Twister: A Runtime for Iterative MapReduce

Jaliya Ekanayake[1,2], Hui Li[1,2], Bingjing Zhang[1,2], Thilina Gunarathne[1,2], Seung-Hee Bae[1,2], Judy Qiu[2], Geoffrey Fox[1,2]
[1]School of Informatics and Computing, [2]Pervasive Technology Institute
Indiana University Bloomington
{jekanaya, lihui, zhangbj, tgunarat, sebae, xqiu,gcf}@indiana.edu

ABSTRACT

MapReduce programming model has simplified the implementation of many data parallel applications. The simplicity of the programming model and the quality of services provided by many implementations of MapReduce attract a lot of enthusiasm among distributed computing communities. From the years of experience in applying MapReduce to various scientific applications we identified a set of extensions to the programming model and improvements to its architecture that will expand the applicability of MapReduce to more classes of applications. In this paper, we present the programming model and the architecture of *Twister* an enhanced MapReduce runtime that supports iterative MapReduce computations efficiently. We also show performance comparisons of Twister with other similar runtimes such as Hadoop and DryadLINQ for large scale data parallel applications.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming - *Distributed programming, Parallel Programming.*

General Terms

Algorithms, Performance, Languages

Keywords

MapReduce, Cloud Technologies, Iterative Algorithms.

1. INTRODUCTION

The data deluge is experiencing in many domains, and in some domains such as astronomy, particle physics and information retrieval, the volumes of data are already in peta-scale. The increase in the volume of data also increases the amount of computing power necessary to transform the raw data into meaningful information. In many such situations, the required processing power far exceeds the processing capabilities of individual computers, mandating the use of parallel/distributed computing strategies. These demanding requirements have led to the development of new programming models and implementations such as MapReduce[1] and Dryad[2].

MapReduce programming model has attracted a great deal of enthusiasm because of its simplicity and the improved quality of services that can be provided. Unlike the classical distributed processing runtimes in which the scheduling decisions are made mainly based on the availability of the computation resources, MapReduce takes a more data centered approach supporting the concept of "moving computations to data". There are many published work including some of ours [3-6], showing the applicability of MapReduce programming model to various data/compute intensive applications.

Classic parallel applications developed using message passing runtimes such as MPI[7] and PVM[8] utilize a rich set of communication and synchronization constructs offered by these runtimes to create diverse communication topologies. In contrast, MapReduce and similar high-level programming models support simple communication topologies and synchronization constructs. Although this limits their applicability to the diverse classes of parallel algorithms, in our previous papers [3-6] we have shown that one can implement many data/compute intensive applications using these high level programming models. When the volume of the data is large, algorithms based on simple communication topologies may produce comparable performances to the algorithms with tight synchronization constraints. These observations also favor MapReduce since its relaxed synchronization constraints do not impose much of an overhead for large data analysis tasks. Furthermore, the simplicity and robustness of these programming models supersede the additional overheads.

When analyzing a range of applications for which the MapReduce can be especially effective, we noticed that by supporting iterative MapReduce computations we can expand its applicability to more fields such as data clustering, machine learning, and computer vision where many iterative algorithms are common. In these algorithms, MapReduce is used to handle the parallelism while the repetitive application of it completes the iterations. Cheng Tao et al. also demonstrated ways of applying MapReduce to iterative machine learning algorithms[9].

There are some existing implementations of MapReduce such as Hadoop[10] and Sphere[11] most of which adopt the initial programming model and the architecture presented by Google. These architectures focus on performing single step MapReduce (computations that involve only one application of MapReduce) with better fault tolerance, and therefore store most of the data outputs to some form of file system throughout the computation. Furthermore, in these runtimes, the repetitive application of MapReduce creates new *map/reduce* tasks in each iteration loading or accessing any static data repetitively. Although these features can be justified for single step MapReduce computations, they introduce considerable performance overheads for many iterative applications.

Twister[12] is an enhanced MapReduce runtime with an extended programming model that supports iterative MapReduce computations efficiently. It uses a publish/subscribe messaging

infrastructure for communication and data transfers, and supports long running map/reduce tasks, which can be used in "configure once and use many times" approach. In addition it provides programming extensions to MapReduce with "broadcast" and "scatter" type data transfers. These improvements allow Twister to support iterative MapReduce computations highly efficiently compared to other MapReduce runtimes. We have published some of the preliminary results obtained during the development of Twister in few other publications [3-6]. In this paper, we discuss the extended MapReduce programming model and the architecture of Twister in detail. We also present some of the new applications that we have developed and their performances.

In the sections that follow, we first give an overview of the MapReduce programming model and the architecture used by most of the MapReduce runtimes. Section 3 introduces Twister programming model comparing it with the typical MapReduce followed by its architecture in section 4. In section 5, we present a set of applications that we have developed using Twister and provide a performance analysis comparing Twister with other parallel/distributed runtimes such as Hadoop, and DryadLINQ [13]. We discuss the related work to Twister in section 6, and in the final section we draw our conclusions and outline future works.

2. MAPREDUCE

2.1 Programming Model

MapReduce is a distributed programming technique proposed by Google for large-scale data processing in distributed computing environments. According to Jeffrey Dean and Sanjay Ghemawat, the input for MapReduce computation is a list of *(key,value)* pairs and each *map* function produces zero or more intermediate *(key,value)* pairs by consuming one input *(key,value)* pair. The runtime groups the intermediate *(key,value)* pairs based on some mechanism like hashing into buckets representing *reduce* tasks. The *reduce* tasks take an intermediate key and a list of values as input and produce zero or more output results [1].

