Y790 Report for 2009 Fall and 2010 Spring Semesters

Hui Li

ID: 0002576169

1. Introduction.

There is data deluge today. In some domains, such as high energy physical, and next generation gene sequencing, the speed to obtain scientific raw data is faster than Moore's law. The volumes of data evolved from MB to GB, and it is in TB scale now. The computing resources infrastructures evolved from mainframe system to mini computer system, MPP, cluster of commodity PC, and Cloud. The programming tools for HPC evolved from C, high performance fortran, OpenMP, MPI, and MapReduce, Dryad/DryadLINQ[1][2].

The MapReduce programming model is proposed by Google, and it has been proved to be a good model for the data intensive computation. There are two main reasons why MapReduce framework satisfied the requirement of processing data intensive applications. First, it is simple, but it is of good theorem foundation. The idea comes from divide and conquer algorithm. The programming model comes from function programming. Second, it can deliver excellent scalability and fault tolerance features which are significant for distributed computation system.

The goal of my Y790 during the last two semesters is to study the features of DryadLINQ and MapReduce programming model, and study their corresponding runtimes: Dryad[1], Twister[3], Hadoop. Further, I applied these runtimes to few data intensive applications, which include: DryadLINQ PhyloD on HIV Gag p17 and p24 protein codons, Twister/Hadoop PageRank on ClueWeb data set which includes 1.4 billion web links, and EMR SWG, which ran on Aamazon cloud infrastructure.

2. Dryad/DryadLINQ

2.1 Dyrad/DryadLINQ

Dryad is a distributed execution engine for coarse grain data parallel applications. A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs[1].

DryadLINQ is a system and a set of language extensions that enable a new programming model for large scale distributed computing. It generalizes previous execution environments such as SQL, MapReduce, and Dryad in two ways: by adopting an expressive data model of strongly typed .NET objects; and by supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language [2].

My experience about the Dryad is that it is coupled with Windows HPC Job Manager and HPC Cluster Manager tightly. So it is easy to monitor the status of compute nodes, and the status of running tasks, which is very convenient for debugging. More important, the process of developing data parallel application with DryadLINQ is as simple as sequential program. The developers do not even know much knowledge about the MapReduce programming model.

2.2 DryadLINQ PhyloD

2.2.1 PhyloD Applicatoin

The **H**uman **L**eukocyte **A**ntigen can help to eliminate the HIV virus. However, the HIV virus can avoid the elimination by evolution of escape mutation. HIV mutations can be considered as HIV codons changing or evolution. The PhyloD**Error! Reference source not found.**[5] application uses statistical method to identify HLA-associated viral evolution from the sample data of HIV-infected individuals.

2.2.2 PhyloD Algorithm

PhyloD is a new statistical package to derive the association among HLA and HIV by counting given sample data. The PhyloD package have three kinds of input data: (i) the phylogenic tree information of the codons, (ii) the information about HLA alleles, and (iii) the information about HIV codons. A run of PhyloD job have three main steps. First, it computes a cross product of input files to produce all allele-codon pairs. Second, it computes the p-value for each pair, which is used to measure the association between allele-codon pair. Third, it computes a q-value per p-value, which is an indicative measure of the significance of the p-value.

The running time of PhyloD algorithm is a function of the number of different HLA alleles -|X|, the number of different HIV codons -|Y|, and the number of individuals in the study -N, which is equal to the number of leaves of the phylogenic tree. To calculate p-value of one allele-codon pair, it will cost O(N*logN) and there are |X|*|Y| allele-codon pairs. So the PhyloD algorithm runs in time O(|X|*|Y|*NlogN). The computation of the p-value of one pair can be done independently of other p-value computations. This makes it easy to implement a parallel version of PhyloD using DryadLINQ.

PhyloD executable allows user to divide the PhyloD job into a set of tasks each of which works on the assigned part of the HLA allele file and the HIV codon file. Assuming the set of HLA alleles is {A[0],A[1],..A[|X|-1]}, and the set of HIV codons is {C[0],C[1],…C[|Y|-1]}, then the set of HLA and HIV pairs is stored in the order of {(A[0],C[0]),(A[1],C[0]),(A2,C[0]),..(A[|X|-1],C[0]); (A[0],C[1]), (A[1],C[1]),(A[2],C[1]),.. (A[|X|-2],C[|Y|-1]),(A[|X|-1],C[|Y|-1])}. Accordingly, PhyloD executable divides the PhyloD job into N tasks (divide the set of all pairs into N partitions) in the same order. The index bounds of set of pairs of the K^{th} task can be calculated by following formulas.

