Applying Twister to Scientific Applications

Bingjing Zhang[1, 2], Yang Ruan[1, 2], Tak-Lon Wu[1, 2], Judy Qiu[1, 2], Adam Hughes[2], Geoffrey Fox[1, 2]
[1]School of Informatics and Computing, [2]Pervasive Technology Institute
Indiana University Bloomington
{zhangbj, yangruan, taklwu, xqiu, adalhugh, gcf}@indiana.edu

Abstract—**Many scientific applications suffer from the lack of a unified approach to support the management and efficient processing of large-scale data. The Twister MapReduce Framework, which not only supports the traditional MapReduce programming model but also extends it by allowing iterations, addresses these problems. This paper describes how Twister is applied to several kinds of scientific applications such as BLAST, MDS Interpolation and GTM Interpolation in a non-iterative style and to MDS without interpolation in an iterative style. The results show the applicability of Twister to data parallel and EM algorithms with small overhead and increased efficiency.**

Keywords-Twister; Iterative MapReduce; Cloud; Scientific Applications

I. Introduction

Scientific applications are required to process large amounts of data. In recent years, typical input data sets have grown in size from gigabytes to terabytes, and even petabyte-scale input data is becoming more common. These large data sets already far exceed the computing capability of one computer, and while the computing tasks can be parallelized on several computers, the execution may still take days or weeks to complete.

This situation demands better parallel algorithms and distributed computing technologies which can manage scientific applications efficiently. The MapReduce Framework [1] is one such kind technology which has become popular in recent years. KeyValue pairs make the input be distributed and processed in parallel at a fine level of granularity. The combination of Map tasks and Reduce tasks satisfies the task flow of most kinds of applications, and these tasks are also well managed under the runtime platform.

This paper introduces the Twister MapReduce Framework [2], an expansion of the traditional MapReduce Framework. The main characteristic of Twister is that it supports not only non-iterative MapReduce applications but also an iterative MapReduce programming model to efficiently support Expectation-maximization (EM) algorithms that suffer from communication complications. These algorithms are common in scientific applications but are not well handled by previous MapReduce implementations such as Hadoop [3].

Twister uses a publish/subscribe messaging middleware system for command communication and data transfers. It supports MapReduce in the manner of "configure once, and run many time" [2]. Data can be easily scattered from the client node to compute nodes and combined back into client node through Twister's API. With these features, Twister supports iterative MapReduce computations efficiently when compared to other MapReduce runtimes. Twister can be applied to Cloud architecture, having been successfully deployed on the Amazon EC2 platform [4].

The main focus of this paper is on the applicability of Twister to scientific problems, as demonstrated through the implementation of several scientific applications. In the following sections, an overview of Twister is first presented, introducing its programming model and architecture. Then, four scientific applications implemented using Twister are discussed. Three of these applications are non-iterative programs (Twister BLAST, Twister GTM Interpolation, and Twister MDS Interpolation), while the final one is Twister MDS, an iterative application. Workflow and the parallel mechanism supported by Twister are also presented within this section. Finally, conclusions based on this work are presented in the last section.

II. Twister Overview

This section gives an overview to Twister MapReduce Framework. Twister's support of non-iterative and iterative MapReduce programming models is discussed first, followed by a description of the Twister architecture.

A. Non-Iterative and Iterative MapReduce Support

Many parallel applications are only required to perform Map and Reduce tasks once, such as WordCount [1]. However, some other applications such as Kmeans [5] and PageRank [6], operate in an iterative pattern. Their parallel algorithms require the program to perform Map and Reduce tasks in iterations in order to get the final result.

The basic idea of Twister is to allow users to configure a MapReduce job only once, and then to run the job in one iteration or several iterations according to the client's request. If one round of Map and Reduce tasks is required, then the job executes exactly as it would using non-iterative MapReduce, and the result is produced directly from the Reduce method. For iterative MapReduce, the output from the "Reduce" is collected by a "Combine" method at the end of each iteration. A client will send intermediate results back to the compute nodes as new set of KeyValue pairs in the next iteration of MapReduce tasks (See Fig. 1).

Another important characteristic of many iterative algorithms is that some sets of input data remain static between iterations. In Twister, these static data are configured with a partition file, loaded into Map or Reduce tasks, and then reused through iterations. This mechanism significantly improves the performance of Twister for iterative MapReduce computing and makes it different from those methods, which mimic iterative MapReduce by simply re-executing MapReduce tasks without caching and reusing data or job configuration. In addition, because the data cached inside of Map and Reduce tasks are static, Twister maintains a "side-effect-free" nature [2].

In this workflow, Twister also provides a fault tolerance solution for the iterative MapReduce programming model. Twister can save the execution state between iterations, and, if execution faults are detected, roll back a few iterations and resume computing.

B. Architecture

Twister has several components. The client side is used to drive MapReduce jobs. Daemons and workers which live on the compute nodes manage MapReduce tasks. Connections between the components are based on SSH and messaging software.