Furthermore, because of its functional programming inheritance MapReduce requires both map and reduce tasks to be "side-effect-free". Typically, the *map* tasks start with a data partition and the *reduce* task performs operations such as "aggregation" or "summation". To support these, MapReduce also requires that the operations performed at the *reduce* task to be both "associative" and "commutative". These are common requirements for general reductions. For example, in MPI the default operations or the user defined operations in MPI_Reduce or MPI_Allreduce are also required to be associative and may also be commutative.

2.2 Architectures

Along with the MapReduce programming model, Jeffrey Dean and Sanjay Ghemawat describe in their paper the architecture that they adopted at Google. Many of their decisions are based on the scale of the problems that they solve using MapReduce and the characteristics of the large computing infrastructure in which these applications are deployed. Apache Hadoop and several other MapReduce runtimes such as Disco [14] and Sector/Sphere also adopt most of these architectural decisions. Below we will list some of the most important characteristics of their runtime as it will be useful to explain and compare the architectural decisions we made in Twister later.

2.2.1 Handling Input and Output Data

Both Google and Hadoop utilize distributed fault-tolerance file systems, GFS[15] and HDFS[10], in their MapReduce runtimes. These file systems use the local disks of the computation nodes to create a distributed file system, which can be used to co-locate data and computation. They also provide a large disk bandwidth to read input data. Both runtimes use the distributed file systems to read the input data and store output results. Moreover, these file systems are built-in with data duplication strategies so that they can recover from failures of individual local disks in data/compute nodes. (Note: in MapReduce domain a "node" typically refers to a computer that is used for both storing data and also for computation. Throughout the paper we also use the term node to refer such a computer).

2.2.2 Handling Intermediate Data

In most MapReduce runtimes the intermediate data produced after the *map* stage of the computation is first stored in local disks of the nodes where they are produced. Then the master scheduler assign these outputs to reduce workers, which will then retrieve the data via some communication protocol such as HTTP and later execute the *reduce* functions. This approach greatly simplifies the handling of failures in the runtime. However, it also adds a considerable performance overhead to the overall computation for some applications.

2.2.3 Scheduling Tasks

Google's MapReduce and Hadoop use a dynamic scheduling mechanism. In this approach, the runtime assigns *map/reduce* tasks to the available computation resources simplifying the optimal utilization of heterogeneous computational resources while the initial assignment of *map* tasks is performed based on the data locality. This approach also provides an automatic load balancing for *map* tasks with skewed data or computational distributions.

2.2.4 Fault Tolerance

Handling failures is one of the key considerations in Google's MapReduce architecture. Their approach of writing every data product to persistent storage simplifies the failure handling logic. In both Google and Hadoop MapReduce, the distributed file systems handle the failures of the disks or nodes using data replication. A failure of a *map* task is handled by rerunning the failed task while a failure of *reduce* task requires downloading the outputs of *map* tasks and re-execution of the *reduce* task. The master process that handles the scheduling and keeps track of the overall computation is assumed to run on a node that is less susceptible to failures. A failure in this node requires restarting of the overall runtime.

3. ITERATIVE MAPREDUCE WITH TWISTER

There are many parallel algorithms with simple iterative structures. Most of them can be found in the domains such as data clustering, dimension reduction, link analysis, machine learning, and computer vision. K-Means[16], Deterministic Annealing Clustering[17], pagerank[18], and dimension reduction algorithms such as SMACOF[19] are all examples of such algorithms. When analyzing algorithms like above, we noticed that the parallel sections of such algorithms can easily be implemented as MapReduce computations so that the overall computation becomes an iterative MapReduce computation.

Further analysis revealed some of the interesting characteristics such as; they utilize two types of data products – static and dynamic, use many iterations until convergence, some requires reduce output as a whole to make the decision to continue or stop iterations. These features demand an extended MapReduce programming model and an efficient runtime implementation, which we try to provide in Twister. In the following section, we discuss the programming extensions we support in Twister in more detail. Figure 1 shows the extended programming model.

3.1 Static vs. Variable Data

Many iterative applications we analyzed show a common characteristic of operating on two types of data products called static and variable data. Static data (most of the time the largest of the two) is used in each iteration and remain fixed throughout the computation whereas the variable data is the computed results in each iteration and typically consumed in the next iteration in many expectation maximization (EM) type algorithms. For example, if we consider K-means clustering algorithm[16], during the n^{th} iteration the program uses the input data set and the cluster centers computed during the $(n-1)^{th}$ iteration to compute the next set of cluster centers. To support *map/reduce* tasks operating with these two types of data products we introduced a "configure" phase for *map* and *reduce* tasks, which can be used to load (read) any static data at the *map* and *reduce* tasks. For example, the typical *map* phase of the computation then consumes the variable data specified as *(key, value)* pairs and the static data (already loaded) producing a set of output *(key, value)* pairs.

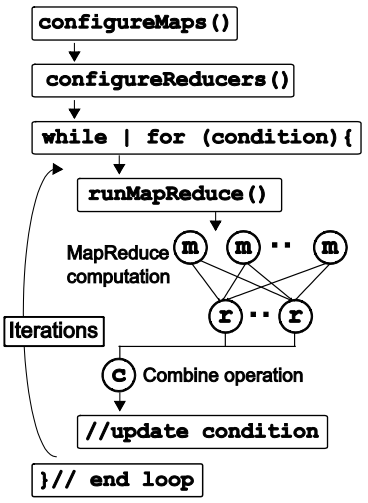

Figure 1. Iterative MapReduce programming model supported by Twister.