Set B = (|X|*|Y|+N-1)/N;

If 0<=K<=N-2

Start index: (Ai,Ci)	A[i] = A[K*B%|X|];	C[i] = C[K*B/|X|]
End index: (Aj,Cj)	A[j] = A[((K+1)*B – 1)%|X|];	C[j] = C[(K+1)*B/|X|]

If K=N-1

Start index: (Ai,Ci)	A[i] = A[K*B%|X|];	C[i] = C[K*B/|X|];
End index: (Aj,Cj)	A[j] = A[|X|-1];	C[j] = C[|Y|-1];

2.2.3 DryadLINQ PhyloD Implementation

Table 1. Details of the computation clusters used in the tests

Cluster ID	Cluster-I	Cluster-II	Cluster-III
Nodes	32	32	8
Total CPU cores	768	256	64
Memory	48GB	8GB	7GB
Platform	Windows HPC Server	RedHat Linux Enterprise 5	Amazon VMs

We implemented a parallel version of the PhyloD application using DryadLINQ and the standalone PhyloD runtime available from Microsoft Research **Error! Reference source not found.**. As mentioned above the first phase of the PhyloD computation requires calculation of p-values for each HLA alleles and HIV codons. To increase the granularity of the parallel tasks, we group the individual computations into computation blocks containing a number of HLA alleles and HIV codons. Next these groups of computations are performed as a set of independent computations using DryadLINQ's "Select" construct. As the number of patients samples in each pair are quite different, the PhyloD tasks are inhomogeneous in running time. To ameliorate this effect we partitioned the data (computation blocks) randomly so that the assignment of blocks to nodes will happen randomly.

After completion of the first step of PhyloD, we get one output file for each task (computation block). The second step will merge the K*M output files together to get one final output file with the q-values of all pairs. The DryadLINQ PhyloD task decomposition and Dryad vertex hierarchy of the DryadLINQ PhyloD are shown in the Figure 1.

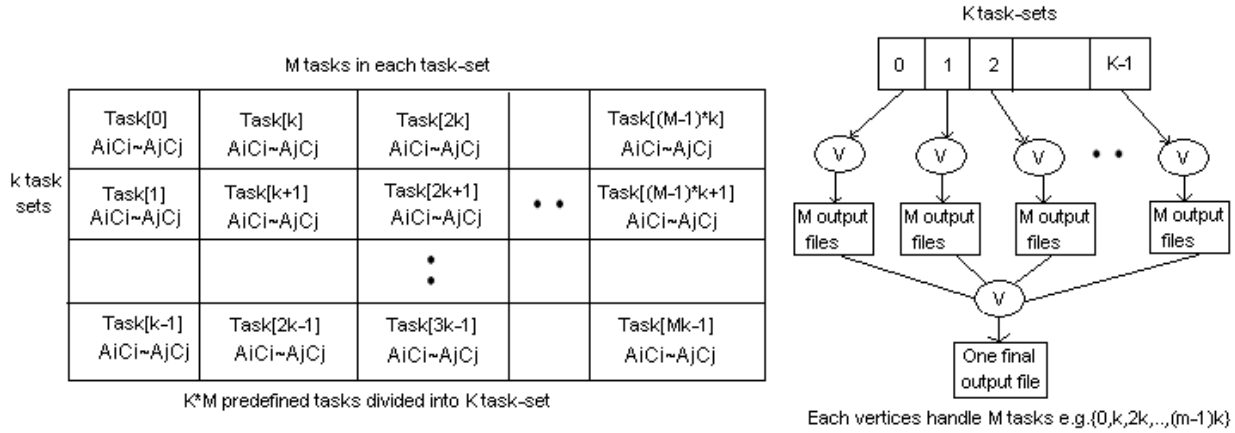

Figure 1. PhyloD task decompostion (left) and the Dryad vertex hierarchy (right) of the DryadLINQ implementation of PhyloD application