The client controls MapReduce jobs through a multi-step process. During configuration, the client assigns MapReduce methods to the job, prepares KeyValue pairs and prepares static data for MapReduce tasks through the partition file if required. Once the job is configured, the client can spawn the MapReduce job and monitor it until completion. Between iterations, the client receives results collected by the Combine method, and, when the job is done, exits gracefully.

Messages including control messages and KeyValue pair data are transmitted through a network of message brokers via a publish/subscribe mechanism. With a set of predefined interfaces, Twister can be assembled with different messaging software. Currently Twister supports two of message brokers: NaradaBrokering [7] and ActiveMQ [8].

Daemons operate on compute nodes, loading the Map and Reduce classes and starting them as Map and Reduce workers, also known as Mappers and Reducers. During initialization, Map and Reduce workers load static data from the local disk according to records in the partition file and cache the data into memory. The workers then execute a Map or Reduce function defined by the users. Twister uses static scheduling for workers in order to take advantage of the local data cache [2]. In this hybrid computing model, worker threads are managed by daemon processes on each node, while, between nodes, daemons communicate with the client through messages.

Twister uses scripts to operate on static input data and some output data on local disks in order to simulate some characteristics of distributed file systems. In these scripts, Twister parallel distributes static data to compute nodes and create partition file by invoking Java classes. For data which are output to the local disks, Twister uses scripts to gather data from all compute nodes on a single node specified by the user.

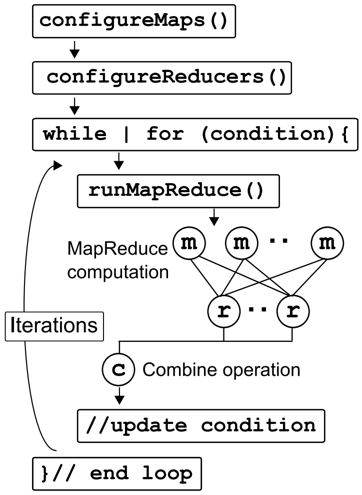

Figure 1. Twister MapReduce workflow [2]

III. Twister Non-Iterative Applications

Twister can support non-iterative applications which exhibit the "Map and then Reduce" or "Map only" pattern. "Map and then Reduce" is a common case in traditional MapReduce programming models. A classic example of this model is WordCount. In this case, every Map task calculates the word count in local partial text and sends the intermediate results to Reduce tasks with the word as Key and the count as Value. Then Reduce tasks collect the partial result and calculate the total count of one word. Meanwhile, "Map only" means data are processed by parallel Map tasks and then output directly. This parallelism method is also frequently used.

In "Map only" style applications, parallelization is commonly achieved by distributing input data sets across several compute nodes and then executing the same stand-alone version of a program on each node. This method, often called binary invoking mode, is often used in data parallel computations for several reasons. Many modern stand-alone scientific programs are complex and updated frequently with new features. Rewriting the parallel version of such a stand-alone program may require too much effort to keep pace with required new features. Consequently, binary invoking has become a viable solution in many cases. The MapReduce framework makes this solution accessible to many applications because it is well suited to handling input data distribution and managing parallel task execution.

Three new non-iterative MapReduce applications, including Twister BLAST, Twister MDS Interpolation, and Twister GTM Interpolation, are introduced in the following sections.

A. Twister BLAST

Twister BLAST is a parallel BLAST application based on the Twister MapReduce framework. A brief introduction to the BLAST software and other parallel BLAST applications, along with a discussion of the characteristics of Twister BLAST, are presented below. Finally, a performance comparison of Twister BLAST and Hadoop BLAST, with detailed analysis, is described.

1) *BLAST Software*

BLAST [9] is a stand-alone local gene search tool which exists in two versions. The original BLAST is written in C, while a newer version, BLAST+ is written in C++. BLAST+ is the version recommended by NCBI, and, consequently, the term BLAST used below refers to BLAST+. The version used here is 2.2.23.

BLAST is a command line tool which accepts input parameters and outputs the result to the screen or a file after its execution. BLAST jobs require two sets of input, namely a query location and a database location [10]. The BLAST query is a file which contains FASTA-format gene sequences which will be searched against the specified input database. The BLAST database is a set of formatted files which contain gene data organized with indices. The total size of the database is usually large, often on the order of gigabytes. A BLAST search consists of three phases [11]. The first phase is the "Setup", during which the query is read into memory and a "lookup" table is built. The next phase is the "Scanning" step, in which each subject sequence in the database is scanned for words matching the query in the "lookup" table. The final phase is the "Trace-back" step, when improved score and insertions/deletions are calculated for query sequences.

A typical BLAST job is very demanding in its use of a system. On an IU PolarGrid [12] node with two 4-core CPUs (Intel Xeon CPU E5410 2.33GHz) and 16 GB memory, searching hundreds of gene sequences with 37 gene letters each through a 10 GB NR database [13], BLAST consumes 100% of one core's CPU and 20% of its total memory under the one-thread mode. Such a job can exhaust all memory on a machine if the input is too large or if there are too many hits to the database [10].