3.2 Long Running Map/Reduce Tasks

The above programming extension adds capabilities of handling both static and variable data in *map/reduce* tasks. However, reading static data in each execution of the MapReduce computation is highly inefficient. Although some of the typical MapReduce computations such as information retrieval consume very large data sets, many iterative applications we encounter operate on moderately sized data sets that can fit into the distributed memory of the computation infrastructure. This observation led us to explore the idea of using long-running *map/reduce* tasks similar to the parallel processes in many MPI applications that last throughout the life of the computation. The long running (cacheable) *map/reduce* tasks eliminates the necessity of reloading static data in each iteration. Current MapReduce implementations such as Hadoop and DryadLINQ do not support this behavior and hence they initiate new *map/reduce* tasks and load static data in each iteration incurring considerable performance overheads for iterative MapReduce computations. Although rare among iterative applications, one can use Twister with extremely large data sets that cannot be fit into the distributed memory of the computation infrastructure by reading data directly from the disks without loading them to memory.

3.3 Granularity of Tasks

The applications presented in the Google's MapReduce paper[1] used fine grained *map* tasks. For example, in word count application, the *map* tasks simply produce *(word, 1)* pairs for each word it encounter. However, for many applications we noticed that by increasing the granularity of the *map* task one can reduce the volume of the intermediate data. In the above example, instead of sending *(word, 1)* for every word, the *map* task can produce partial sums such as *(word, n)*. With the option of configurable *map* tasks, the *map* task can access large blocks of data/or files. In Twister, we adopt this approach in many of our data analysis applications to minimize the intermediate data volumes and to allocate more computation weight to *map* stage of the computation. Hadoop uses an intermediate combiner operation just after the *map* stage of the computation to support similar behavior.

3.4 Side-effect-free Programming

At first glance the concept of long-running *map/reduce* tasks seems to violate the "side-effect-free" nature of MapReduce allowing users to store state information in *map/reduce* tasks. However, in the case of a failure, Twister programming model only guarantees the restoring of static configurations such as data that can be reloaded using a data partition or static parameters shared from the main program. Any transient information stored in *map/reduce* tasks will be lost. Therefore the users of the Twister runtime can chose to use the fault tolerance capabilities by storing only the static configurations in long running *map/reduce* tasks or use the long running tasks to develop MapReduce applications with transient states stored in them (i.e. with side effects) without the fault tolerance capabilities.

3.5 Combine Operation

In Google's MapReduce architecture the outputs of the *reduce* tasks are stored in the distributed file system (GFS) in separate files. However, most iterative MapReduce computations require accessing the "combined" output of the *reduce* tasks to determine whether to proceed with another iteration or not. In Twister we have introduced a new phase to MapReduce named "Combine" that acts as another level of reduction (Note: this is different to the local combine operation that runs just after the *map* tasks in Hadoop). One can use the *combine* operation to produce a collective output from all the *reduce* outputs.

3.6 Programming Extensions

We have also incorporated a set of programming extensions to MapReduce in Twister. One of the very useful extensions is `mapReduceBCast(Value value)`. As the name implies this extension facilitates sending a single Value (Note: MapReduce uses *(key,value)* pairs) to all *map* tasks. For example, the "Value" can be a set of parameters, a resource (file or executable) name, or even a block of data. Apart from the above, the "configure" option described in section 3.1 is supported in Twister multiple ways. *Map* tasks can be configured using a "partition-file" – a file containing the meta-data about data partitions and their locations. In addition one can configure map/reduce tasks from a set of

values. For example `configureMaps(Value[]values)` and `configureReduce(Value[]values)` are two programming extensions that Twister provides. Twister also provides broadcast style operation between *map* and *reduce* phases allowing it to support complex parallel algorithms. We will discuss how these extensions are supported in the coming section.

4. TWISTER ARCHITECTURE

Twister is a distributed in-memory MapReduce runtime optimized for iterative MapReduce computations. It reads data from local disks of the worker nodes and handles the intermediate data in distributed memory of the worker nodes. All communication and data transfers are performed via a publish/subscribe messaging infrastructure. Figure 2 shows the architecture of the Twister runtime. (Note: we will simply use the term "broker network" to refer to the messaging infrastructure throughout the discussion).

Twister architecture comprises of three main entities; (i) client side driver (Twister Driver) that drives the entire MapReduce computation, (ii) Twister Daemon running on every worker node, and (iii) the broker network. During the initialization of the runtime, Twister starts a daemon process in each worker node, which then establishes a connection with the broker network to receive commands and data. The daemon is responsible for managing *map/reduce* tasks assigned to it, maintaining a worker pool to execute *map* and *reduce* tasks, notifying status, and finally responding to control events. The client side driver provides the programming API to the user and converts these Twister API calls to control commands and input data messages sent to the daemons running on worker nodes via the broker network.

