HyMR: a Hybrid MapReduce Workflow System

1st Author
1st author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
1st author's E-mail address

2nd Author
2nd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
2nd E-mail

3rd Author
3rd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
3rd E-mail

ABSTRACT
Various distributed computing models have been developed for high performance computing to process increasing computational data. Among them, MapReduce is one of the most popular choices and widely used. Several distributed workflow systems already exist to solve the problem which contains several MapReduce jobs. However, they have limited supports for some features such as fault tolerance and efficient execution for iterative applications inside the workflow. In this paper, we described HyMR: a hybrid MapReduce workflow system based on two different MapReduce frameworks. HyMR greatly improved the performance of data processing over the workflow systems based on a single MapReduce framework. HyMR optimized scheduling for individual jobs and supports fault tolerance for the entire workflow pipeline. A distributed file system is used for fast data sharing between jobs. We also compared a pipeline using HyMR to the workflow model based on a single MapReduce framework. The result proves that the hybrid model has a higher efficiency.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – Distributed Applications; D.2.11 [**Software Engineering**]: Software Architectures – Domain-specific architectures.

General Terms
Algorithms, Management, Performance, Design

Keywords
MapReduce, Iterative Algorithms, Workflow Management

1. INTRODUCTION
The computing power of individual computers is overwhelmed to process the ever-growing amount of data collected by modern instruments. For applications with different characteristics, various sequential and parallel paradigms have been proposed. Many of the tools are computational intensive, and some of them are iterative algorithms. Similar to these tools, current distributed workflow has large amount of data and heavy computations. Apparently, a workflow system based on single machine is not sufficient for large amount of data as many of the tools require the use of high-performance computing resources in order to achieve acceptable performance.

Various parallel programming models have been developed to alleviate the problem brought by the increasing size of data, such as Dryad [1], Sector/Sphere [2] and MapReduce [3]. Among those, Hadoop [4], an open source MapReduce implementation has been proved useful by large corporation, such as Facebook [5], Yahoo [6] and academic research, such as [7-8]. MapReduce allows the user to specify a map function which reads a key/value pair and generate a set of intermediate key/value pairs; and a reduce function which merges all the associate values belongs to a same key. Therefore, the applicability of MapReduce has been strained to limited classes of applications. For more complicated parallel applications, including iterative applications, e.g. Expectation Maximization algorithms [9], Hadoop is not sufficient. Within a parallel iterative application, Hadoop treats each iteration as a standalone MapReduce job, naturally the overhead of restarting and rescheduling map reduce tasks increases as the iterations increases.

To amend this disadvantage, several iterative MapReduce frameworks [10, 27-30] have been developed. Among these, Twister is the earliest [10]. It considers an iterative parallel application as one iterative MapReduce job. So the map and reduce tasks are scheduled before it starts and the static data is kept in distributed memory to avoid overhead of repeatedly loading during each iteration. Even though Twister can run iterative parallel applications much faster than Hadoop, it lacks distributed file system and fault tolerance support, which makes it inefficient in a workflow with multiple MapReduce jobs and large data flow between jobs.

To overcome the shortage of Hadoop and Twister used in a workflow which contains iterative MapReduce jobs, we have utilized the advancements of each MapReduce runtime. We built the Hybrid-MapReduce workflow system (HyMR), which supports both Hadoop and Twister runtimes to process workflows with multiple MapReduce jobs. First of all, since Twister is designed for iterative parallel applications, the iterative jobs inside a pipeline can be implemented using Twister instead of Hadoop. So HyMR can handle iterative jobs better than purely using Hadoop for all the distributed jobs. Secondly, we added support of Hadoop Distributed File System (HDFS) to Twister, so data flow between jobs in pipeline uses HDFS to achieve a better performance than transferring data manually among local file system of each compute node. So the total time for running a workflow pipeline will decrease accordingly. Thirdly, since Hadoop has a better fault tolerance scheme, the non-iterative applications can be implemented using Hadoop. So if a compute node fails during the execution, HyMR does not have to restart the whole job but let Hadoop handles the failure. Fourthly, we added multiple-user support and used XML to record the workflow's

Conference'10, Month 1–2, 2010, City, State, Country.

status, where can be extended to webserver so user can monitor and submit jobs online.

We have evaluated HyMR on real cluster using a pipeline contains three distributed jobs and several single node jobs. The pipeline contains two MapReduce jobs and one iterative MapReduce job. We have implemented the pipeline in Twister-workflow and in HyMR-workflow to compare the results.

This paper is organized as follows: Section 2 presents the detail of the design and implementation of HyMR-workflow, Section 3 has introduced the workflow pipeline we have been testing with some improvements over HyMR. Section 4 demonstrates the efficiency of HyMR with experiments which includes performance comparison, scale up test and fault tolerance test. In Section 5, we talked about related work of iterative MapReduce and distributed workflow system. In Section 6, we summarize the contribution of our work and describe the future work.

2. HyMR Workflow System

HyMR comprises an interface, an instance controller, a job controller and a runtime support system as shown in Figure 1. HyMR can run on traditional HPC clusters and supports TORQUE [11] so far. When a workflow is submitted, HyMR interacts with TORQUE to request resources. Once the resources are allocated, HyMR will configure and start runtimes. After that, jobs in the workflow are executed accordingly.

User interface enables the user to design a workflow by giving a definition file. The file describes a specific workflow, which can include Hadoop configuration, Twister configuration and job executions. The job definition file can be written either in *XML* or a simple properties file. The interface can generate an instance file in *XML* format so that the user can monitor the progress of the instance. This design has the potential to monitor and submit the jobs from a web server.

Instance controller controls a workflow instance, which is an actual executing copy of a workflow definition. It will generate a Direct Acyclic Graph (DAG) based on the description given in the definition file where the vertices are jobs and the edges are the dependencies between jobs. The instance controller processes the control flow accordingly by invoking the job controller once a job's dependents are finished. If a workflow contains MapReduce job, the instance controller will notify the runtime controller system where corresponding runtime will start if needed.

Runtime controller starts Hadoop and Twister accordingly at beginning of the workflow, called persistent runtime. Because by using TORQUE, the compute nodes allocated for an instance are consistent during the execution. So for a pipeline contains more than one distributed job, the persistent runtime reduces time cost for it only starts and stops the runtimes once, instead of starting the runtime at the beginning and stopping the runtime at the end of every distributed job. After the instance is finished or failed, the controller will then shut down the runtimes.