BLAST can also be executed under a multi-thread mode. Under this mode, BLAST can utilize multiple cores but still uses 20% of total memory. However, the BLAST job does not fully utilize eight cores during the entire run. For example, on the node with settings mentioned above but executing BLAST with 8 threads, CPU usage is not always 800% but occasionally dropped down. The reason is that BLAST is only multi-threaded in its "Scanning" stage. The chart below shows the execution time comparison and the speedup of using 8-thread mode with different input sizes. The speedup value is greatly affected by database loading time when the input size is small and then become stable as the input size grows larger than 100 sequences. However, all of the speedup values are below 7.8, which is still less than 8. This means using multi-thread mode will not be as efficient as multi-process mode in the case that the node can provide enough memory for the execution of multiple BLAST processes (See Fig. 2).

2) *BLAST Parallelism Method*

Several kinds of parallel BLAST applications have already been implemented, including MPI BLAST [14], Cloud BLAST [15], and Azure BLAST [16]. This section will introduce these technologies through a timeline.

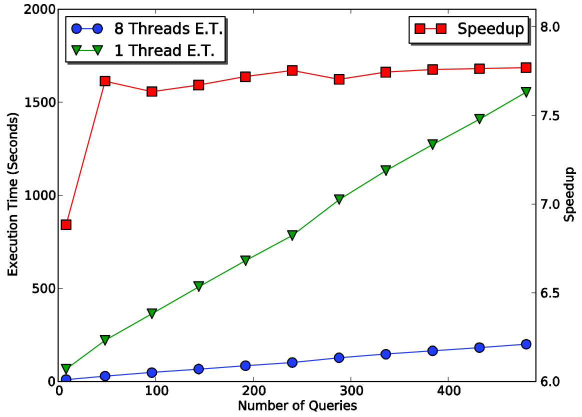

Figure 2. Execution time and speedup between 1 thread and 8 threads on one PG node under various input size

MPI BLAST uses the MPI library [17] to support parallel BLAST computations. It modifies the original BLAST program through a combination of the NCBI toolkit [18] and the MPI library. The query and the database are both partitioned. Once MPI BLAST starts, it distributes the database partitions to all compute nodes, and then uses one process to dynamically schedule query chunks to different workers. Because of the database segmentation, every worker cannot produce a complete output. As a result, one process is used to merge the result and output it to a shared directory. The database is segmented because the MPI BLAST designers believe that the database is too large to fit into memory or even to be stored on local disk [19]. However, database segmentation also generates substantial communication work and modern clusters have large memory and disks which can easily hold gigabyte-level database volumes. In addition, the latest version of MPI BLAST is based on an older version of BLAST which may lack new features and performance enhancements. Furthermore, MPI BLAST doesn't have fault tolerance support, which is a serious limitation because BLAST jobs usually require long execution time.

Cloud BLAST uses the Hadoop MapReduce Framework to support parallel BLAST on cloud platforms. Hadoop is used here for resolving issues like data splitting, finding worker nodes, load balancing, and fault tolerance [15]. MapReduce computing is used in "Map only" style, and original data are split into small chunks and distributed to workers. On each node, input data chunks are processed by invoking the BLAST binary and searching through a local database copy. The outputs are stored in HDFS [20]. Due to this computing style, Cloud BLAST has a lower communication cost. It has been proved that this kind of architecture performs better than MPI BLAST, as well as being scalable, maintainable and fault tolerable [15].

Azure BLAST is very similar to Cloud BLAST in computing style, but is directly supported by the Azure Cloud Platform [21] rather than a MapReduce framework. However, compared with Hadoop, the Azure platform still provides similar functionalities such as data splitting, finding workers, load balancing, and fault tolerance.

3) Twister BLAST Solution

Twister BLAST is a parallel BLAST application which is supported by Twister. Based on the analysis of the three existing parallel BLAST applications discussed above, Twister BLAST also uses the state-of-the-art binary invoking parallelism. As mentioned previously, this style brings scalability and simplicity to program and database maintenance. The flexibility of the Twister framework allows this program to run on a single machine, a cluster, or Amazon EC2 cloud platform.

Before Twister BLAST execution, query chunks are distributed to all compute nodes through Twister scripts because the gene query could be sufficiently large that it cannot be loaded into the client's memory all at once. The gene query is then sent through twister scripts and a partition file is created to record the location of these query chunks. The partition file will replace KeyValue pairs and be configured to Map tasks as input information.

The BLAST Database is also replicated to all of the compute nodes. Though copying the entire database through network may by very costly, it nevertheless makes it easy to manage database versions and brings efficiency for later BLAST execution. In order to replicate the database across the network quickly, compression techniques are used here. A BLAST Database, such as the 10 GB NR database, will be compressed into 3 GB and then be distributed. Once the database copies arrive at compute nodes, they are extracted in parallel through a set of Map tasks. This reduces the time needed for replication to one third of the original time.