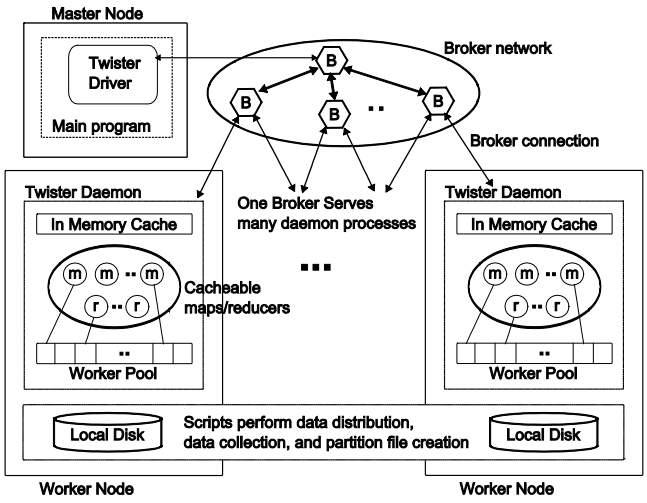

Figure 2. Architecture of Twister.

Twister uses a publish/subscribe messaging infrastructure to handle four types of communication needs; (i) sending/receiving control events, (ii) send data from the client side driver to the Twister daemons, (iii) intermediate data transfer between map and reduce tasks, and (iv) send the outputs of the reduce tasks back to the client side driver to invoke the combine operation. Currently, it supports NaradaBrokering[20] and ActiveMQ[21] messaging infrastructures. However, Twister architecture clearly separates the communication logics from the implementation of the other components so that it is straightforward to use other messaging infrastructures such as those are based on persistent queues.

4.1 Handling Input and Output Data

Twister provides two mechanisms to access input data for *map* tasks; (i) read data from the local disks of worker nodes and (ii) receive data directly via the broker network. The first option allows twister to start MapReduce computations using large data sets spread across the worker nodes of the computing infrastructure. Twister assumes that the data read from the local disks are maintained as files and hence supports file based input format, which simplifies the implementation of the runtime. The use of the native files allows twister to pass data files directly to any executable (may be a script running as a *map* or *reduce* computation) as command line arguments - a feature not possible with file systems such as HDFS. A possible disadvantage of this approach is that it does require the user to break up large data sets into multiple files.

The meta-data regarding the input file distribution across the worker nodes is read from a file called "partition-file". Currently, the partition file contains a list of tuples consisting of file_id, node_id, file_path,replication_no fields in them. The concept of the partition-file in Twister is inspired by the DryadLINQ's partitioned-file mechanism. Twister provides a tool to perform typical file system operations across the worker nodes such as (i) create directories, (ii) delete directories, (iii) distribute input files across worker nodes, (iv) copy a set of resources/input files to all worker nodes, (v) collect output files from the worker nodes to a given location, and (vi) create partition-file for a given set of data that is distributed across the worker nodes. Although these features do not provide the full capabilities that one can achieve via a distributed file system such as GFS or HDFS, the above services try to capture the key requirements of running MapReduce computations using the data read from local disks to support the concept of "moving computation to data". Integrating a distributed file system such as HDFS or Sector [11] with Twister is an interesting future work.

Twister also supports sending input data for *map* task directly via the broker network as well. It will be inefficient to send large volumes of input data via the broker network for map tasks. However, this approach is very useful to send small variable data (Note: please refer to the discussion of static vs. variable data in section 3.1) to map tasks. For example, a set of parameters, set of rows of a matrix, a set of cluster centers are all such data items.

4.2 Handling Intermediate Data

To achieve better performance, Twister handles the intermediate data in the distributed memory of the worker nodes. The results of the *map* tasks are directly pushed via the broker network to the appropriate *reduce* tasks where they get buffered until the execution of the *reduce* computation. Therefore, Twister assumes that the intermediate data produced after the map stage of the computation will fit in to the distributed memory. To support scenarios with large intermediate results, one can extend the Twister runtime to store the *reduce* inputs in local disks instead of buffering in memory.

4.3 Use of Pub/Sub Messaging

The use of publish/subscribe messaging infrastructure improves the efficiency of Twister runtime. However, to make the runtime scalable the communication infrastructure should also be scalable. NaradaBrokering messaging infrastructure we used in Twister can be configured as a broker network (as shown in figure 2), so that the Twister daemons can connect to different brokers in the network reducing the load on a given broker. This is especially useful when the application uses `mapReduceBcast()` with large data sets. A benchmark performed using 624 Twister daemons revealed that by using 5 brokers (connected hierarchically with 1 root broker and 4 leaf brokers) rather than 1

broker can improve the broadcast time by 4 folds for 20MB messages.

4.4 Scheduling Tasks

The cacheable *map/reduce* tasks used in Twister are only beneficial if the cached locations remain fixed. Therefore, Twister schedules *map/red*uce tasks statically. However, in an event of failure of worker nodes, it will reschedule the computation on different set of nodes. The static scheduling may lead to un-optimized resource utilization with skewed input data or execution times of the *map* tasks. However, one can minimize this effect by randomizing the input data assignment to the *map* tasks.

4.5 Fault Tolerance

Twister provides fault tolerance for iterative MapReduce computations. Our approach is to save the application state of the computation between iterations so that in the case of a failure the entire computation can be rolled back few iterations. Supporting individual map or reduce failures require adopting an architecture similar to Google, which will eliminate most of the efficiencies that we have gained using Twister for iterative MapReduce computations. Therefore, we decided to provide fault tolerance support only for iterative MapReduce computations in Twister based on the following three assumptions: (i) Similar to Google and Hadoop implementations, we also assume that the master node failures are rare and hence provide no support for master node failures and (ii) the communication infrastructure can be made fault tolerance independent of the Twister runtime, and (iii) the data is replicated among the nodes of the computation infrastructure. Based on these assumptions we try to handle failures of *map/reduce* tasks, daemons, and worker nodes failures.

