MapReduce for Data Intensive Scientific Analyses

Jaliya Ekanayake and Shrideep Pallickara
Department of Computer Science
Indiana University, Bloomington, IN 47404, USA
{jekanaya, spallick}@indiana.edu

Abstract

Most scientific data analyses comprise analyzing voluminous data collected from various instruments. Efficient parallel/concurrent algorithms and frameworks are the key to meeting the scalability and performance requirements entailed in such scientific data analyses. The recently introduced MapReduce technique has gained a lot of attention from the scientific community for its applicability in large parallel data analyses. Although there are many evaluations of the MapReduce technique using large textual data collections, there have been only a few evaluations for scientific data analyses. The goals of this paper are twofold. First, we present our experience in applying the MapReduce technique for two scientific data analyses: (i) High Energy Physics data analyses; (ii) Kmeans clustering. Second, we present CGL-MapReduce, a stream based MapReduce implementation and compare its performance with Hadoop.

1. Introduction

Computation and data intensive scientific data analyses are increasingly prevalent. In the near future, it is expected that the data volumes processed by applications will cross the peta-scale threshold, which would in turn increase the computational requirements. Two exemplars in the data-intensive domains include High Energy Physics (HEP) and Astronomy. HEP experiments such as CMS and Atlas aim to process data produced by the Large Hadron Collider (LHC). The LHC is expected to produce tens of Petabytes of data annually even after trimming the datasets via multiple layers of filtrations. In astronomy, the Large Synoptic Survey Telescope produces data at a nightly rate of about 20 Terabytes.

Data volume is not the only source of compute intensive operations. Clustering algorithms used for DNA sequencing are especially compute intensive even though the datasets are comparably smaller than the physics and astronomy domains.

The use of parallelization techniques and algorithms is the key to achieve better scalability and performance for the aforementioned types of data analyses. Most of these analyses can be thought of as a Single Program Multiple Data (SPMD) [5] algorithm or a collection thereof. These SPMDs can be implemented using different techniques such as threads, message passing, and *MapReduce* (explained in section 2) [9].

There are several considerations in selecting an appropriate implementation strategy for a given data analysis. These include data volumes, computational requirements, algorithmic synchronization constraints, quality of services, easy of programming and the underlying hardware profile.

We are interested in the class of scientific applications where the processing exhibits the *composable* property. Here, the processing can be split into smaller computations, and the partial-results from these computations merged after some post-processing to constitute the final result. This is distinct from the tightly-coupled parallel applications where the synchronization constraints are typically in the order of microseconds instead of the 50-200 millisecond coupling constraints in composable systems. The level of coupling between the sub-computations is higher than those in the decoupled-model such as the multiple independent sub-tasks in job processing systems such as Nimrod [14].

The selection of the implementation technique may also depend on the quality of services provided by the technology itself. For instance, consider a SPMD application targeted for a compute cloud where hardware failures are common, the robustness provided by the MapReduce implementations such as Hadoop is an important feature in selecting the technology for this SPMD. On the other hand, the sheer performance of the message passing techniques is a desirable feature for several SPMD algorithms.

When the volume of the data is large, even tightly-coupled parallel applications can sustain less stringent synchronization constraints. This observation also favors the MapReduce technique since its relaxed synchronization constraints do not impose much of an overhead for large data analysis tasks. Furthermore, the simplicity and robustness of the programming model supersede the additional overheads.

To better understand these observations, we have selected two scientific data analysis tasks viz. HEP data analysis and Kmeans clustering [8]. We have implemented the tasks in the MapReduce programming model. Specifically, we implement these programs using Apache's MapReduce implementation – Hadoop [1], and also using CGL-MapReduce, a novel stream based MapReduce implementation developed by us. We compare the performance of these implementations in the context of these scientific applications and make recommendations regarding the usage of MapReduce techniques for scientific data analyses.

The rest of the paper is organized as follows. Section 2 gives an overview of the MapReduce technique and a brief introduction to Hadoop. CGL-MapReduce and its programming model are introduced in Section 3. In section 4, we present the scientific applications we used to evaluate the MapReduce technique while the section 5 presents the evaluations and a detailed discussion on the results that we obtained. The related work is presented in section 6 and in the final section, we will give our conclusions.

2. The MapReduce

In this section, we present a brief introduction of the MapReduce technique and an evaluation of existing implementations.

2.1. The MapReduce Model

MapReduce is a parallel programming technique derived from the functional programming concepts and is proposed by Google for large-scale data processing in a distributed computing environment. The authors [9] describe the MapReduce programming model as follows:

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation.

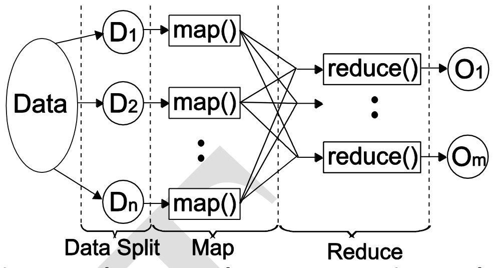

Figure 1. The MapReduce programming model

Counting word occurrences within a large document collection is a typical example used to illustrate the MapReduce technique. The data set is split into smaller segments and the map function is executed on each of these data segments. The map function produces a <key,value> pair for every word it encounters. Here, the “word” is the key and the value is 1. The framework groups all the pairs which has the same key (“word”) and invokes the reduce function passing the list of values for a given key. The reduce function adds up all the values and produces a count for a particular key, which in this case is the number of occurrences of a particular word in the document set. Figure 1 shows the data flow and different phases of the MapReduce framework.