Twister BLAST also uses Map tasks to parallelize BLAST jobs. The Twister BLAST client sends job property messages through a set of message brokers to drive Map tasks. Then Twister starts Map tasks according to the partition file. Each Map task invokes the BLAST program with the query file location and other input command variables defined by user. Once jobs are completed, Twister reports the status to the client program. Outputs can be collected to one node by Twister scripts (See Fig. 3).

In addition, another important fact observed by domain experts may give us a chance to extend the Twister BLAST solution. Gene queries generated by Bioinformatics researchers can easily contain duplicates. There are already several tools to remove the duplication [22-26]. However, there is no scalable solution to handle large inputs. Here, Twister can be used to solve this problem by using a WordCount like MapReduce job before executing the parallel BLAST job. Once the original query is partitioned and distributed to all nodes, Map tasks can remove local duplicates, and then send KeyValue pairs, each of which uses a gene sequence as Key and a tag as Value. After receiving these KeyValue pairs, Reduce tasks can generate non-duplicate gene sequences with a unique tag. Assuming that this result set can be much smaller than the original data set, we can use the Twister Combine method to collect these gene sequences back to the client and then re-assign them to KeyValue pairs and send them back to Map tasks to do parallel BLAST. Depending on the quality of the inputs, Twister BLAST can likely provide a significant improvement in overall BLAST throughput utilizing this method.

4) Performance Tests and Comparison with Hadoop BLAST

A set of performance tests has also been conducted for Twister BLAST on the Indiana University Polar Grid Cluster by using 32 nodes. Each node has two 4-core CPUs (Intel Xeon CPU E5410 2.33GHz) and 16 GB memory, along with a Gigabit Ethernet network interface. Here, Twister BLAST is compared with the Hadoop BLAST implementation.

Hadoop BLAST basically has the same style as the implementation mentioned in the Cloud BLAST paper. It uses HDFS to hold the compressed BLAST program and database, and then uses a distributed cache to allocate them to the local disk. Hadoop BLAST equally splits the query file into sequence chunks, and copies them to HDFS. Once the program and data are prepared, they are downloaded, extracted and taken as input by the assigned Map task.

Query sequences are selected from the data provided by Center for Genomics and Bioinformatics [27]. It consists of 115 million sequences and each of them has a uniform length of 37 DNA letters. For fairness, removing duplicates are not considered in this experiment. The BLAST job is parallelized by using 256 map tasks. By changing input size, the performance is examined as the growth of total execution time with the input size.

Together, replication of the NR database and query distribution took 1006 seconds for transferring 2.9 GB compressed data and extracting them using Twister BLAST, while Hadoop BLAST uses 693 seconds for the same operations. For the BLAST execution stage, the result, as shown in Fig. 4, shows that the execution time is proportional to the number of gene sequences. Compared with Hadoop BLAST, Twister BLAST has little overhead to computation and is also slightly faster than Hadoop BLAST due to Twister's lightweight [2].

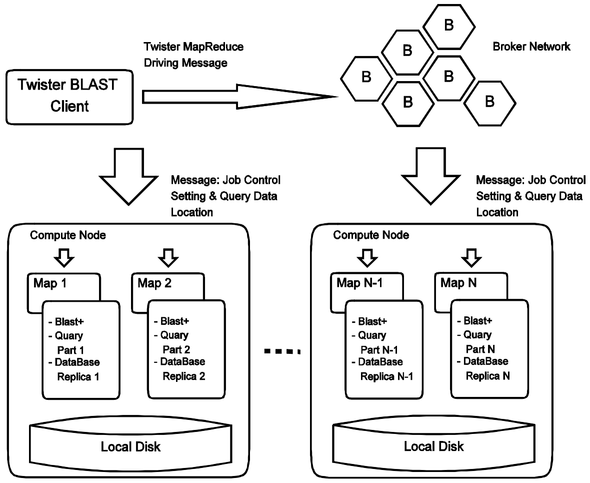

Figure 3. Twister-BLAST workflow and architecture

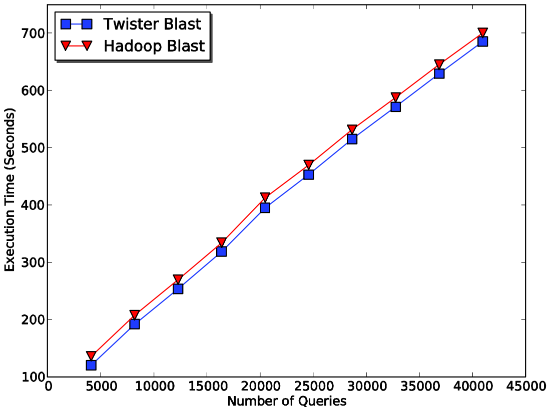

Figure 4. Performance comparison with Twister BLAST and Hadoop BLAST on 32 PG nodes

However, because of Twister's static scheduler, it cannot dynamically schedule queries to Map tasks. In the experiment, due to the characteristics of the queries, the result shows that Map tasks have different execution times and the final execution time is decided by the longest Map task execution time. By randomizing the input data, this issue can be improved but not solved.