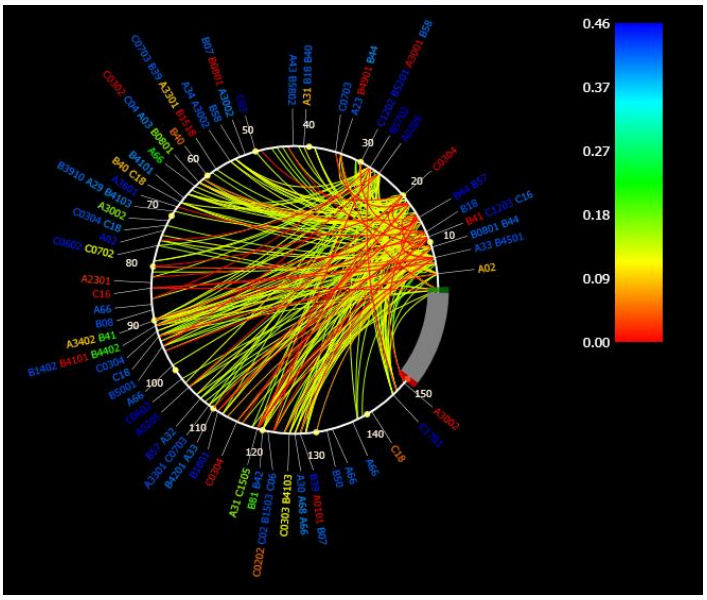

Figure 2. Part PhyloD DryadLINQ result about HIV Gag p17 and p24 protein codons

To explore the results of HLA-codon and codon-codon associations, Microsoft Research developed PhyloD viewer. Figure 2 is a PhyloDv**Error! Reference source not found.** picture of part of PhyloD results for HIV Gag p17 and p24 protein codons with the DryadLINQ implementation. HLA-codon associations are drawn as external edges, whereas codon-codon associations are drawn as arcs within the circle. Colors indicate p-values of the associations. Some associations showed on this figure have already been well-studied by scientists before. For example, the B57 allele has been proved to be strongly associated with effective HIV control **Error! Reference source not found.**.

2.2.4 DryadLINQ PhyloD Performance

We investigated the scalability and speed up of DryadLINQ PhyloD implementation. The data set includes 136 distinct HLA alleles and 841 distinct HIV codons, resulting in 114376 HIL-HIV pairs. The cluster *Ref C* is used for these studies. Figure 9 depicts the speedup of running 114376 pairs when number of cores increasing. When the number of cores increase from 192 to 384, the speed up is not as good as cases with smaller number of cores. This increasing of overhead is due to the granularity becoming smaller with the increase of number of cores. The speed up would have been better on a larger data set.

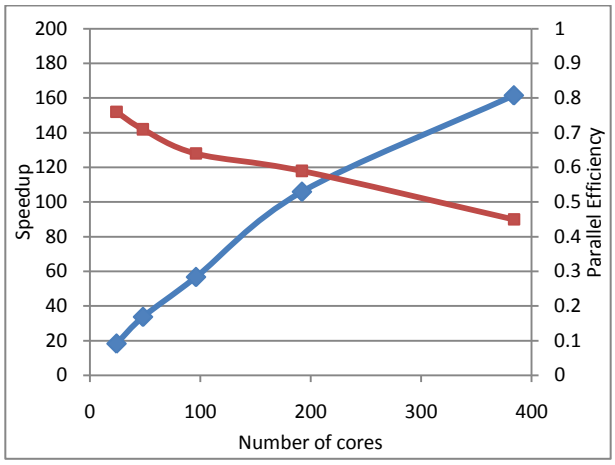

Figure 3. Speed up and parallel efficiency

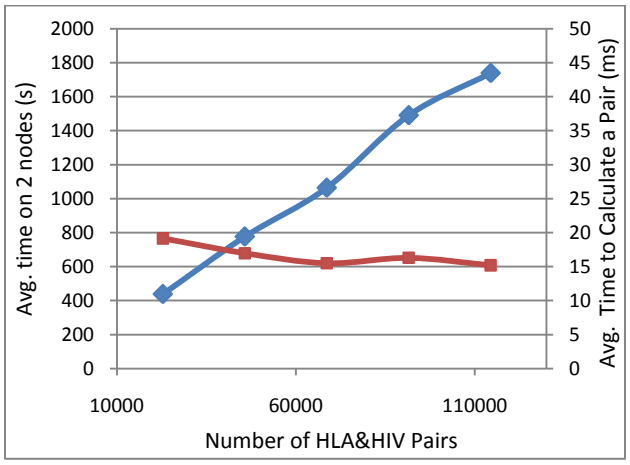

Figure 4. Scalability with increasing data size

In the scalability experiment, we run data sets with increasing data size on 2 compute nodes. As shown in figure 10 the DryadLINQ PhyloD implementation scales well when data size increase.