Job controller controls a job execution. An instance controller will invoke it when all the jobs that a job depends on are finished inside the instance. A job controller can support jobs vary from single node job, such as single thread and multi-thread jobs, to distributed jobs, such as MapReduce and iterative MapReduce jobs. As we have added support for Twister fault tolerance, for a Twister job, a Twister fault detector will be invoked and keep checking whether every daemon is alive. Twister fault detector will kill the current job once it detects a Twister daemon failure. No such fault detector is needed for Hadoop since it can handle the fault tolerance by itself. Once a job fails during execution, the job controller will notify the instance controller to restart the corresponding runtime if this failure is a Hadoop or Twister runtime failure. Besides runtime failures, if a job fails, the job controller will simply return an error to instance controller and be marked as failed. If retry times of a certain job is below maximum retry number allowed, instance controller will restart the job. Otherwise, it will kill all running jobs and marked this instance as failed.

Hadoop and Twister both have advantages and disadvantages comparing to each other. By using HyMR for a pipeline includes multiple MapReduce jobs, we are trying to eliminated the shortage of the two runtimes and achieve a better performance by merging them.

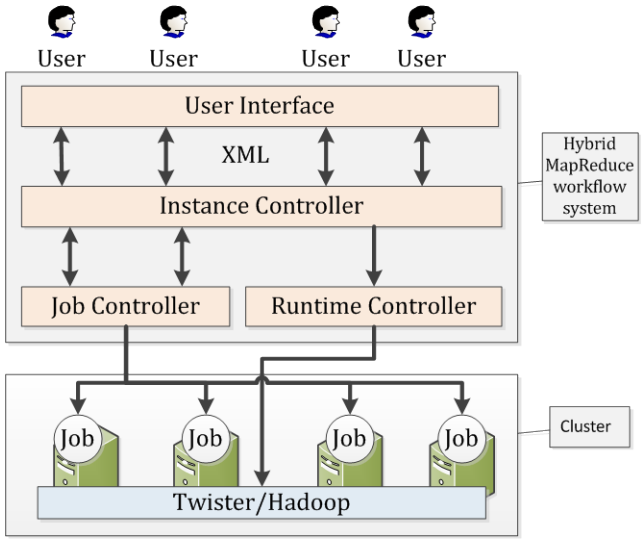


Figure 1 The design and architecture of HyMR interacts with a cluster

2.1 MapReduce and Hadoop

MapReduce is a parallel programming model proposed by Google to support large-scale data processing. Hadoop is an open source implementation of MapReduce with a distributed file system (HDFS) [12]. Each MapReduce job takes a set of key/value pairs as input, and produces a set of key/value pairs. The computation of MapReduce jobs is split into 2 phases: *map* and *reduce*. In map phase, map function takes an input key/value pair, read the data accordingly from HDFS, and produces a set of intermediate key/value pairs. Hadoop groups together all intermediate values associated with the same intermediate key and passes them to the reduce function. In reduce phase, each reduce operation accepts an intermediate key and all values associate with that key. It merges together these values to form a possibly smaller set of values, emits key/value pairs of the final output, and writes the final output to HDFS. The execution of a typical MapReduce job on Hadoop is shown in Figure 2a.

Hadoop adopts master-slave architecture. The master node runs a namenode, a secondary namenode and a job tracker. The namenode is mainly used for HDFS for hosting the file system index; the secondary namenode can generate snapshots of the namenode's memory structures, thus preventing file system corruption and reducing loss of data; the job tracker allocates work to the task tracker nearest to the data with an available slot. The slave node runs a datanode and a task tracker: datanode contains blocks of data inside HDFS where multiple datanodes

can serve up distributed data over network; task tracker will spawn java processes as workers to execute the work received from job tracker.

Hadoop supports fault tolerance in both MapReduce execution and HDFS. The HDFS supports fault tolerance by with providing file replicas among the slave nodes. In case one or several datanodes fail, the integrity of the distributed files won't be harmed by using the replicas from other running datanodes. During a MapReduce job execution, once a task tracker fails, all the unfinished tasks on that task tracker will be scheduled to the empty slots of other task tracker. So if any of the slave nodes fail during an execution of Hadoop MapReduce job inside a workflow, HyMR will let Hadoop handle the failure. However, if the namenode or job tracker fails, Hadoop will fail and wait for the user to manually restart it. Therefore, to overcome this, HyMR will automatically restart Hadoop and the MapReduce job once the masternode fails.

2.2 Iterative MapReduce and Twister

Many data analysis applications require iterative computations. The algorithms such as K-Means [13], Deterministic Annealing Clustering [14] and Dimension Reduction algorithm [15] can be paralleled with MapReduce paradigm. This type of applications has a common feature that is to keep running iteratively until the computation satisfies a condition to converge to a final result.

Hadoop has been proved to be useful in large scale data parallel distributed computing job. However, it does not directly support iterative data analysis applications. Instead, the iterations must be orchestrated manually using a driver program where each iteration is piped as a separate MapReduce job. It brought several shortages: First of all, the iteration number is manually set by the user, it is impossible to make the program converge to meet a certain condition; Secondly, the static data needs to be load from disk to memory in every iteration which can produce large network I/O and disk I/O overhead; Thirdly, the job tracker needs to reschedule map and reduce tasks every iteration, which brings numerous scheduling overhead.

So we use Twister, the earliest iterative MapReduce framework to run the iterative parallel applications inside our workflow. Twister uses pub/sub messaging for all the communication/data transfer where a broker will be running on one of the compute nodes during the execution. This node is classified as head node in HyMR. All the other nodes are classified as compute nodes where the Twister daemon will be running on. Twister daemon can connect to broker and spawn threads executing map and reduce tasks.

The client driver of Twister is used to support iterations of map and reduce tasks. The map and reduce tasks won't exit after one iteration unless the client driver sends the exit signal. So an iterative MapReduce job is not considered as multiple MapReduce jobs as in Hadoop. Twister added support for long running mappers and reducers with data kept in memory. This design eliminates the overhead of data reloading from disk to memory across iteration boundaries. Twister schedules map and reduce tasks before the first iteration so that they are processed by the same mappers and reducers in every iteration. This can eliminate the scheduling overhead of task rescheduling. Overall, Twister is optimized for iterative parallel applications, for which Hadoop is inefficient. Even though Twister runs fast for iterative parallel applications, Twister lacks several features for large-scale usage:

1) **Distributed file system support**. Hadoop uses HDFS for large distributed data storage while Twister needs data staging before MapReduce job starts. Usually in data staging phase, a Network File System (NFS) is used to store all the data first, and then the user needs to partition the file and distribute the file to each compute node's local disk. A typical Twister MapReduce workflow job is shown in Figure 2b. This could result in a high network overhead if several MapReduce jobs were piped on a large number of compute nodes. The data transfer time cost from one MapReduce job's result to another MapReduce job's input will be high.

2) **Fault tolerance**. Even though Twister stated that it supports fault tolerance in a heartbeat and checkpoint fashion, in practice, Twister can detect if daemon running on compute node has failed, but it couldn't resume the running application. So the application will be paused until the user manually restarts the Twister daemons and the MapReduce job.