B. *Twister MDS Interpolation*

Twister MDS Interpolation is a parallel method for MDS Interpolation using the Twister Framework. The implementation and performance testing of this program are discussed below.

1) *MDS Interpolation*

Multidimensional scaling (MDS) [28] is known as a dimension reduction algorithm which is widely used in statistics analysis. It is usually used to investigate large data points which may approach 100k in quantity. However, if this algorithm is computed directly, its time complexity is $O(N^2)$ level, where N is the total number of points. Because storing and calculating matrix requires large memory, this algorithm is also memory-bound. As a result, it is very difficult to run MDS on over 1 million data points. Now, with MDS interpolation, these problems can be overcome by processing the full dataset based on the result from a subset of it.

MDS interpolation is an out-of-sample problem [29]. The data subset which is produced from a full MDS calculation is the sample, and the rest of the dataset are the out-of-sample points. The time complexity of MDS interpolation is $O(KM)$, where K is the number of sample points and M is the number of out-of-sample points. This greatly reduces the time required to do dimension reduction of MDS and makes processing millions of points possible.

In order to find a new mapping position for an out-of-sample point, we first do normal MDS on the selected n points as sample points from the full dataset to reduce the dimension toL, and then select k nearest neighbors $p_1, \cdots, p_k$ from the sample points for an x from the out-of-sample points. By using this information, we can construct a STRESS function and minimize it. This method which is similar to an MDS algorithm is known as SMACOF [30]. Since only one point is movable among the sample points, we set the weight to 1 for simplification. The STRESS function is

$$\sigma(X) = \sum_{i<j\leq N}\left(d_{ij}(X) - \delta_{ij}\right)^2 = C + \sum_{i=1}^{k} d_{ix} - 2\sum_{i=1}^{k} \delta_{ix} d_{ix} \qquad (1)$$

Here δ_{ij} is the original dissimilarity value between p_i and x, d_{ix} is the Euclidean distance in L dimension between p_i and x, and C is a constant.

According to Seung-Hee Bae's method [31], we can minimize this STRESS function by the following equation.

$$x^{[t]} = \overline{p} + \frac{1}{k}\sum_{i=1}^{k} \frac{\delta_{ix}}{d_{iz}}\left(x^{[t-1]} - p_i\right) \qquad (2)$$

Here $d_{iz} = \left\|p_i - x^{[t-1]}\right\|$ and $\overline{p}$ is the average of k sample points' mapping results. The stopping criteria for this algorithm would be

$$\Delta\sigma\left(S^{[t]}\right) = \sigma\left(S^{[t-1]}\right) - \sigma\left(S^{[t]}\right) < \theta \qquad (3)$$

Here $S = P \cup \{x\}$ and θ is the given threshold value. We then take this $x^{[t]}$ as our result.

2) *Parallel MDS Interpolation Approach*

There are already some types of parallel MDS interpolation methods [32], such as the applications under MPI.net [33] and Dryad [34]. But this time we are going to show how to use Twister to do it. Even though MDS interpolation can dramatically reduce the time required to perform the dimension reduction computation, the memory issue cannot be solved by the algorithm itself because distance matrix file for 1 million data points could require up to 6 TB and it is very costly to move this distance file around the compute nodes. As a result, in Twister MDS interpolation, the algorithm implementation is vector-based, and the raw dataset is read instead of the Euclidean distance dataset. The raw-data file is split into equally sized files and distributed over compute nodes. Twister then uses the partition file to locate the raw data chunks. Then Twister MDS Interpolation creates map tasks on each node and uses "Map only" mode to start map tasks according to the data locations in the partition file. Data are processed by functions encapsulating the MDS interpolation algorithm in each map task and output can be collected by Twister Script.

3) *Performance Test*

Performance tests are done for Twister MDS Interpolation on the Indiana University PolarGrid machine which is mentioned in Section 3.1. The numbers of nodes used in the tests are 8 nodes, 16 nodes and 32 nodes. The respective tests utilize 64 cores, 128 cores and 256 cores. The input dataset is taken from PubChem [35], its original size is 18 million data points. In the experiment, we take 4 million and 8 million data points from this original set (See Fig. 5).

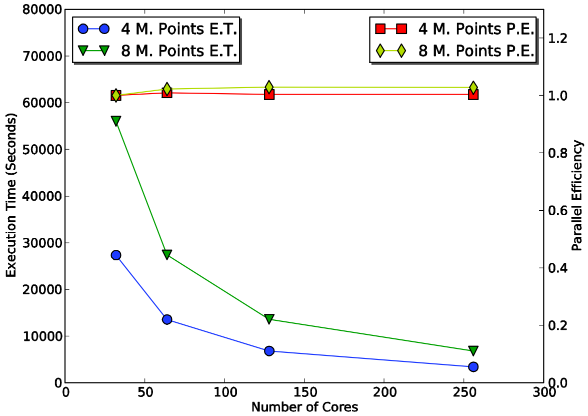

Figure 5. Twister MDS Interpolation execution time and parallel efficiency

In Fig. 5, parallel efficiency is shown on the right y-axis and computation time is shown on the left y-axis. The x-axis shows the core number. The efficiency of computing is calculated as following:

$$\mathrm{ParallelEfficiency}(\eta_i) = \frac{T(p_1)}{\alpha . T(p_i)} \quad (4)$$

Here $T(p_i)$ is the execution time on i nodes, p_1 is the smallest number of nodes running the program, and $\alpha = p_i/p_1$.