The combine operation is an implicit global barrier in iterative MapReduce computations. This feature simplifies the amount of state Twister need to remember in case of a failure to recover. To enable fault tolerance Twister saves the configurations (information about the static data) used to configure *map/reduce* tasks before starting the MapReduce iterations. Then it also saves the input data (if any) that is sent directly from the main program to the *map* tasks. In case of a failure Twister simply re-configures *map/reduce* tasks using the available resources scheduling them based on the data locality, and restart the computation from the last saved state. If there are no replications of a particular data partition available among the remaining computation nodes the recovery operation will terminate.

5. APPLICATIONS AND PERFORMANCES

We have implemented a series of MapReduce applications using Twister, and the details of some of these applications have been presented in our previous publications [4-6]. Here we will describe some new applications that we have developed using Twister and provide a performance comparison with other MapReduce runtimes such as Hadoop and DryadLINQ. For performance analysis we used two computation clusters as follows.

Table 1. Details of the computation clusters used.

Cluster ID	Cluster-I	Cluster-II
# nodes	32	230
# CPUs in each node	6	2
# Cores in each CPU	8	4
Total CPU cores	768	1840
CPU	Intel(R) Xeon(R) E7450 2.40GHz	Intel(R) Xeon(R) E5410 2.33GHz
Memory Per Node	48GB	16GB
Network	Gigabit	Gigabit

Cluster-I can be booted in to both Linux (Red Hat Enterprise Linux Server release 5.4 -64 bit) and Windows (Windows Server 2008 -64 bit) while the Cluster-II runs Red Hat Enterprise Linux Server release 5.4 -64 bit operating system. We use the academic release of DryadLINQ, Apache Hadoop version 0.20.2, and Twister for our performance comparisons. Both Twister and Hadoop use JDK (64 bit) version 1.6.0_18, while DryadLINQ and MPI uses Microsoft .NET version 3.5.

5.1 Pairwise Distance Calculation

Calculating similarity or dissimilarity between each element of a data set with each element in another data set is a common problem and is generally known as an All-pairs[22] problem. The application we have selected calculates the Smith Waterman Gotoh(SW-G)[23] distance (say δ_{ij} –distance between gene i and gene j) between each pair of genes in a given gene collection.

We mapped the above application to the MapReduce programming model by adopting a coarse grain task decomposition approach. To clarify our algorithm, let's consider an example where N gene sequences produces a pairwise distance matrix of size NxN. We decompose the computation task by considering the resultant matrix and group the overall computation into a block matrix of size DxD. Due to the symmetry of the distances δ_{ij} and δ_{ji} we only calculate the distances in the blocks of the upper triangle of the block matrix as shown in Figure 3 . The blocks in the upper triangle are used as the values for *map* tasks along with the block coordinates as the keys. Once *maps* calculate the SW-G distance for a given block, it will emit two copies of the resulting matrix of distances corresponding to the results for the current block (i,j) and the block (j,i) due to symmetry. The block (j,i) is marked to read as a transpose matrix. The row number of a given block is used as the input key for the *reduce* tasks, which simply collect the data blocks corresponding to a row and write to output files after organizing them in their correct order. At the end of the computation all the blocks corresponding to a single row block will be written to a data file by the *reduce* tasks. We have developed three implementations of the same application using Twister, Hadoop, and DryadLINQ runtimes. In both Hadoop and Twister programs the calculation of the SW-G distance is done using the JAligner[24] program, a java implementation of the NAligner[24] program which we have used in DryadLINQ. More information about these implementations can be found in [25].

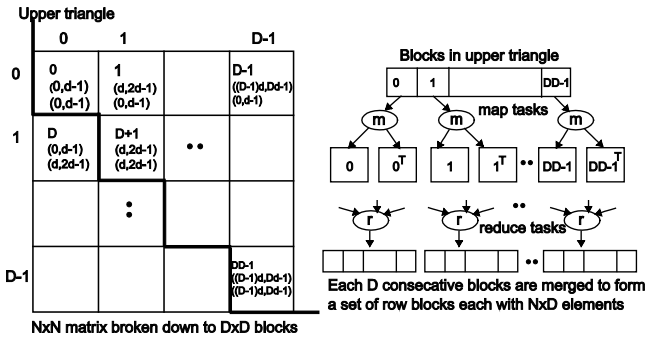

Figure 3. Twister implementation of the SW-G distance calculation program.

SW-G distance calculation is a typical MapReduce computation similar to the "word-count" or "grep" applications. However, unlike those synthetic applications, the SW-G performs considerable amount of computation at the *map* task and transfer a large matrix as the intermediate results between *map* and *reduce* phases. We use this application to demonstrate the ability of Twister to support typical MapReduce computations although the runtime is optimized for iterative MapReduce computations. A High Energy Physics data analysis that belongs to the same class of applications was explained in our previous work[4].

