opteron 2.3 Ghz) per node, 16GB memory per node cluster. Hadoop Cap3 tests were performed on a 24 core (Intel Xeon 2.4 Ghz) per node, 48 GB memory per node cluster which was configured to use only 8 cores per node. Classic cloud Azure tests we performed on Azure small instances where a single instance is considered as a single core in the figure 10. Classic cloud EC2 tests were performed on EC2 Large, High-CPU-Extra-Large (HCXL) as well as on High-Memory-Quadruple-Extra-Large (HM4XL) instances separately. HM4XL and HCXL instances were considered 8 cores per instance while ‘Large’ instances were considered 2 cores per instance.

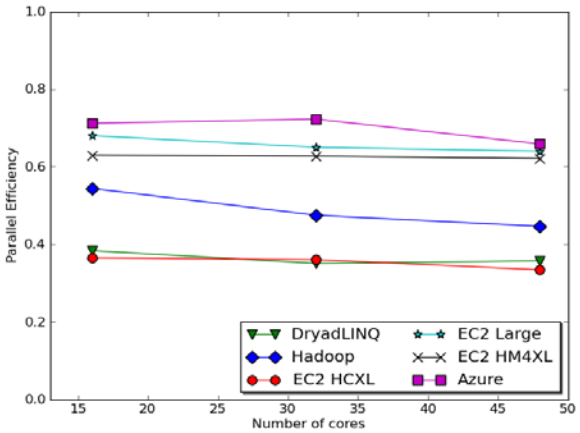

Figure 13: GTM interpolation parallel efficiency

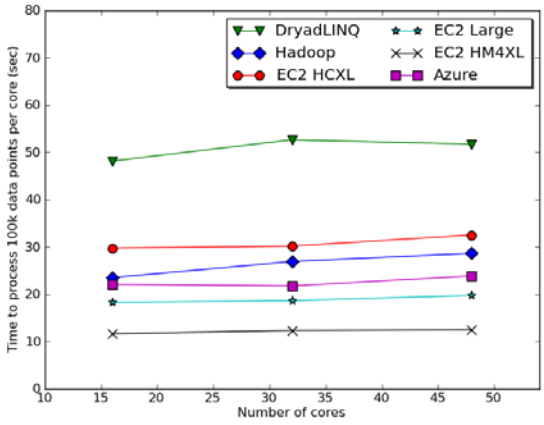

Figure 14 : GTM interpolation performance per core

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 as the cause of the lower efficiency numbers. Among the EC2 different instances, large instances achieved the best parallel efficiency and High-Memory-Quaraple-Extra-Large instances gave the best performance while High-CPU-Extra-Large 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, where platforms with less memory contention (less CPU cores sharing a single memory) performed better. As we can notice, 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.

Computational tasks of GTM applications were much finer grain than in Cap3 or MDS interpolation. Compressed data splits, which were unzipped before handing over to the executable, were used due to the large size of the input data. When the input data size is larger, Hadoop & DryadLINQ applications have an advantage of data locality based scheduling over EC2. Hadoop and DryadLINQ model brings computation to the data optimizing the I/O load, while the classic cloud model brings data to the computations.

7. Related works

There exist many studies [19-21] of benchmarking existing traditional 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 works [11] we analyzed the overhead of virtualization and the effect of inhomogeneous data on the cloud oriented programming frameworks. Also Ekanayake and Fox [22] analyzed the overhead of MPI running on virtual machines under different VM configurations and under different MPI stacks.

In addition to the biomedical applications we have discussed in this paper, we also developed distributed pair-wise sequence alignment applications using the Map Reduce programming models [11]. There are other bio-medical applications developed using Map Reduce programming frameworks such as CloudBurst[23], which performs parallel genome read mappings. CloudBLAST[23] performs distributed BLAST computations using Hadoop and implements an architecture similar to the Hadoop-Blast used in this paper. AzureBlast [24] presents a distributed BLAST implementation similar to the BLAST implementation we implemented using our classic cloud model.

CloudMapReduce[25] is an effort to implement a map reduce framework utilizing the Amazon cloud infrastructure services. Amazon Web Services [6] also offer MapReduce as an one demand cloud service through the Elastic Map Reduce service. We are developing a MapReduce framework for Windows Azure, AzureMapReduce [26], using Azure cloud infrastructure services.

Walker [27] presents 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, etc,. Our cost estimation in 4.3 is based on pure performance of the application in different environments, purchase cost of the cluster and the estimation of maintenance cost. Walker also highlights the advantages of the mobility user's gain through the ability to perform short-term leases from clouds computing environments, allowing them to adopt the latest technology. Wilkening et al[28] presents a cost based feasibility study for using BLAST in EC2 and concludes the cost in clouds is slightly higher than using compute clusters. They benchmarked the BLAST computation directly inside the EC2 instances without using a distributed computing framework and also assume the local cluster utilization to be 100%.