The current data set we have is too small for a definitive study. We intend further study of the DryadLINQ PhyloD application behavior with larger data sets. Further related job may deploy the DryadLINQ PhyloD on Azure infrastructure.

3. Twister

3.1 Twister Architecture

Twister is an enhanced MapReduce runtime that supports iterative MapReduce computations efficiently. There are two main features that allow Twister to support iterative MapReduce computations highly efficiently. First, it caches the static data in the memory, which enable these static data can be used in "configure once and use many times" approach. Second, it uses a publish/subscribe messaging infrastructure for communication and data transfers [3].

3.2 Twister PageRank

PageRank algorithm calculates numerical value to each web page in World Wide Web, which reflects the probability that the random surfer will access that page. The process of PageRank can be understood as a Markov Chain which needs recursive calculation to converge. An iteration of the algorithm calculates the new access probability for each web page based on values calculated in the previous computation. The iterating will not stop

until the difference (δ) is less than a predefined threshold, where the δ is the vector distance between the page access probabilities in Nth iteration and those in $(N+1)^{th}$ iteration.

There already exist many published work optimizing PageRank algorithm, like some of them accelerate computation by exploring the block structure of hyperlinks[10][11]. In this paper we do not create any innovative PageRank algorithm, but rather implement the most general RageRank algorithm [12] with MapReduce programming model on Twister system. The web graph is stored as an adjacency matrix (AM) and is partitioned to use as static data in map tasks. The variable input of map task is the initial page rank score. The output of reduce task is the input for the map task in the next iteration.

By leveraging the features of Twister, we did several optimizations of PageRank so as to extend it to larger web graphs; (i) configure the adjacency matrix as a static input data on each compute node and (ii) broadcast variable input data to different compute nodes so that the map tasks on the same node access the same copy of data using the object cache of Twister. Further optimizations that are independent of Twister include; (i) increase the map task granularity by wrapping certain number of URLs entries together and (ii) merge all the tangling nodes as one node to save the communication and computation cost.

3.3 Hadoop PageRank

To make the performance comparison of Twister and Hadoop, we also implemented the Hadoop PageRank. Its algorithm is similar to the Twister PageRank, but there still few differences due to the features of Hadoop. We store all the input data, adjacency matrix files, into the HDFS. All the intermediate output (the updated partial PageRank scores) are written into HDFS during multiple iterations, rather than stored in the memory. Besides, we write an extended class of FileInputFormat, so that Map task can read one am file as data.

Further, we did following optimizations to try to get the best result for Hadoop PageRank. 1) Write the merge class to do a minor merge of the intermediate results produced on the same compute node. 2) Tune the number of mappers and reducers per node to get the better results. 3) Assign the replication of data in HDFS as three. 4) Enable unlimited JVM reuse for mappers and reducers.

3.4 Performance Analysis

We make performance comparison between Twister PageRank and Hadoop PageRank with ClueWeb data set[13] collected in January 2009 and with the computing resources of Cluster-II in the table I. We built the adjacency matrix (AM) with ClueWeb data set, and splitted it into more than 4200 sub AM files. Then we made five sub data sets with these AM files, which named as CWDS1~5. Table 2 summarizes the characteristics of parts of the five sub data sets.