We identified samples of the human and Chimpanzee Alu gene sequences using Repeatmasker[26] with Repbase Update [27] and produced a data set of 50000 genes replicating a random sample of 10000 genes from the original data. We used this data set to measure parallel performance of DryadLINQ, Hadoop, and Twister runtimes. Figure 4 shows the parallel efficiency (η) of each runtime under varying data sizes calculated using the following formula in which p is the number of parallel units, $T(p)$ is the running time with p parallel units, and $T(1)$ is the sequential running time.

$$Parallel\ Efficiency\ (\eta) = \frac{T(1)}{p.T(p)} \qquad (1)$$

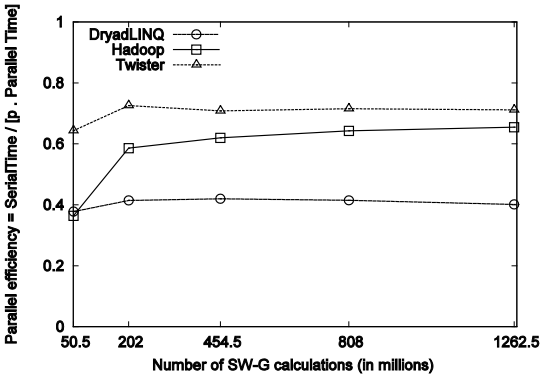

Figure 4. Parallel Efficiency of the different parallel runtimes for the SW-G program (Using 744 CPU cores in Cluster-I).

For the above calculation, we estimated the serial running time by simply summing up the times spent on each map and reduce tasks. The results clearly show that all three runtimes achieve maximum efficiencies and maintains them with the increase of data. Although the absolute efficiency is not correctly reflected by the estimated serial time, it provides a valuable base point for our comparisons. Since this is a typical MapReduce computation, we expect all runtimes to achieve higher absolute efficiencies. Twister outperforms Hadoop, because of its faster data communication mechanism, and the lower overhead in the static task scheduling. Moreover, in Hadoop each map/reduce task is executed as separate process (Java Virtual Machine - JVM) where as Twister uses a hybrid approach in which the map/reduce tasks assigned to a given daemon is executed within one JVM. The Lower efficiency in DryadLINQ was mainly due to an inefficient task scheduling mechanism used in the initial academic release[3].

To evaluate the scalability of the Twister runtime further, we performed another benchmark using 1632 CPU cores of Cluster-II. In this evaluation, the Twister runtime is configured to use a daemon in each CPU core simulating a cluster of 1632 single core nodes. The efficiencies calculated for this evolution shows a value of 79% indicating that the runtime is scalable to such number of nodes. These results also prove that Twister is capable of running typical MapReduce computations although we have added enhancements focusing on iterative MapReduce computations.

5.2 Multidimensional Scaling

Multidimensional scaling (MDS) is a general term for the techniques to configure low dimensional mappings of given high-dimensional data with respect to the pairwise proximity information, while the pairwise Euclidean distance within the target dimension of each pair is approximated to the corresponding original proximity value. In other words, it is a non-linear optimization problem to find low-dimensional configuration which minimizes the objective function, called STRESS[28] or SSTRESS [29].

Among many MDS solutions, we are using a well-known EM-like method called SMACOF (Scaling by Majorizing of COmplicated Function)[19] in this paper. SMACOF is based on iterative majorization approach and is calculated by iterative matrix multiplication. For the stop condition, SMACOF algorithm measures the STRESS value of current mapping and compare to the STRESS value of the previous mapping result. If the difference of STRESS value between previous one and the current one is smaller than threshold value, then it stops iteration. For details of the SMACOF algorithm, please refer to[30].

We implemented the above algorithm using Twister and evaluated its performance and scalability characteristic. As we have shown in [3, 4] both Hadoop and DryadLINQ showed extremely high overheads for iterative applications such as K-Means clustering or matrix multiplication. The MDS uses three MapReduce computations in a single iteration involving two matrix- vector multiplications and one STRESS calculation. Thus we expect both Hadoop and DryadLINQ to be highly inefficient for this application and hence did not implement MDS using those runtimes. To evaluate the performance of our implementation, we used a data set comprising of 35339 genes producing 1.24 billion pair-wise distances. Estimating the serial running time for MDS application is not straightforward and hence we calculated the parallel efficiency using the formula (2) below in which $\alpha = p1/p2$ and $p2$ is the smallest number of CPU cores for the experiment, so $alpha \geq 1$. This will calculate the parallel efficiency with respect to the minimum number of CPU cores used for the experiment. The outcome of this benchmark is shown in Figure 5.

$$Parallel\ Efficiency\ (\eta) = \frac{T(p2)}{\alpha.T(p1)} \qquad (2)$$

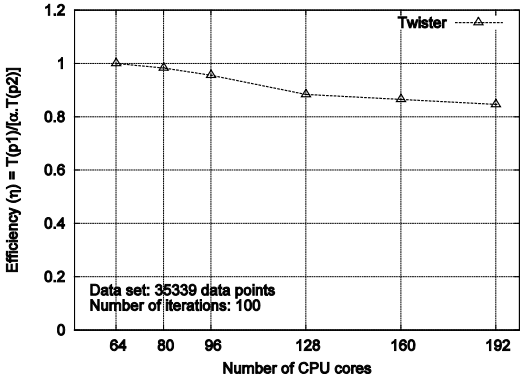

Figure 5. Efficiency of the MDS application (in Cluster–II).

For the selected data set, Twister maintains higher efficiencies (>80%) for considerable number of CPU cores. With large data,

we expect it to maintain similar efficiencies for even higher number of CPU cores.

5.3 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 δ is the vector distance between the page access probabilities in N^{th} 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[31, 32]. In this paper we do not create any new PageRank algorithm, but implement the most general RageRank algorithm [33] 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) used the broadcast feature to send input data (variable data) the *map* tasks. 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.

We investigated Twister PageRank performance using ClueWeb data set [34] collected in January 2009. We built the adjacency matrix using this data set and tested the page rank application using 32 computer nodes of Cluster-II. Table 2 summarizes the characteristic of three ClueWeb data sets we used in our tests.