3) **Dynamic Scheduling**. Twister can be faster on iterative applications by using static scheduling than Hadoop since it eliminates the rescheduling overhead. However, for non-iterative MapReduce job, static scheduling could result in slow execution while each map task's time cost is non-deterministic. The typical Twister usage is to partition tasks beforehand. Compared to Hadoop which can utilize the resources by scheduling map task dynamically, Twister can only run map tasks where the partition number is equal to core number.

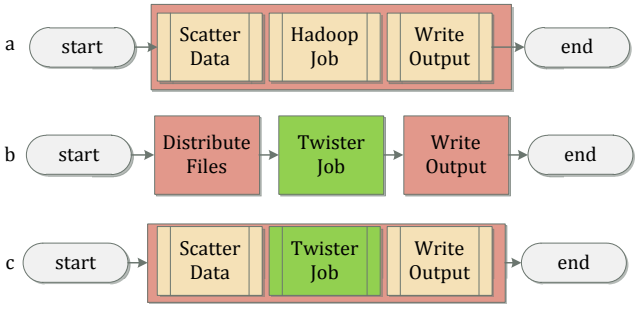


Figure 2 (a) Hadoop Job (b) Twister Job (c) Twister Job with HDFS

The data staging for (a) and (c) is handled by HDFS implicitly. For (b), an explicit data staging is happened before Twister job

2.3 HyMR improvements

HyMR uses a static node allocation while executing a pipeline. As the node list will be given by TORQUE at the beginning of the workflow, the HyMR will select one of the nodes as head node where it runs namenode, secondary namenode and job tracker from Hadoop, and broker from Twister; all the other nodes are compute nodes where they runs datanode, task tracker from Hadoop and Twister daemon from Twister.

HyMR has several improvements trying to resolve the problem that Hadoop has with iterative applications and Twister has with multiple MapReduce job in a pipeline:

Faster execution: because Hadoop runtime is not efficient for iterative parallel applications which Twister is highly efficient for, so by using Twister for iterative parallel applications inside a pipeline can greatly improve performance over a pipeline using purely Hadoop. Therefore, inside a pipeline executed by HyMR, all the iterative parallel applications are run on Twister. The performance of hybrid usage of Twister and Hadoop will be more efficient compared to solely use Hadoop.

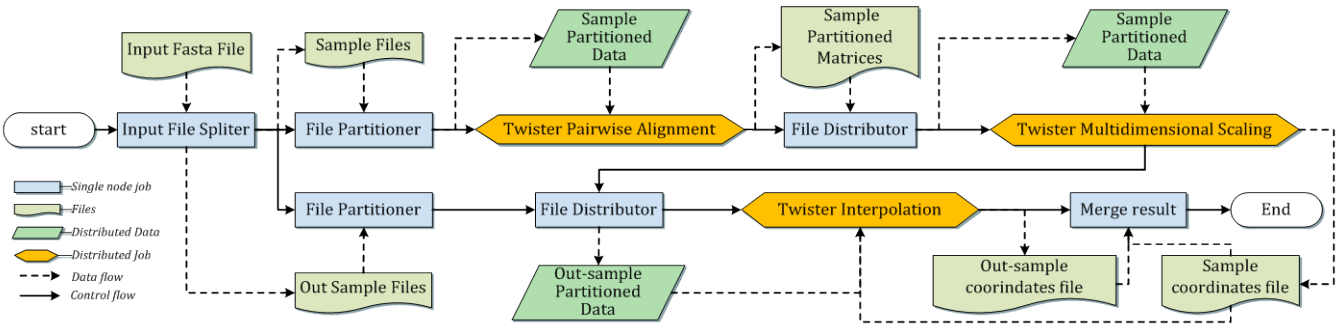


Figure 3 A distributed bioinformatics visualization pipeline based on Twister

Distributed File system support: a pipeline purely using Twister lacks the support of distributed file system. Even though Twister can process iterative parallel applications much faster than Hadoop with data already distributed onto each compute nodes' local disk, the file partition and distributing could still be time consuming. When a Twister MapReduce job finishes, the output data is written to a shared storage. The next MapReduce job will need to perform data staging phase again for the previous output data. Hence, we integrated the HDFS to Twister supported component inside HyMR, where the Twister MapReduce job can directly write and read data from HDFS without having the data staging phase beforehand. This may cause slower execution of the distributed job where HDFS I/O has higher overhead than local disk. But this could result in a faster execution for a pipeline consists by multiple MapReduce job to avoid intermediate data staging. So a pipeline uses HyMR will be faster than not only a purely Hadoop pipeline, but also a purely Twister pipeline during a fault-free execution. An HDFS-enabled Twister MapReduce component inside HyMR is shown in Figure 1c. The data staging phase are not needed anymore, and similar to Hadoop MapReduce job, the HDFS enabled Twister MapReduce workflow is considered as one job.

Better Fault Tolerance: Since Twister lacks fault tolerance support, once one or more Twister daemons fail during the computation, Twister client driver will be paused unless the user manually kill it and restart the whole process. Therefore, in our hybrid MapReduce model, we use Hadoop for non-iterative applications and Twister for iterative applications. If any of the tasks fails during the execution of a Hadoop job, HyMR will not be notified since the Hadoop will handle the fault. However, if the process on headnode fails, the job controller in HyMR will be notified and restart Hadoop and MapReduce job. Additionally, since the iterative MapReduce jobs have to be executed on Twister, we have implemented a heartbeat failure checker for Twister inside the job controller as mentioned previously. The instance controller will immediately terminates the Twister job, restart Twister and reschedule the Twister job inside the pipeline once a failure of Twister daemon is detected. In this way, the user doesn't need to monitor the pipeline all the time and manually reschedule the job. In another word, this mechanism can save the time between the runtime failure and when the user discovers it.

3. A Bioinformatics Visualization Pipeline

We have implemented a bioinformatics data visualization pipeline using HyMR in two ways. One implementation Hybrid-pipeline is to use our hybrid model that MapReduce jobs are implemented by Hadoop and iterative MapReduce job are implemented by Twister with HDFS. The other implementation Twister-pipeline is to use Twister for all the distributed jobs. This data visualization pipeline contains three distributed jobs and several single node jobs. Among the distributed jobs, Pairwise Sequence Alignment and MDS Interpolation are MapReduce job while Multidimensional Scaling is an iterative MapReduce job.

The Twister-pipeline is shown in Figure 3. The first job, "Input File Splitter" splits the input file into a sample file and an out-samples file, which are required in Twister Pairwise Sequence Alignment (Twister-PSA) and Twister MDS Interpolation (Twister-MI-MDS). A data staging is required for each Twister MapReduce job for fast execution where the staging area is the local disk on each compute node. Therefore, files are partitioned and distributed before generating the partition data for Twister MapReduce job. Because the outputs from Twister-PSA are already partitioned matrix files, so before the Twister Multidimensional Scaling (Twister-MDS) begins, only a file distributor is needed for data staging. Finally, after all the MapReduce job is finished, the job "Merge Result" will merged the sample result and out-sample result to generate the final output.