The parallel efficiency is around 1 even as the number of cores increases; this is because there is no communication between nodes when we run the MDS Interpolation in parallel. So, with increasing number of cores, Twister MDS interpolation performs better.

C. *Twister GTM Interpolation*

Twister GTM Interpolation is a new method of parallelizing GTM Interpolation. We use the binary GTM program and information from the results of running the full GTM algorithm to design the new program.

1) *GTM Interpolation*

The Generative Topographic Mapping (GTM) algorithm is used to find an optimal representation of data from high dimensional space to low dimensional space. It seeks a non-linear mapping of user-defined K points in the low dimensional space for N data points in a way that these K points can represent the N data points in the original high dimensional space [36]. The time complexity of this problem is $O(KN)$. Although this algorithm is faster than MDS, since $K < N$, it is still a challenge to compute large datasets, such as 8 million data points.

To solve this issue, GTM Interpolation was designed to first perform normal GTM on a subset of the full dataset, known as samples. The remaining out-of-sample data can be learned from previous samples. Since the out-of-sample data are not involved in the computing intensive learning process, GTM Interpolation can be very fast. However, for more complicated data, there are some complex ways to interpolate GTM [37-39].

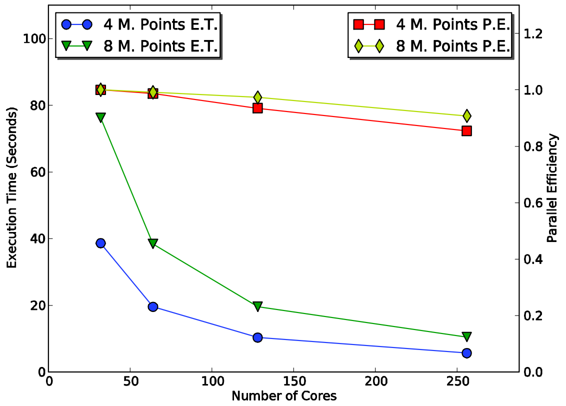

Figure 6. Twister GTM Interpolation execution time and parallel efficiency

According to Jong Choi's work [31], a simple interpolation can be accomplished as described here. For example, to process 26 million data points, 100k data are first sampled from the original dataset. Then GTM is performed on this sample data to find an optimal K cluster center and a coefficient β for this sample set. This information is stored in several files. After that, for the remaining out-of-sample data M, a $K \times M$ pairwise distance matrix D is computed with entries d_{ij} which is a Gaussian probability between the sample data and out-of-sample data. So the responsibility matrix R can be computed as

$$R = D\emptyset(ee^t D) \quad (5)$$

Here $e = (1, \cdots, 1)^t \epsilon R^k$ and $\emptyset$ represents element wise division.

Finally, with this information, we can construct a GTM map $\overline{Z} = R^t Z$ as where Z is the matrix representation of the sample points.

2) *Parallel GTM Interpolation Approach*

Previously, GTM Interpolation has been parallelized by using Dryad, Hadoop and Amazon EC2 [32]. Here, we present the Twister parallelization of this program. The Twister GTM Interpolation software can split the raw data file from the out-of-sample data file. Each partition will have a mapper created to process that chunk. Once this is done, Twister will invoke GTM Interpolation on each chunk to process the data. The mappers will process each block individually, and, the results can be collected by using Twister script commands.

3) *Performance Test*

The performance test is also done on Indiana University PolarGrid. 4 million and 8 million points from the PubChem data [35] are selected, and the sample data size is 100k.

As can be seen in Fig. 6, GTM-Interpolation runs very fast on PolarGrid, requiring 76 seconds to run on 4 nodes (32 cores), and Twister's parallel efficiency remains above 0.85 as the number of cores increases. This indicates favorable performance for a parallel program, and we anticipate that

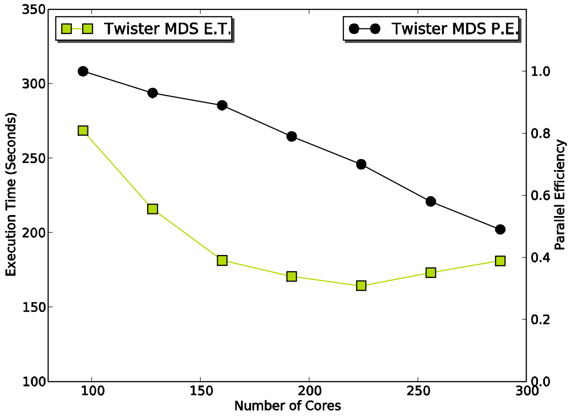

Figure 7. Twister MDS execution time and parallel efficiency

with increasing number of cores, even above 256 cores, the parallel efficiency will remain above 0.8 and become more stable.