Table 2. Characteristics of data sets (B stands for Billions)

ClueWeb data set	CWDS1	CWDS3	CWDS5
Number of AM partitions	4000	2400	800
Number of web pages	49.5M	31.2M	11.7M
Number of links	1.40B	0.83B	0.27B
Average out-degree	28.3	26.8	22.9

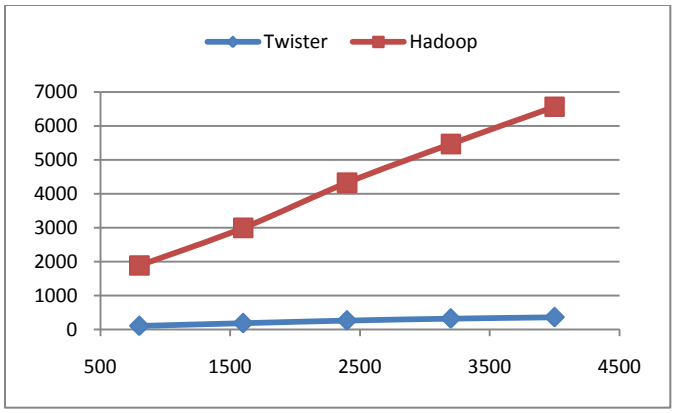

Figure 5. Job turnaround rime for 20 iterations of the PageRank implementation

We run both Twister and Hadoop PageRank for 20 iterations with the same data sets and with the same compute resources list in Cluster II. Figure 6 shows the job turnaround time of Twister/Hadoop PageRank in 5 tests. Both Twister and Hadoop PageRank have the linear increase in the job turnaround time when the size of data set increase. But we find that Hadoop PageRank is much slower than Twister PageRank due to the accumulated cost of reading/writing the input data and intermediate data from/to hard disk in the 20 iterations.

4. EMR SWG

Amazon Elastic MapReduce (EMR) is a web service that enables users easily and cost-effectively process vast amounts of data. It utilizes a hosted Hadoop framework running on the web-scale infrastructure of Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3). It allows user to dynamically obtain the resources pool based on requirement of their data intensive applications. Besides, users can focus on crunching or analyzing the data without setting up or tuning the Hadoop clusters [14].

The implementation of EMR SWG[9] is similar to Hadoop SWG, but there are two small differences. First, EMR use Hadoop 0.18 with patches rather than 0.20.x. Second, the SWG input data is stored in the S3 storage service in advance. When the computation starts, the input data will be downloaded to VMs for the further processing. We run the EMR SWG with five data sets with 8 High-CPU Extra large Instances of EC2. The computation complexity of SWG is $O(N^2)$, but there is performance degradation in EMR for large input data. Further optimizations or study should be to done for this problem.

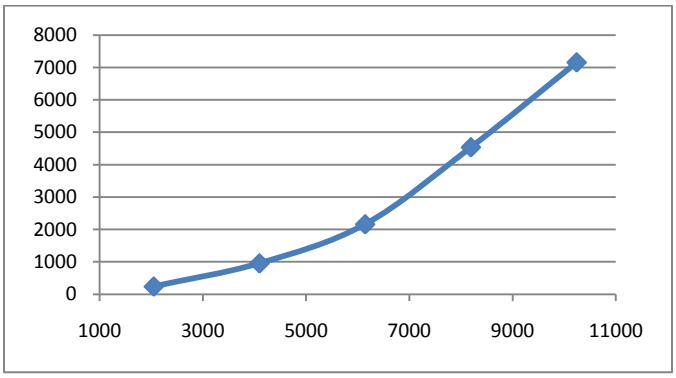

Figure 6. EMR SWG job turnaround time when input data size increase

5. Future work

5.1 HDFS

Some primary study of HDFS and Sector distributed file system has been made. Besides, the Hadoop PageRank application already employs the HDFS. The future job is to merge the HDFS with Twister, so as that Twister can handle the large scale input data more conveniently. Actually, it is easy for Twister to read/write files from/to HDFS with HDFS API. The point is how to use it in an appropriate way. In other words, the Map task of Twister should be scheduled to the compute node where the data file located.

Figure 1 shows the possible architecture and workflow for Twister to employ HDFS. There are three main steps in the work flow. 1) Twister Client invokes the HDFS API to query the file name and its storage location (host name or IP) from the HDFS name node. 2) It creates the data partition file which records all pairs of file name and its location. 3) It schedules the Map tasks according to the created partition file. Further, there are still few problems to consider about. First, there may be multiple data replication in HDFS, so it needs to decide which replication will be used for computation. Second, in this scheme, the work load balance depends on the data distribution in HDFS. The work load may not be even among multiple compute nodes. For example, the node with large disk space may process more Map tasks. Besides, this method only allows Twister to access the data in the unit of file. To make Twister access the data in the unit of block, we need to extend input format class.