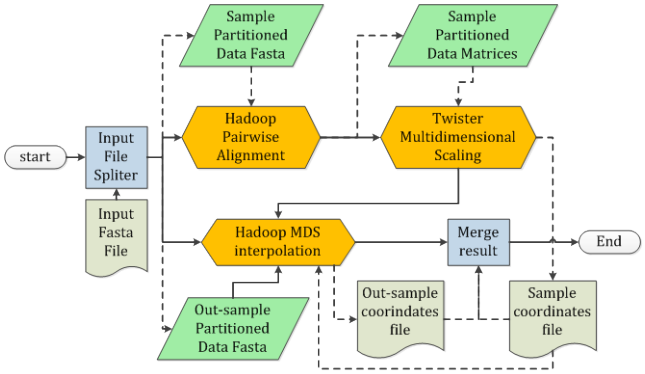


Figure 4 A distributed bioinformatics visualization pipeline based hybrid usage of Hadoop and Twister

The Hybrid-pipeline is shown in Figure 4. It is very similar to the Twister-pipeline. But it only has two single node jobs: "Input File Splitter" and "Merge Result", which are the same components as in Twister-pipeline. No explicit data staging phase is needed as data are stored in shared HDFS. For Hadoop Pairwise Sequence Alignment (Hadoop-PSA) and Hadoop MDS Interpolation (Hadoop-MI-MDS), the distributed input files are put into the HDFS before the MapReduce job starts; the distributed input data for HDFS Twister Multidimensional Scaling (HDFS-Twister-MDS) can be read directly from output of Hadoop-PSA inside HDFS.

As Figure 3 and Figure 4 show, even though these two pipelines have similar DAG, the data flow of Twister-pipeline is more complicated than that of Hybrid-pipeline. The simplified DAG of Hybrid-pipeline shows that all the intermediate data are stored in

HDFS, which is highly optimized for handling parallel file transferring.

3.1 Pairwise Sequence Alignment

Sequence alignment arranges the sequences of DNA, RNA, or protein to identify similar regions. Pairwise sequence alignment (PSA) does all-pair sequence alignment over a given sequence dataset, which is usually in FASTA format, where the result is generated as an all-pair dissimilarity matrix [16]. There are many pairwise sequence alignment algorithms, such as Smith-Waterman-GOTOH (SWG) [17] and Needleman-Wunsch [18]. In this particular pipeline, we use SWG algorithm for the each pair's sequence alignment since our SWG java version is implemented as a light weighted library. Despite the different alignment algorithm choices, the parallelism method of calculating the all-pair dissimilarity matrix remains the same.

The parallelism of Twister-PSA and Hadoop-PSA program is also the same. The input FASTA format gives a sequence begins with a single-line description, followed by lines of sequence data. It is partitioned into row blocks, where each row block contains the same number of sequences. Each row block pairs compute a result distance matrix block. The structure of the parallelized result matrix is given in Figure 5.

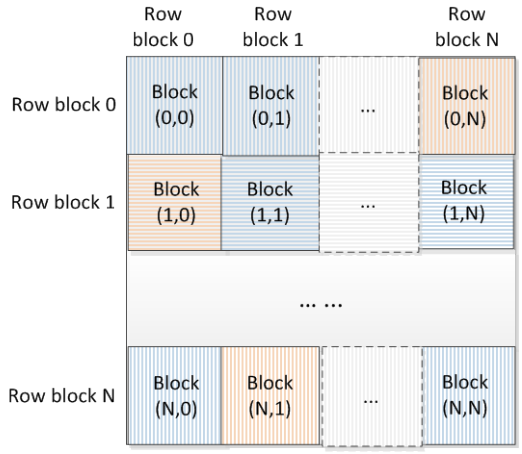


Figure 5 Architecture of parallel pairwise sequence alignment

So as shown in Figure 5, the dissimilarity matrix can be computed using *N*(N+1)/2* blocks where *N* is the number of row blocks. In the given example, the diagonal blocks need to be computed, and *block(0,1)*, *block(1,N)* and *block(N,0)* need to be computed where the *block(1,0)*, *block(N,1)* and *block(0,N)* are symmetric to them correspondingly. In traditional all pairs problem, each matrix block is considered as a map task. So if there are *N* input row blocks from the sequence file, there will be *N*(N+1)/2* map tasks and *N* reduce tasks. In Hybrid-pipeline, as our Twister component requires the partition number equals to the total core number of the compute nodes we used, therefore, the partition number in the Hadoop component should also be the same. However, as we have tested this pipeline on hundreds of cores scale, the map task number of Hadoop-PSA will be too many if we use the traditional way. This could result in huge amount of time cost in scheduling the map tasks. So we have proposed a new way of giving key/value pairs to the Hadoop map tasks.

In our implementation, an entire matrix row block, which contains all the *N* blocks in that row, will be considered as a map task instead of one block. In Figure 5, the row block 0 contains *block(0,0)*, *block(0,1)* to *block(0,N)*. So each map task will contain either *N/2* or *(N+1)/2* matrix blocks' computation. In this way, the map task number can be reduced from $O(N^2)$ to $O(N)$ where the granularity of each map task is increased. This could also result in reducing the overhead of job scheduling in Hadoop. As each task is computational intensive and have nearly equally amount of computation, this new assignment should result in reducing the overall computation time compare to the original one.

3.2 Multidimensional Scaling

Multidimensional scaling (MDS)[19] is a set of related statistical techniques often used in information visualization for exploring similarities or dissimilarities in data. An MDS algorithm can read from a dissimilarity matrix and assign a location for each item in the matrix to an R-dimensional space. Scaling by Majorizing a Complicated Function (SMACOF) [20] algorithm is one of the MDS algorithms that have been proved to be most fast and efficient on a distributed system.[21-22] SMACOF is an iterative algorithm where it uses an Expectation Maximum (EM) method to minimize the object function value, called stress of target dimensional space mapping [23]. We have used the parallel SMACOF application inside our pipeline.