8. Conclusion

We have demonstrated that clouds offer attractive computing paradigms for three loosely coupled scientific computation applications. Cloud infrastructure based models as well as the MapReduce based frameworks offered good parallel efficiencies in most of the cases, given sufficiently coarser grain task decompositions. The higher level MapReduce paradigm offered a simpler programming model. Also by using three different kinds of applications we showed that selecting an instance type which suits your application can give significant time and monetary advantages. Our previous work has tackled a broader range of data intensive applications under MapReduce and also compared them to MPI on raw hardware. The cost effectiveness of cloud data centers combined with the comparable performance reported here suggests that loosely coupled science applications will increasingly be implemented on clouds and that using MapReduce frameworks will offer convenient user interfaces with little overhead.

Acknowledgment

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 which 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.

9. References

[1] T. Hey, S. Tansley, and K. Tolle, *Jim Gray on eScience: a transformed scientific method*: Microsoft Research, 2009.

[2] J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM,* vol. 51, no. 1, pp. 107-113, 2008.

[3] *Apache Hadoop,* Retrieved April 20, 2010, from ASF: http://hadoop.apache.org/core/.

[4] Y. Yu, M. Isard, D. Fetterly *et al.*, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in Symposium on Operating System Design and Implementation (OSDI), San Diego, CA, 2008.

[5] X. Huang, and A. Madan, "CAP3: A DNA sequence assembly program.," *Genome Res,* vol. *9*, no. *9*, pp. 868-77, 1999.

[6] *Amazon Web Services*, vol. 2010, Retrieved April 20, 2010, from Amazon: http://aws.amazon.com/.
[7] *Windows Azure Platform*, Retrieved April 20, 2010, from Microsoft: http://www.microsoft.com/windowsazure/.
[8] J. Varia, *Cloud Architectures*, Amazon Web Services. Retrieved April 20, 2010 : http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf.
[9] D. Chappell, *Introducing Windows Azure*, December, 2009: http://go.microsoft.com/?linkid=9682907.
[10] M. Isard, M. Budiu, Y. Yu *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, 2007, pp. 59-72.
[11] J. Ekanayake, T. Gunarathne, J. Qiu, and G. Fox. *Cloud Technologies fo*r Bioinformatics Applications, Accepted for publication in Journal of IEEE Transactions on Parallel and Distributed Systems, 2010.
[12] G. C. Christiam Camacho, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer and Thomas L Madden, "BLAST+: architecture and applications," *BMC Bioinformatics 2009, 10:421*, 2009.
[13] NCBI. "BLAST," http://blast.ncbi.nlm.nih.gov
[14] J. Y. Choi, *Deterministic Annealing for Generative Topographic Mapping GTM*, 2009.
[15] T. Kohonen, "The self-organizing map," *Neurocomputing*, vol. 21, pp. 1--6, 1998.
[16] C. M. Bishop, and M. Svensén, "GTM: A principled alternative to the self-organizing map," *Advances in neural information processing systems*, pp. 354--360, 1997.
[17] A. Dempster, N. Laird, and D. Rubin, "Maximum Likelihood from incomplete data via the EM algorithm," *Journal of the Royal Statistical Society. Series B*, pp. 1--38, 1977.
[18] S.-H. Bae, J. Y. Choi, J. Qiu *et al.*, "Dimension reduction and visualization of large high-dimensional data via interpolation," in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, Illinois, 2010.
[19] E. Walker, "Benchmarking Amazon EC2 for high-performance scientific computing,"*;login: The USENIX Magazine*, vol. 33, no. 5.
[20] C. Evangelinos, and C. N. Hill, "Cloud Computing for parallel Scientific HPC Applications: Feasibility of running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2.," in Cloud computing and it's applications (CCA-08), Chicago, IL, 2008.
[21] J. Ekanayake, and G. Fox, "High Performance Parallel Computing with Clouds and Cloud Technologies."
[22] J. Ekanayake, and G. Fox, "High Performance Parallel Computing with Clouds and Cloud Technologies," in 1st International Conference on Cloud Computing, Munich, Germany, 2009.
[23] A. Matsunaga, M. Tsugawa, and J. Fortes, "CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications ".
[24] W. Lu,J. Jackson, and R. Barga, "AzureBlast: A Case Study of Developing Science Applications on the Cloud," in ScienceCloud: 1st Workshop on Scientific Cloud Computing co-located with HPDC 2010 (High Performance Distributed Computing), Chicago, IL, 2010.
[25] *cloudmapreduce*, Retrieved April 20, 2010: http://code.google.com/p/cloudmapreduce/.
[26] T. Gunarathne,T.L. Wu, J. Qui and G. Fox, "MapReduce in the Clouds for Science ", Accepted for 2nd International Conference on Cloud Computing 2010, Indianapolis.
[27] W. Edward, "The Real Cost of a CPU Hour,", Computer, vol. 42, pp. 35-41, 2009.
[28] J. Wilkening, A. Wilke, N. Desai *et al.*, "Using clouds for metagenomics: A case study.", IEEE international conference on Cluster Computing, CLUSTER '09, New Orleans, LA