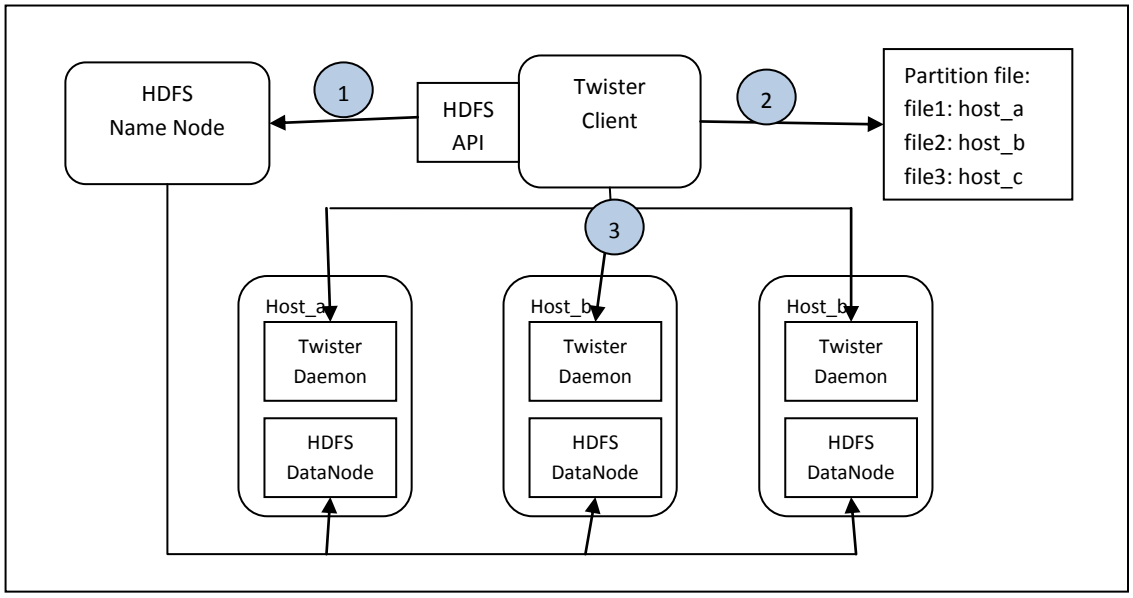

Figure 7. Architecture of Twister coupled with HDFS

5.2 FutureGrid

As Cloud technologies can help users leverage the computation resources more conveniently with lower cost, both the industry and academic community put much effort in this domain. EMR is good example of this kind of effort. It not only provides users elastic computer resources, but also hides the details of setting up resources pool, configure Hadoop cluster from the users. Deploying Twister on FutureGrid is the academic version of EMR in some degree.

The first step is to deploy Twister in FutureGrid as software, so that FutureGrid users can use Twister runtime. The next step is to couple Twister with Eucalyptus. The following figure shows the possible architecture. The Twister Client provides three kinds of interfaces for the users: web console, web service API, command API. There are two

components in Eucalyptus[15]. The storage controller provides a mechanism for storing and accessing virtual machine images and user data. The cloud controller is the entry-point into the cloud for users and administrators. There are three main steps in this flow chart. 1) Twister Client uploads the user data, and application jar file onto the storage Cloud. 2) Twister Client obtains a resources pool from Cloud controller dynamically. 3) The compute nodes download the user data and application jar file from storage Cloud to their own local disk.