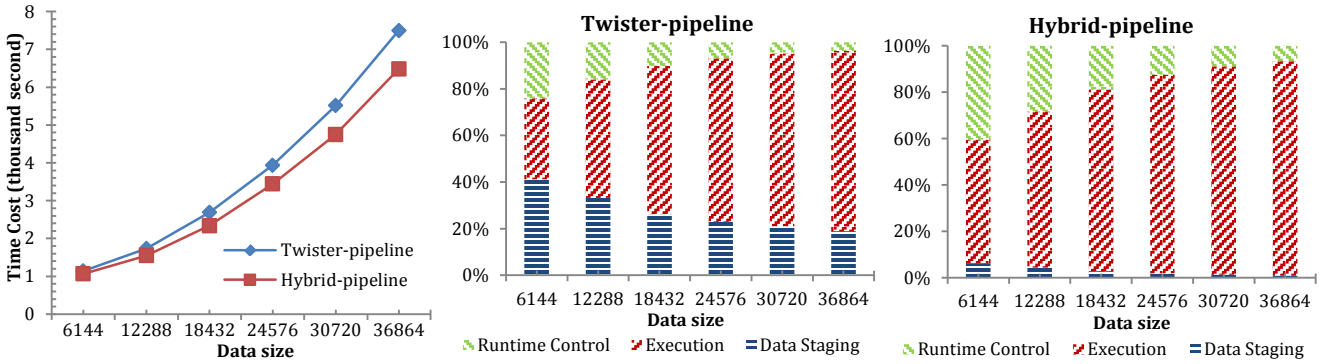
As the input of MDS is a dissimilarity matrix, the output from PSA component can be directly read as input for MDS component in the pipeline. In both Twister and Hybrid pipelines, we only used Twister-MDS as the second distributed component. The iterative MapReduce SMACOF application uses two MapReduce computations in a single iteration which involves on matrix multiplication and one stress calculation.[24] Hadoop is expected to be highly inefficient for this application. The difference between the Twister-MDS inside these two pipelines is that in the Twister-pipeline, the partitioned matrixes are transferred and read from each compute node's local disk; in the Hybrid-pipeline, as the support to HDFS has been added to the Twister-MDS, the data matrixes can be directly load from HDFS.

3.3 MDS Interpolation

As the memory usage for SMACOF is $O(N^2)$, it has become a limitation for running large scale sequences on parallel MDS application. To overcome this problem, Bae et al. has proposed develop a simple interpolation approach based on pre-mapped MDS result of the sample of the given data. The algorithm proposed is called Majorizing Interpolation MDS (MI-MDS).[25]

As the input for MI-MDS is the mapping result from MDS and a set of sequences, we split the input sequences of pipeline to two sets: one is called sample set, another is called out-sample set.[26] The PSA and MDS can be applied on the sample set, and generate the sample mapping result. Then by using the sample mapping result and the out-sample set, we can execute the MI-MDS to get the out-sample result. Finally, at the end of the pipeline, the sample result and out-sample result can be merged as the final output.

As the interpolated points from out-sample set in MI-MDS are totally independent one another, the MI-MDS algorithm is pleasingly-parallel. We have implemented Hadoop-MI-MDS for Hybrid-pipeline and Twister-MI-MDS for Twister-pipeline. In both implementations, the sample mapping result is read from a shared location where the size of it is very small. Hadoop-MI-MDS uses the HDFS to store the out-sample data. Twister-MI-MDS partitions out-sample set first, and then transfer it to each compute node's local disk. And in Twister-pipeline, out-sample data partitioning is started in parallel with sample data partitioning.



a. Time cost of Twister-pipeline and Hybrid-pipeline

b. Time proportion of different stages in Twister-pipeline

c. Time proportion of different stages in Hybrid-pipeline

Figure 6 Performance comparison chart between Twister-pipeline which only use Twister as MapReduce runtime and Hybrid-pipeline use both Hadoop and Twister. Input data size is increased from 6144 to 36864 and core number is fixed to 768

4. Experiments

The experiments were done on PolarGrid (PG) using 129 nodes, which contains 1 head node and 128 compute nodes. PG is a cluster composed of IBM HS21 Bladeservers and IBM iDataPlex dx340 rack-mounted servers with Red Hat Linux. The compute nodes in our experiments are iDataPlex dx340 rack-mount servers with 8-core nodes. So the total number of cores we used is up to 1024. The data is selected within 16S rRNA data from the NCBI database. The total input sequence number is 36864. All of the tests are done on Twister-pipeline and Hybrid-pipeline. Because MDS component needs an average number of 400 iterations to converge, so we did not implement Hadoop-pipeline which will be highly inefficient.

4.1 Performance Comparison

This test ran on 96 nodes, 768 cores with the number of sequences ranging from 6144 to 36864 on 96 nodes, 768 cores. Half of the sequence set is classified as sample set and the other half is out-sample set. The test purpose is to prove the overall performance of Hybrid-pipeline is better with a larger scale of data comparing to the Twister-pipeline.

These two pipelines have been described in section 3. The time cost of both pipelines can be divided into three portions: data staging time, job execution time and runtime control time. Twister MapReduce job needs a data staging to each compute node's local disk every time before it executes. The data-staging time includes the file partitioning, local directory created on each compute node's local disk and file transferring time. The execution time includes Twister-PSA, Twister-MDS and Twister-MI-MDS's execution time. Reading and writing data to local disk is also includes in execution time. Runtime control time is the time of starting and stopping Twister runtime by HyMR. The time cost of Hybrid-pipeline can be divided into the same three proportions as well. The data staging time is the time for Hadoop to partition and scatter the key/value pairs before map tasks. The execution time in Hybrid-pipeline includes the time of executing Hadoop-PSA, HDFS-Twister-MDS and Hadoop-MI-MDS. The time of reading and writing into HDFS is also considered into executing time. The runtime control time is time to start and stop both Hadoop and Twister. The comparison of the overall time differences between these two different pipelines is shown in Figure 6a. Hybrid-pipeline has lower time cost than Twister-pipeline. The time difference between two pipelines increases as the data size increases. The detailed proportion of time cost between these two pipelines is shown in Figure 6b and 6c.

For Twister-pipeline, data staging includes partitioning and transferring the sample/out-sample FASTA files before Twister-PSA and Twister-MI-MDS components and the sample dissimilarity matrices before Twister-MDS. The out-sample FASTA files has a smaller size comparing to the dissimilarity matrices, so the data staging time for Twister-pipeline has been increased from 459 seconds to 1389 seconds mainly because of the increasing time of transfer each 2D matrix to the compute node. In Hybrid-pipeline, data scatter only happens inside the Hadoop-PSA and Hadoop-MI-MDS components. There is no data distribution time for HDFS-Twister-MDS as it can directly read partitioned matrices from the output of Hadoop-PSA in HDFS. The data partition time isn't increasing because the input file size only increases from 3MB to 18MB. The number of key/value pairs is fixed since the total core number is fixed and we used 2 times the core number of map tasks for both of Hadoop-PSA and Hadoop-MI-MDS. Therefore, this value remains around 65 seconds. As mentioned earlier, distributed job execution time includes the time of reading and writing data for both of the pipelines. Hybrid-pipeline's execution time is higher than Twister-pipeline as reading from HDFS is slower than reading from each compute node's local disk. Also scheduling overhead in Hybrid-pipeline is higher than Twister-pipeline because Hadoop uses dynamic scheduling while Twister adapts a static fashion. Runtime control time of Hybrid-pipeline is higher than Twister-pipeline because of starting both Hadoop and Twister cost more time than only starting Twister.