IV. Twister Iterative Applications

The unique feature of Twister is to support the iterative MapReduce programming model, in which the client can drive Twister to finish a MapReduce job in iterations. The performance is optimized by caching static data to be used throughout the computation and by using a message infrastructure for communication. Faults are handled between iterations. Here, Twister MDS is introduced to illustrate how iterative MapReduce works in Twister.

A. Twister MDS

Multidimensional scaling (MDS) is a set of algorithms which can map high dimensional data to low dimensional data with respect to the pairwise proximity information. In this algorithm, the pairwise Euclidean distance within the target dimension of each pair is approximated to the corresponding original proximity value. This procedure is called STRESS [40], and a non-linear optimization algorithm to find the low-dimensional dataset which minimizes the objective function.

Because a large high dimension distance matrix is involved, MDS is a very data intensive computing. The iterative MapReduce programming model has been applied to this algorithm in an attempt to facilitate its execution on ever-larger datasets. Specifically, the Twister MDS application is implemented and its performance and scalability is evaluated.

To reduce the memory requirement on a single node, the original distance matrix is partitioned into chunks by rows. These chunks are distributed to all compute nodes, and the partition information is recorded in a partition file. These data chunks are assigned to Map tasks in a one to one mapping and then held in memory and used throughout the subsequent iterations.

Twister MDS shows how the concept of "configure once and run several times" works. After initialization, it configures three Twister jobs. Two of them are matrix-vector multiplications and the other is a STRESS value calculation. Once these jobs are configured, the client begins to do iterations. In each loop, the client will invoke these three jobs sequentially. The matrix result obtained from the previous job is collected by the client and used as KeyValue pairs input for the following job. Since the intermediate matrix result is required by all Map tasks of the next job according to the algorithm, they are sent through the runMapReduceBCast method which can broadcast the data value to all nodes with different keys. Once a loop is done, the mapping matrix result and STRESS values are used as input for next loop. The client can control the number of iterations, and once the maximum number of iterations have completed, the client stops computing.

To evaluate the performance of Twister MDS, a Twister environment with one ActiveMQ message broker was established, and Twister MDS was run with 100 iterations. A metagenomics dataset comprised of 30000 data points with nearly 1 billion pair-wise distances is tested here. Because this large dataset cannot be handled on a single machine, the method for calculating parallel efficiency used in the sections above is applied again. In other words, parallel efficiency is calculated with respect to the minimum number of CPU cores used in the experiment.

However, parallel efficiency drops greatly as the number of cores increases (See Fig. 7), and total execution time grows beyond a certain number of cores. The reason for this degraded performance is that the cost of data broadcasting increases as the number of cores grows. For example, in the case where 288 cores are used, more than half of the execution time is used in data transmission. Though the communication burden of broadcasting data is due to the nature of the algorithm itself and the problem can be eased by using more than one broker, this illustrates the limitation of one message broker and that broadcasting data through the broker should be implemented carefully in Twister iterative application design.

V. Conclusions and Future Work

In this paper, we present four parallel applications: Twister BLAST, Twister MDS Interpolation, Twister GTM Interpolation, and Twister MDS, along with details about their implementations and associated performance measurements. We show that Twister can be applied not only to applications with a non-iterative MapReduce programming model, but also to an iterative MapReduce programming model. New implementations extend the scope of existing applications using Twister. With iterative MapReduce functions, data partitioning, caching, and reusable configuration, Twister can solve problems in a flexible and efficient fashion.

As a runtime of iterative MapReduce, Twister aims to provide functionalities to accelerate the computation of iterative algorithms. However, it is limited by the availability of messaging middleware. Though having a flexible interface to allow the use of multiple messaging software packages is advantageous, Twister's performance largely depends on the performance of the specific messaging middleware adopted. For instance, the messaging performance of Twister implementation of the MDS iterative algorithm, are clearly influenced by the large volume of temporary results that must be broadcasted between iterations. This highlights an interesting research issue of balancing the requirement of employing an iterative algorithm with the capability of the messaging middleware. Twister scripts can simulate some functions of distributed file systems but needs further optimization. In future work, we will integrate Twister with a customized messaging middleware and a distributed file system.

ACKNOWLEDGMENT

Twister BLAST is supported by the National Institutes of Health under grant 5 RC2 HG005806-02.

REFERENCES

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

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

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

[4] Amazon, Amazon Web Services.http://aws.amazon.com/.

[5] J. B. MacQueen. Some Methods for classification and Analysis of Multivariate Observations. in 5-th Berkeley Symposium on Mathematical Statistics and Probability: University of California Press.

[6] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine; Available from: http://infolab.stanford.edu/~backrub/google.html.