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

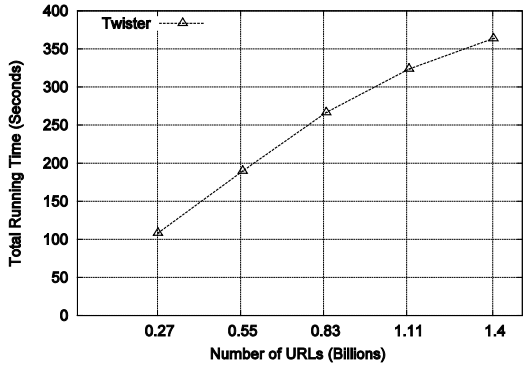

Figure 6. Total running rime for 20 iterations of the Pagerank implementation (Using 256 CPU cores in Cluster-II).

Figure 6 shows the scalability of the pagerank application under different data sizes. We also calculated the efficiency of the PageRank application using formula (2) above with p1 and p2 times are taken from runs on 128 and 256 CPU cores respectively for the CWDS3 data set. The results revealed that the Twister version of the application can maintain above 80% efficiency at 256 CPU cores as well.

6. RELATED WORK

MapReduce simplifies the programming of many pleasingly parallel applications. Currently, there are several MapReduce implementations available based on the Google's MapReduce architecture and some of which have improvements/features over the initial MapReduce model proposed by Google. However for our knowledge there are no other implementations that support features such as long running *map/reduce* tasks or the MapReduce extensions to support iterative MapReduce computations efficiently for large-scale data analysis applications as in Twister.

The paper presented by Cheng-Tao et al. discusses their experience in developing a MapReduce implementation for multi-core machines[9]. They used the MapReduce runtime to implement several machine learning algorithms showing that MapReduce is especially effective for many algorithms that can be expressible in certain "summation form". Phoenix runtime, presented by Colby Ranger et al., is a MapReduce implementation for multi-core systems and multiprocessor systems [35]. The evaluations used by Ranger et al. comprises of typical use cases found in Google's MapReduce paper such as word count, reverse index and also iterative computations such as Kmeans. Some of our design decisions in Twister were inspired by the benefits obtained in these shared memory runtimes. For example, in the above runtimes the data transfer simply requires sharing memory references, in Twister we use distributed memory transfers using pub/sub messaging. Sending some data value to all map tasks is a trivial operation with shared memory, in Twister we introduced `mapReduceBcast()` to handle such requirements.

Sphere[11] is a parallel runtime that operates on Sector[11] distributed file system. Sector is similar to HDFS in functionality; however it expects the data to be stored as files and leaves the data splitting for the users to manage. Unlike *map/reduce* Sphere executes user defined functions on these data splits. The authors show that it can also be used to execute MapReduce style computations as well. However we noticed that their approach requires more user involvement in managing the computations. Supporting MapReduce in various programming languages is a motivation in many map reduce runtimes such as Disco, Qizmt, and Skynet.

Parallel runtimes that support Directed Acyclic Graph (DAG) based execution flows provide more parallel topologies compared to the MapReduce programming model. Condor DAGMan [36] is a well-known parallel runtime that supports applications expressible as DAGs. Many workflow runtimes supports DAG based execution flows as well. In these runtimes the parallel task can read from several input sources and produce one or more outputs. Typically, the granularity of the computations executed in these tasks is larger than the granularity of the computations performed in *map/reduce* functions in MapReduce. For example, in workflow runtimes a task can be a separate parallel program running on multiple computers. One can simulate the DAGs using MapReduce by orchestrating multiple MapReduce computations. In this regard, Twister will support it better due to its capabilities to send input data directly from the main program

to map/reduce tasks and collect the reduce outputs back to the main program.

Microsoft Dryad[2] also uses a DAG based execution model in its distributed execution engine. The task granularity in vertices in Dryad is more similar to the task granularity of *map/reduce* in MapReduce and hence the authors call it a superset of MapReduce. Extending the Twister runtime to execute more general user defined functions and DAG based execution flows is an interesting future work. However, programming such a runtime is not straightforward and that could be the very reason why Microsoft introduced DryadLINQ.

DryadLINQ provides a LINQ [37] based programming API for Dryad and hence it is more suitable for applications that process structured data. Performing computations that involve legacy applications or scripts using DryadLINQ is not so straightforward. DryadLINQ also supports "loop unrolling" a feature that can be used to create aggregated execution graphs combing a few iterations of iterative computations. The number of iterations that can be unrolled depends on the application and the available memory in the machine. Typically it is only a few iterations. Therefore, as we have shown in our previous paper[3] it does not reduce the overhead of the programming model for iterative applications. Iterative applications we have tested perform up to 10,000 iterations even in our initial modest size problems. They benefit greatly from the long running map/reduce computation tasks in Twister. Furthermore, DryadLINQ also uses file based communication mechanism to transfer data incurring higher overheads.

Swift [38] is a scripting language and an execution and management runtime for developing parallel applications with the added support for defining typed data products via schemas. Its main focus is expressing computations with simple parallel structures that are coupled with data partitions more easily and scheduling those using Grid/Cluster infrastructures. Once a data partition is available one can easily uses MapReduce to schedule a "map-only" operation (Twister also support this) to process them as a many task computation without using the full MapReduce cycle. Swift also support iterative execution of parallel tasks, but does not provide optimizations such as long running tasks or faster data transfers as in Twister.

There is a rich set of future research topics examining additional features for Twister based on lessons from the other projects discussed in this section.