For Twister-pipeline, since data staging time is increasing, the proportion of it in Figure 6b doesn't decrease a lot while the data size increases. But the proportion of data scatter time keeps decreasing in Figure 6c for Hybrid-pipeline as job execution time increases. The runtime control time for both of the pipelines is the same with different data size since node number is fixed. Because Twister-pipeline always performs better in distributed job execution, and data staging time cost in Twister-pipeline is significantly more than data scatter time cost in Hybrid-pipeline, so the proportion of execution time in Twister-pipeline is less than Hybrid-pipeline.

In this experiment we illustrate that the overall performance has been improved up to 15% by using Hybrid-pipeline over Twister-pipeline in Figure 6a.

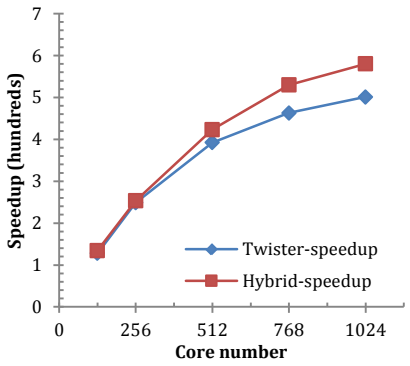


Figure 7 Speed up of Twister-pipeline and Hybrid pipeline with input data size 24576

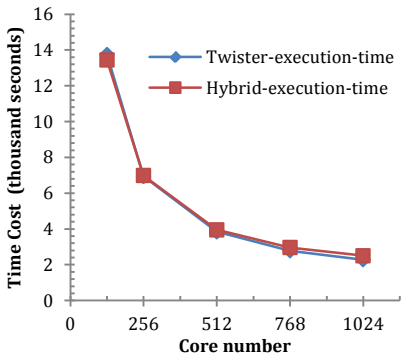


Figure 8 Time cost of three distributed job PSA, MDS and MI-MDS in both Twister-pipeline and Hybrid-pipeline with input size 24576

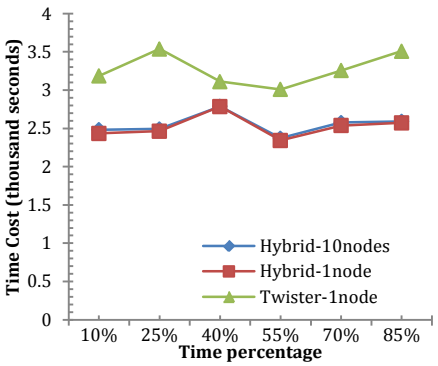


Figure 9Time cost of Twister-pipeline and Hybrid-pipeline with fault with input data size 18432

4.2 Scale Up Test

This test is done with an input data size of 24576 sequences. Five tests were done by using node number 16, 32, 64, 96, 128 and core number 128, 256, 512, 768 and 1024. We have used the performance on 128 cores as a baseline where the speed up can be calculated using Eq.1 :

$$S_n = K * \frac{T_k}{T_n} \qquad \text{(Eq 1)}$$

where the T_k is the time cost of Kcores, T_n is the time cost of N cores, and S_n is the speed up on N cores. In our pipeline, $T_k$is the time of Twister-pipeline running on 128 cores where $K = 128$

The speedup result is shown in Figure 7. Hybrid-pipeline has better speedup than Twister-pipeline when number of cores increases. From 128 cores to 256 cores, the speed up is around 248 for Twister-pipeline and 254 for Hybrid-pipeline which shows highly efficiency for both of the pipelines. However, when core number keeps increasing, Twister-pipeline is less efficient than Hybrid-pipeline mainly because of data staging time increases. As Twister-pipeline use file transfer operation from a single node for data flow between jobs, the network overhead increases when node number increases. Therefore, even though the data size is fixed, the data staging phase takes 470 seconds on 128 cores and it takes 996 seconds on 1024 cores to finish the data distribution. The data staging time of Hybrid-pipeline also increases as the key/value pair increases. However, it only cost 15 seconds on 128 cores and 108 seconds on 1024 cores. The runtime control time also increases when core number increases. This is because Twister daemon number and Hadoop datanode and tasktracker number equal the number of compute nodes. Hence, more compute nodes are, more processes needs to be created for Twister and Hadoop. Naturally the runtime control time increases.

In Figure 8, the execution time of both Twister-pipeline and Hybrid-pipeline is showed. Twister takes less time on executing distributed jobs as the core number increases. So the time difference increases when core number increases. It shows that Hadoop has higher scheduling overhead as well as HDFS has a higher overhead for more data blocks that on a larger number of cores. Twister-pipeline took 2283 seconds of execution and Hybrid-pipeline took 2503 seconds on 1024 cores. However, this factor is not dominating Hybrid-pipelines executing time when core number is relatively small.

After adding each portion of time together, even though Hybrid-pipeline's distribution job execution time is higher, the overall execution time of the pipeline of Hybrid-pipeline is lower due to a relatively small time cost on data staging. So the speedup of Hybrid-pipeline in figure 7 shows a maximum of 15% higher efficiency than Twister-pipeline.

4.3 Fault Tolerance Test

The fault tolerance test is done on 96 nodes, 768 cores with an input size of 18432 sequences, where all the workers on 1 node or 10 nodes are killed in different time during the computation.

To fully test the fault tolerance in HyMR, the failure is done at the distributed job execution time. We have chosen the failure time at 10%, 25%, 40%, 55%, 70%, 85% of distributed job execution time inside the pipelines first, then we add the runtime start time and data distribution time in so the failure times can covers all the three distributed jobs. This setting gives two fault happens in the PSA component, one fault happens in the MDS component and three faults in the MI-MDS component. A fault happens by killing all the tasks on the given node. For Hadoop component, we directly kill the task processes from the MapReduce job. And for Twister component, since the daemon spawns thread to execute the MapReduce tasks, killing the tasks will result in failure of the Twister daemons.

Both Hybrid-pipeline and Twister-pipeline are tested with the workers killed on 1 node and 10 nodes. For a Twister-pipeline, a Twister daemon failure will result in restarting the MapReduce job it runs; For the Hybrid-pipeline, if the fault happens within Hadoop component, the Hadoop runtime will handle the failure by rescheduling the map task and restarting map/reduce tasks. If the fault happens within Twister component, a same fault recovery strategy will be used as in the Twister-pipeline.

Finally, as both of the pipelines are managed by HyMR system, all the jobs can be automatically recovered once a failure happens.

The fault tolerance test result is shown in Figure 9. In our experiments, the 10 nodes failure and 1 node failure gives the same result within in Twister-pipeline. This is because either fault will make the workflow system to restart the Twister runtime and rerun the MapReduce job. So in our result Figure 8, only 1 time line is displayed for Twister-pipeline.