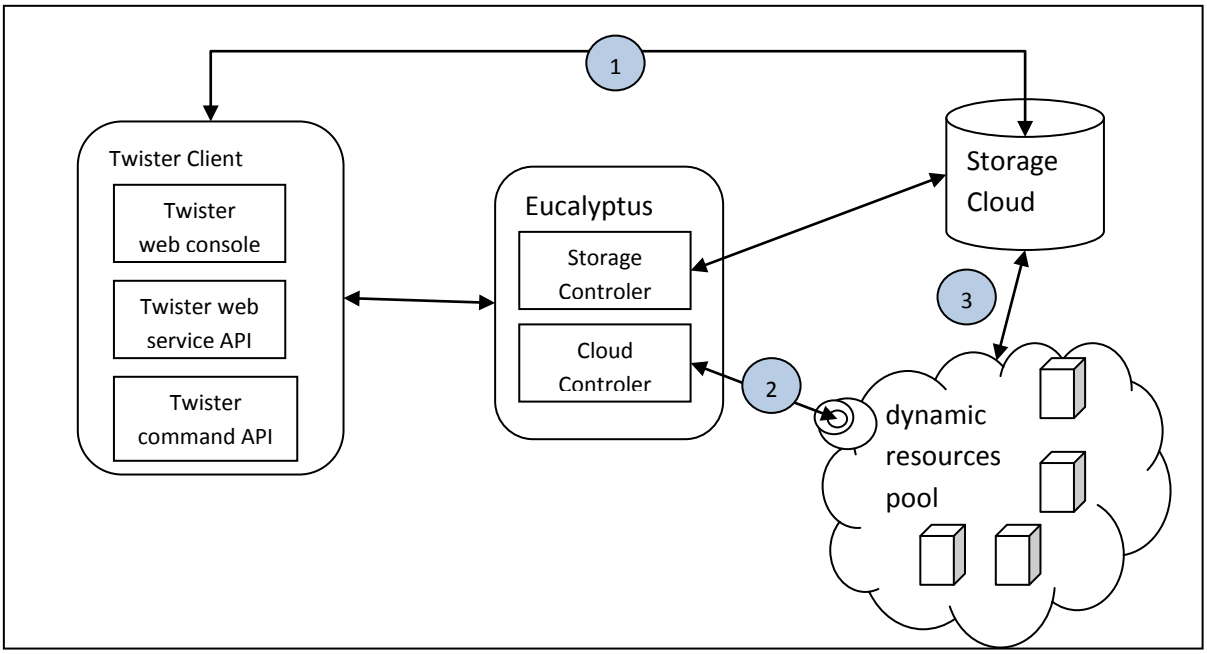

Figure 8. Architecture of Twister coupled with Eucalyptus

6. Reference

[1] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," presented at the Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, Lisbon, Portugal, 2007.

[2] Y. Yu, M. Isard, D. Fetterly, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language" presented at the 8th USENIX Symposium on Operating Systems Design and Implementation.

[3] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, Geoffrey Fox Twister: A Runtime for Iterative MapReduce, Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference, Chicago, Illinois, June 20-25, 2010.

[4] J.M. Carlson, (2008). Phylogenetic Dependency Networks: Inferring Patterns of CTL Escape and Codon Covariation in HIV-1 Gag. PLoS Comput Biol .

[5] PhyloD, http://research.microsoft.com/en-us/um/redmond/projects/MSCompBio/.

[6] J. Ekanayake, T. Gunarathne, J. Qiu, Cloud Technologies for Bioinformatics Applications, Technical report. January 4, 2010. (submitted to the Journal of IEEE Transactions on Parallel and Distributed Systems)

[7] PhyloDView,http://research.microsoft.com/en-us/um/redmond/projects/MSCompBio/PhyloDViewer/

[8] M, A. M. Altfeld (2003). Influence of HLA-B57 on clinical presentation and viral control during acute HIV-1 infection. AIDS .

[9] T.F. Smith, M.S.Waterman. Identification of common molecular subsequences. Journal of Molecular Biology 147:195-197, 1981.

[10] Y. Zhu, S. Ye, and X. Li, "Distributed PageRank computation based on iterative aggregation-disaggregation methods," presented at the Proceedings of the 14th ACM international conference on Information and knowledge management, Bremen, Germany, 2005.

[11] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub, "Exploiting the Block Structure of the Web for Computing PageRank," Stanford InfoLab, Technical Report2003

[12] The Power Method. Available: http://en.wikipedia.org/wiki/Pagerank#Power_Method

[13] 2009, The ClueWeb09 Dataset. Available: http://boston.lti.cs.cmu.edu/Data/clueweb09/

[14] EMR Available: http://aws.amazon.com/documentation/elasticmapreduce/

[15] D. Nurmi, R. Wolski, C. Grzegorczyk. The Eucalyptus Open-source Cloud-computing System. CCGRID, 2009