7. CONCLUSIONS AND FUTURE WORK

In this paper we discussed our experience in designing and implementing Twister - a distributed in-memory MapReduce runtime optimized for iterative MapReduce computations. We have discussed the extended programming model of Twister and its architecture comparing them with the typical MapReduce and its current architectures showing how Twister extends the envelop of MapReduce to more classes of applications. We have also presented the results of a set of applications with voluminous data sets. Some of the benchmarks performed with Twister use a 1632 CPU core cluster. The results, including some of the complex iterative applications such as MDS, indicate that Twister performs and scales well for many iterative MapReduce computations

We plan to extend our future research in three areas; (i) research on different communication infrastructures that can be used with Twister and identify ways to reduce the load on messaging infrastructure, (ii) explore the possible distributed file systems that can be incorporated with Twister to provide better data handling capabilities and better fault tolerance while retaining most of the efficiencies we have in Twister intact, and (iii) extending the programming model further to support more classes of applications. With the above enhancements, Twister will provide a valuable tool for MapReduce that supports data-intensive disciplines such as physics, chemistry and the medical and life sciences as well.

8. REFERENCES

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

[2] 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.

[3] J. Ekanayake, A. Balkir, T. Gunarathne, G. Fox, C. Poulain, N. Araujo, and R. Barga, "DryadLINQ for Scientific Analyses," presented at the 5th IEEE International Conference on e-Science, Oxford UK, 2009.

[4] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for Data Intensive Scientific Analyses," presented at the Proceedings of the 2008 Fourth IEEE International Conference on eScience, 2008.

[5] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. Fox, "High Performance Parallel Computing with Clouds and Cloud Technologies," in *Cloud Computing and Software Services: Theory and Techniques*, ed: CRC Press (Taylor and Francis).

[6] G. Fox, S.-H. Bae, J. Ekanayake, X. Qiu, and H. Yuan, "Parallel Data Mining from Multicore to Cloudy Grids," presented at the International Advanced Research Workshop on High Performance Computing and Grids (HPC2008), Cetraro, Italy, 2008.

[7] *MPI (Message Passing Interface)*. Available: http://www-unix.mcs.anl.gov/mpi/

[8] *PVM (Parallel Virtual Machine)*. Available: http://www.csm.ornl.gov/pvm/

[9] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," in *NIPS*, ed: MIT Press, 2006, pp. 281-288.

[10] *Apache Hadoop*. Available: http://hadoop.apache.org/

[11] Y. Gu and R. L. Grossman, "Sector and Sphere: the design and implementation of a high-performance data cloud," *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, vol. 367, pp. 2429-2445, 2009.

[12] *Twister:A Runtime for Iterative MapReduce*. Available: http://www.iterativemapreduce.org/

[13] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and C. J., "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," in *Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[14] *Disco project*. Available: http://discoproject.org/

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 29-43, 2003.

[16] 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.

[17] K. Rose, E. Gurewwitz, and G. Fox, "A deterministic annealing approach to clustering," *Pattern Recogn. Lett.*, vol. 11, pp. 589-594, 1990.
[18] S. Brin and L. Page. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Available: http://infolab.stanford.edu/~backrub/google.html
[19] J. de Leeuw, "Applications of convex analysis to multidimensional scaling," *Recent Developments in Statistics*, pp. 133-145, 1977.
[20] S. Pallickara and G. Fox, "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids," presented at the Middleware 2003, 2003.
[21] *ActiveMQ*. Available: http://activemq.apache.org/
[22] 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.
[23] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology* vol. 162, pp. 705-708, 1982.
[24] *Source Code. Smith Waterman Software*. Available: http://jaligner.sourceforge.net/
[25] J. Qiu, J. Ekanayake, T. Gunarathne, J. Y. Choi, S.-H. Bae, Y. Ruan, S. Ekanayake, S. Wu, S. Beason, G. Fox, M. Rho, and H. Tang, "Data Intensive Computing for Bioinformatics," in *Data Intensive Distributed Computing*, ed: IGI Publishers, 2010.
[26] A. F. A. Smit, R. Hubley, and P. Green. (2004, *Repeatmasker*. Available: http://www.repeatmasker.org
[27] J. Jurka, "Repbase Update:a database and an electronic journal of repetitive elements," *Trends in Genetics*, vol. 6, pp. 418-420, 2000.
[28] J. Kruskal, "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis," *Psychometrika*, vol. *29*, pp. 1-27, 1964.
[29] Y. Takane, Young, F. W., & de Leeuw, J., "Nonmetric individual differences multidimensional scaling: an alternating least squares method with optimal scaling features," *Psychometrika*, vol. *42*, pp. 7-67, 1977.
[30] I. Borg, & Groenen, P. J., *Modern Multidimensional Scaling: Theory and Applications*: Springer, 2005.
[31] 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.
[32] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub, "Exploiting the Block Structure of the Web for Computing PageRank," Stanford InfoLab, Technical Report2003.
[33] *The Power Method*. Available: http://en.wikipedia.org/wiki/Pagerank#Power_Method
[34] (2009, *The ClueWeb09 Dataset*. Available: http://boston.lti.cs.cmu.edu/Data/clueweb09/
[35] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *13th International Symposium on High-Performance Computer Architecture*, 2007, pp. 13-24.
[36] C. Team. (2009, *Condor DAGMan*. Available: http://www.cs.wisc.edu/condor/dagman/.
[37] *LINQ Language-Integrated Query*. Available: http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
[38] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. v. Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," in *IEEE Congress on Services*, 2007, pp. 199-206.