In a fault-free execution with input size of 18432, the Twister-pipeline takes totally 2691 seconds with 1685 seconds of computation in average to finish while Hybrid-pipeline takes average 2338 seconds with 1804 seconds of computation to finish. At time point 10% and 25%, the Hybrid-pipeline was executing

Hadoop-PSA component; at time point 55%, 70% and 85%, Hybrid-pipeline was executing Hadoop-MI-MDS component. So Hadoop will handle the failure by re-schedule all failed workers to available slots. Hybrid pipeline with 80 tasks killed on 10 nodes uses slightly more time than 8 tasks killed on 1 node because of the scheduling overhead. With Twister-pipeline, since all the component are executed by Twister, when fault happens, the HyMR will restart the Twister-runtime at an average time of 267 seconds, then re-run the Twister job. Therefore, Twister-pipeline takes longer than Hybrid-pipeline to recover from failure. The Hybrid-pipeline also performs more stable than Twister-pipeline at different failure time where each core are scheduled with 2 map tasks where Twister can only handle 1 map task per core. So Twister takes longer time to recover then Hadoop when the time point increases within in a same component. Take an example at time point 55%, 70% and 85%, Twister-MI-MDS fails when the MI-MDS has been executed after 81, 334 and 587 seconds, the overall Twister-pipeline job execution time increases as well. In contrast, even though Hadoop-MI-MDS fails at these 3 different times, since the map task only takes half of a Twister map task time to finish, the job execution time doesn't increases much as in the Twister-pipeline. Last thing to mention is in time point 40%, as both of the pipelines were execution Twister-MDS, so the fail recover time for them are the similar, where Twister-MDS restarts after 173 seconds and HDFS-Twister-MDS restarts after 143 seconds.

5. Related Work

HyMR uses Hadoop to support MapReduce jobs and Twister to support iterative MapReduce jobs. Besides Twister, several other iterative MapReduce frameworks were developed over the past years. We chose Twister over the other iterative MapReduce runtimes because it is a light weighted fully fledged framework. Spark [27] is a MapReduce framework using memory cache to store the static data during iterative computations. It uses resilient distributed dataset (RDD) to provide fault tolerance. An RDD is a read-only collection of objects contains additional information from other tasks. Once a node fails, RDDs on other nodes can recover the failed RDD using resilient information. However, this approach uses additional memory on each compute node. It is not ideal to solve the memory bound problem such as MDS. Additionally, Spark is not a standalone runtime because it is based on another cluster manage system called Nexus [28]. Haloop [29] is an extension to the existing Hadoop system. Different from Twister, it uses local disk instead of memory to cache the static data during each iteration. Although it modified Hadoop job scheduler to avoid rescheduling overhead from each iteration, its map and reduce tasks still need to read and write to local disk in every iteration. Therefore, unlike Twister, it does not eliminate disk I/O cost. Pregel [30], an iterative graph processing framework proposed by Google, excels in processing iterative graph applications, such as PageRank, but lacks support for general iterative applications.

According to the taxonomy proposed by Yu et al. [31], current workflow systems can be divided into several categories. HyMR is a concrete workflow model since the resources has been allocated during an execution. Different from HyMR, most classic workflow system built on Grid adopts abstract model. Pegasus [32] maps complex scientific workflow into various Grid resources. It focused on optimizing available resources allocation by using certain artificial intelligence techniques. Kepler [33] models a workflow as independent components and communicates through well-defined interfaces. So adding more components into the workflow will not harm the integrity of existing ones. It also provides a graphic environment to design and reuse the Grid workflows. Taverna [34] allows automation of experimental methods through the use of a number of different (local or remote) services from a very diverse set of domains – biology, chemistry and medicine to music, meteorology and social sciences. It uses *XML* based language to describe the data model which includes control flow, data flow, input and processors. All above workflow systems can enable distributed computing. However, they are more focused on Grid resources allocation when a job needs to use multiple processors and they do not directly support MapReduce. In contrast, HyMR is focused on executing efficiently on available resources such as cluster and cloud by importing Hadoop and Twister.

There are already several existing MapReduce workflow systems. Sawzall [35] is a script language where its data processing built on top of Google MapReduce. Sawzall consists by an interpreter runs in the map phase to instantiate partitions over multiple nodes, a program which execute once for each partition and an aggregator accumulates the results from the program to a final output. It provides an efficient way to schedule data on multiple machines. Pig Latin [36] is similar to Sawzall, but based on Hadoop. It has extended features such as cogrouping, which can be used for join operation. As it is constructed to mimic the behavior of a relational database, it is more suitable for processing relational intermediate data inside the dataflow. Oozie [37] is a workflow system to manage data processing job for Hadoop. It supports MapReduce job, Hadoop Streaming, Pig and HDFS. Its design is similar to classical Grid workflow system but mainly process MapReduce job and store the data into HDFS. MRGIS [38] is also a scripting-based parallel programming tool, similar to the above systems. However, it is only specialized for GIS applications on MapReduce computing infrastructure. CloudWF [39] is a computational workflow system specially based on cloud infrastructure where Hadoop is installed. It aims to simplify the complication of running a general purpose distributed pipeline on MapReduce and take full potential advantages from the infrastructure underlying Cloud. Compare to these workflow systems, the design of HyMR is similar to Oozie, MRGIS and CloudWF. Decentralization of space and time is achieved by HyMR execution. The resources from cluster are also fully utilized by importing Twister and Hadoop. Additionally, none of the existing workflow system can directly support parallel iterative applications using MapReduce paradigm. HyMR can support a pipeline which includes iterative parallel applications more efficiently. Our previous work introduced Tigr-Twister-workflow [40] which merges Twister with Tigr-workflow [41] to support both iterative and non-iterative bioinformatics applications running on cluster with a web interface. But this system is less efficient since it lacks several features such as fault tolerance and distributed file system support.

6. Conclusion and Future Work

In this paper we designed and implemented a new distributed workflow system HyMR which is based on different MapReduce systems: Hadoop and Twister. The pros and cons of using these two runtimes are analyzed. The architecture of HyMR is described in detail. We explained how HyMR is extended to support both MapReduce and iterative MapReduce applications. A bioinformatics visualization pipeline is proposed to test HyMR. Several applications including an iterative application are included in this pipeline. A Twister-pipeline and Hybrid-pipeline is implemented and used to test the efficiency and fault tolerance of HyMR. In our experiments, we demonstrated that the Hybrid-

pipeline is more efficient than Twister-pipeline, which explains the motivation for our HyMR.

In the future, we would like to extend our system on more dynamic computing resources, such as cloud. A web server could be established on top layer so the user can submit distributed jobs using HyMR. Several other distributed computing frameworks such as Sector/Sphere could be added into HyMR to improve overall performance.