[7] NaradaBrokering. Scalable Publish Subscribe System, 2010 [accessed 2010 May]; Available from: http://www.naradabrokering.org/.

[8] Apache, "ActiveMQ," http://activemq.apache.org/, 2009.

[9] NCBI. BLAST, 2010; Available from: http://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastNews#1.

[10] NCBI. BLAST Command Line Applications User Manual, 2010; Available from: http://www.ncbi.nlm.nih.gov/bookshelf/br.fcgi?book=helpblast&part=CmdLineAppsManual.

[11] C. Camacho, et al., BLAST+: architecture and applications. BMC Bioinformatics 2009, 10:421, 2009.

[12] PolarGrid. Cyberinfrastructure for Polar Expeditions, 2010 [accessed 2010 January]; Available from: http://www.polargrid.org/polargrid/index.php/Main_Page.

[13] NCBI. Databases available for BLAST search; Available from: http://www.ncbi.nlm.nih.gov/blast/blast_databases.shtml.

[14] A. Darling, L. Carey, and W. Feng, The Design, Implementation, and Evaluation of mpiBLAST. In: Proc ClusterWorld, 2003. 2003.

[15] A. Matsunaga, M. Tsugawa, and J. Fortes. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. in IEEE Fourth International Conference on eScience (eScience '08). 2008. Indianapolis, IN.

[16] W. Lu, J. Jackson, and R. Barga, AzureBlast: A Case Study of Developing Science Applications on the Cloud, in ScienceCloud: 1st Workshop on Scientific Cloud Computing co-located with HPDC 2010 (High Performance Distributed Computing). 2010, ACM: Chicago, IL.

[17] "MPI," Message Passing Interface, http://www-unix.mcs.anl.gov/mpi/, 2009.

[18] NCBI, NCBI Toolkit.http://www.ncbi.nlm.nih.gov/BLAST/developer.shtml

[19] H. Lin, et al., Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture, in SC2008. 2008.

[20] Hadoop Distributed File System HDFS, 2009 [accessed 2009 December]; Available from: http://hadoop.apache.org/hdfs/.

[21] Windows Azure Platform, Retrieved April 20, 2010, from Microsoft: http://www.microsoft.com/windowsazure/.http://www.microsoft.com/windowsazure/.

[22] ElimDupes; Available from: http://hcv.lanl.gov/content/sequence/ELIMDUPES/elimdupes.html.

[23] geneious; Available from: http://www.geneious.com/default,1266,new_features.sm.

[24] V. Seguritan and F. Rohwer, FastGroup: A program to dereplicate libraries of 16S rDNA sequences. BMC Bioinformatics, 2001. 2:9.http://www.ncbi.nlm.nih.gov/pmc/articles/PMC59723/

[25] D. Kerk, et al., The Complement of Protein Phosphatase Catalytic Subunits Encoded in the Genome of Arabidopsis. Plant Physiology, 2002. 129: p. 908–925.

[26] S. Drabenstot, et al., FELINES: a utility for extracting and examining EST-defined introns and exons. Nucleic Acids Research, 2003. 31.

[27] Center for Genomics and Bioinformatics; Available from: http://cgb.indiana.edu/.

[28] J. Kruskal and M. Wish, Multidimensional Scaling. 1978: Sage Publications Inc.

[29] M. Trosset and C. Priebe, The Out-of-Sample Problem for Classical Multidimensional Scaling. 2006, Bloomington, IN: Indiana University.

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

[31] J. Choi, S. Bae, J. Qiu, G. Fox, Dimension Reduction and Visualization of Large High-dimensional Data via Interpolation, in HPDC'10 2010: Chicago, Illinois USA.

[32] T. Gunarathne, T. Wu, J. Qiu, and G. Fox, Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications, in Proceedings of the Emerging Computational Methods for the Life Sciences Workshop of ACM HPDC 2010 conference. 2010: Chicago, Illinois.

[33] Indiana University Bloomington Open Sysem Lab. MPI.NET, 2008; Available from: http://osl.iu.edu/research/mpi.net/.

[34] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks, in ACM SIGOPS Operating Systems Review. 2007, ACM Press. p. 59-72.

[35] NCBI. PubChem; Available from: http://pubchem.ncbi.nlm.nih.gov/.

[36] C. Bishop, M. Svensén, and C. Williams, GTM: The generative topographic mapping. Neural computation, 1998. 10: p. 215--234.

[37] M. Carreira-Perpinan and Z. Lu. The Laplacian eigenmaps latent variable model. in 11th Int. Workshop on Artifical Intelligence and Statistics. 2007.

[38] A. Kaban. A scalable generative topographic mapping for sparse data sequences. in the International Conference on Information Technology: Coding and Computing. 2005.

[39] F. Nie S. Xiang, Y. Song, C. Zhang and C. Zhang, Embedding new data points for manifold learning via coordinate propagation. Knowledge and Information Systems, 2009. 19(2): p. 159-184.

[40] J. Kruskal, Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. Psychometrika, 1964. 29: p. 1-27.