7. ACKNOWLEDGMENTS

Our thanks to Joe Rinkovsky as an administrator from UITS in Inidiana University that he gave us support while using Polar Grid for our experiments. And Saliya Ekanayake from SALSA-HPC group in Indiana University provided us the SWG library.This work is under National Health Institute Grant.

8. REFERENCES

[1] Isard, M., M. Budiu, et al. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. ACM SIGOPS Operating Systems Review, ACM Press. 41: 59-72.

[2] Gu, Y. and R. Grossman (2009). "Sector and Sphere: The Design and Implementation of a High Performance Data Cloud." Crossing boundaries: computational science, e-Science and global e-Infrastructure I. Selected papers from the UK e-Science All Hands Meeting 2008 Phil. Trans. R. Soc. A 367: 2429-2445.Tavel, P. 2007. *Modeling and Simulation Design*. AK Peters Ltd., Natick, MA.

[3] Dean, J. and S. Ghemawat (2008). "MapReduce: simplified data processing on large clusters." Commun. ACM 51(1): 107-113.

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

[5] Borthakur, D., J. SenSarma, et al. (2011). Apache Hadoop Goes Realtime at Facebook SIGMOD. Athens, Greece, ACM. 978: 4503-0661.

[6] Yang, H.-c., A. Dasdan, et al. Map-reduce-merge: simplified relational data processing on large clusters. Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM. Beijing, China, ACM.

[7] Ekanayake, J., S. Pallickara, et al. (2008). MapReduce for Data Intensive Scientific Analyses. Fourth IEEE International Conference on eScience, IEEE Press.

[8] Qiu X., Ekanayake J., et al. (2009). Using MapReduce Technologies in Bioinformatics and Medical Informatics. Using Clouds for Parallel Computations in Systems Biology workshop at SC09, Portland, Oregon.

[9] Dempster, A.P.; Laird, N.M.; Rubin, D.B. (1977). "Maximum Likelihood from Incomplete Data via the EM Algorithm". Journal of the Royal Statistical Society. Series B (Methodological) 39 (1): 1–38. JSTOR 2984875. MR0501537.

[10] J.Ekanayake, H.Li, et al. (2010). Twister: A Runtime for iterative MapReduce. Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010. Chicago, Illinois, ACM.

[11] TORQUE resource manager, Garrick Staples, SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ISBN 0-7695-2700-0

[12] "Hadoop Distributed File System HDFS." Retrieved December, 2009, from http://hadoop.apache.org/hdfs/.

[13] J. B. MacQueen, "Some Methods for Classification and Analysis of MultiVariate Observations," in Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability. vol. 1, L. M. L. Cam and J. Neyman, Eds., ed: University of California Press, 1967.

[14] K. Rose, E. Gurewwitz, and G. Fox, "A deterministic annealing approach to clustering," Pattern Recogn. Lett., vol. 11, pp. 589-594, 1990.

[15] J. de Leeuw, "Applications of convex analysis to multidimensional scaling," Recent Developments in Statistics, pp. 133-145, 1977.

[16] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain, "All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids," in IEEE Transactions on Parallel and Distributed Systems, 2010, pp. 33-46.

[17] O. Gotoh, "An improved algorithm for matching biological sequences," Journal of Molecular Biology vol. 162, pp. 705-708, 1982.

[18] Needleman, Saul B.; and Wunsch, Christian D. (1970). "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (3): 443–53. doi:10.1016/0022-2836(70)90057-4. PMID 5420325.

[19] Kruskal, J. B. and M. Wish (1978). Multidimensional Scaling, Sage Publications Inc.

[20] I. Borg, & Groenen, P. J., Modern Multidimensional Scaling: Theory and Applications: Springer, 2005.

[21] Bae, S.-H. (2008). Parallel Multidimensional Scaling Performance on Multicore Systems. Proceedings of the Advances in High-Performance E-Science Middleware and Applications workshop (AHEMA) of Fourth IEEE International Conference on eScience, Indianapolis: IEEE Computer Society.

[22] Bronstein, M. M., A. M. Bronstein, et al. (2006). Multigrid multidimensional scaling. Numerical Linear Algebra with Applications, Wiley.

[23] Kearsley, A. J., R. A. Tapia, et al. (1995). The Solution of the Metric STRESS and SSTRESS Problems in Multidimensional Scaling Using Newton's Method. Houston, Tx, Rice University.

[24] Zhang, B., Y. Ruan, et al. (2010). Applying Twister to Scientific Applications. Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference Indianapolis, IN, IEEE: 25-32.

[25] Seung-Hee Bae, J. Y. C., Judy Qiu, Geoffrey C. Fox (2010). Dimension Reduction and Visualization of Large High-dimensional Data via Interpolation. HPDC'10. Chicago, Illinois USA.

[26] Priebe, M. W. T. a. C. E. (2006). The Out-of-Sample Problem for Classical Multidimensional Scaling. Bloomington, IN, Indiana University.

[27] Zaharia, M., M. Chowdhury, et al. (2010). Spark: cluster computing with working sets. HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, Berkeley, CA, USA, ACM.

[28] Hindman, B., A. Konwinski, et al. (2009). Nexus: A Common Substrate for Cluster Computing.

[29] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. 2010. HaLoop: efficient iterative data processing on large clusters. Proc. VLDB Endow. 3, 1-2 (September 2010), 285-296.

[30] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 international conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 135-146. DOI=10.1145/1807167.1807184 http://doi.acm.org/10.1145/1807167.1807184

[31] Yu, J. and R. Buyya (2005). "A Taxonomy of Workflow Management Systems for Grid Computing." Journal of Grid Computing 3(3): 171-200.

[32] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. H. Su, K. Vahi, M. Livny. Pegasus: Mapping Scientific Workflow onto the Grid. Across Grids Conference 2004, Nicosia, Cyprus, 2004.

[33] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the KEPLER System. Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows, to appear, 2005

[34] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver and K. Glover, M.R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics, 20(17):3045-3054, Oxford University Press, London, UK, 2004.

[35] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming, 13(4):277–298, 2005.

[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD), pages 1099–1110. ACM, 2008.

[37] "Oozie." from http://yahoo.github.com/oozie/.

[38] Chen, Q., L. Wang, et al. (2008). MRGIS: A MapReduce-Enabled High Performance Workflow System for GIS. eScience '08. IEEE Fourth International Conference on. Indianapolis, IN.

[39] M.G. Jaatun, G. Zhao, and C. Rong (Eds.): CloudWF: A Computational Workflow System for Clouds Based on Hadoop. CloudCom 2009, LNCS 5931, pp. 393–404, 2009

[40] Hemmerich, C., A. Hughes, et al. (2010). Map-Reduce Expansion of the ISGA Genomic Analysis Web Server. CloudCom 2010. Indianapolis, IN.

[41] "TIGR-workflow." From: http://tigr-workflow.sourceforge.net/.