2.1. Existing Implementations

Google's MapReduce implementation is coupled with a distributed file system named Google File System (GFS) [17]. According to J. Dean et al., in their MapReduce implementation, the intermediate <key, value> pairs are first written to the local files and then accessed by the reduce tasks. The same architecture is adopted by the Apache's MapReduce implementation - Hadoop. It uses a distributed file system called the Hadoop Distributed File System (HDFS) to store data as well as the intermediate results. HDFS maps all the local disks to a single file system hierarchy allowing the data to be dispersed at all the data/computing nodes. HDFS also replicates data on multiple nodes so that a failure of nodes containing a portion of the data will not affect computations, which use that data. Hadoop schedules the MapReduce computing tasks depending on the data locality and hence improving the overall I/O bandwidth. This setup is well suited for an environment where Hadoop is installed in a large cluster of commodity machines.

Hadoop stores the intermediate results of the computations in local disks, where the computation tasks are run, and then informs the appropriate workers to retrieve (pull) them for further processing. Although this strategy of writing intermediate result to the file system makes Hadoop a robust technology, it introduces an additional step and a considerable communication overhead, which could be a limiting factor for some MapReduce computations. Different strategies such as writing the data to files after a certain number of iterations or using redundant reduce tasks may eliminate this overhead and provide a better performance for the applications.

Apart from Hadoop, we found details of a few more MapReduce implementations targeting multi-core and shared memory architectures, which we will discuss in the related work section.

3. CGL-MapReduce

CGL-MapReduce is a novel MapReduce runtime that uses streaming for all the communications, which eliminates overheads associated with communicating via a file system. The use of streaming enables the CGL-MapReduce to send the intermediate results directly from its producers to its consumers. Currently, we have not integrated a distributed file system such as HDFS with CGL-MapReduce, and hence the data should be available in all computing nodes or in a typical distributed file system such as NFS. The fault tolerance support for the CGL-MapReduce will harness the reliable delivery mechanisms of the continent dissemination network that we use. Figure 2 shows the main components of the CGL-MapReduce.

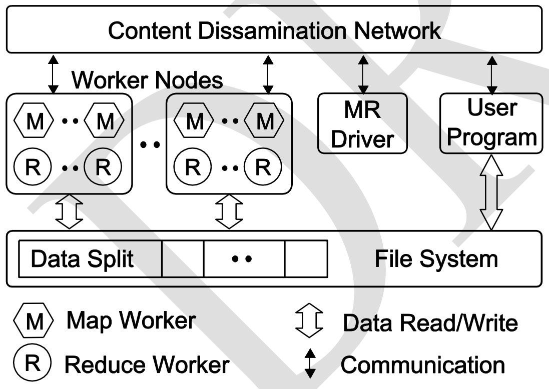

Figure 2. Components of the CGL-MapReduce

CGL MapReduce runtime comprises a set of workers, which perform map and reduce tasks and a content dissemination network that handles all the underlying communications. As in other MapReduce runtimes, a master worker (MRDriver) controls the other workers according to instructions given by the user program. However, unlike typical MapReduce runtimes, CGL-MapReduce supports both single-step and iterative MapReduce computations.

3.1. MapReduce computations with CGL-MapReduce

The different stages, which the CGL-MapReduce passes through during typical MapReduce computations, are shown in Figure 3 and the description of each stage follows.

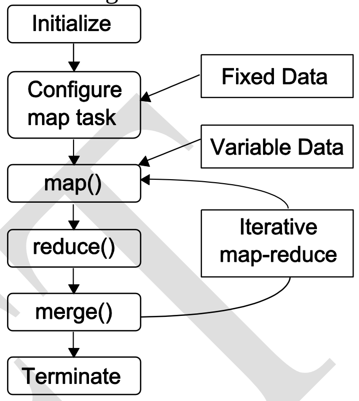

Figure 3. Various stages of CGL-MapReduce

Initialization Stage – The first step in using CGL-MapReduce is to start the MapReduce workers and configure both the map and reduce tasks. CGL-MapReduce supports configuring map/reduce tasks and reusing them multiple times with the aim of supporting iterative MapReduce computations efficiently. The workers store the configured map/reduce tasks and use them when a request is received from the user to execute the map task. This configuration step, which occurs only once, can be used to load any fixed data necessary for the map tasks. For typical single pass MapReduce computations, the user may not need to implement the configuration step.

Map Stage – After the workers are initialized, the User program instructs the MRDriver to start the map computations by passing the variable data (<key,value> pairs) to the map tasks. MRDriver relay this to the workers, which then invoke the configured map tasks. This approach allows the User program to pass the results from a previous iteration to the next iteration in the case of iterative MapReduce.

The outputs of the map tasks are transferred directly to the appropriate reduce workers using the content dissemination network.

Reduce Stage – Reduce workers are initialized in the same manner as the map workers. Once initialized, the reduce workers wait for the map outputs. MRDriver instructs the reduce workers to start executing the reduce tasks once all the map tasks are completed. Output of the reduce computations are also sent directly to the user program.

Combine Stage – Once the user program receives all the outputs of the reduce computations; it may perform a combine operation specified by the user. For a typical single-pass MapReduce computation this step can be used to combine the results of the reduce tasks to produce the final results. In the case of iterative MapReduce computations, this step can be used to compute the deciding value to continue the iterations.

Termination Stage – The user program informs the MRDriver its status of completing the MapReduce computation. The MRDriver also terminates the set of workers used for the MapReduce computation.

3.2. Implementation

CGL-MapReduce is implemented in Java and utilizes NaradaBrokering [18], a streaming-based content dissemination network developed by us. The CGL-MapReduce research prototype provides runtime capabilities of executing MapReduce computations written in the Java language. MapReduce tasks written in other programming languages require wrapper map and reduce tasks in order for them to be executed using CGL-MapReduce.

As mentioned in the introduction, the fault tolerance is an important aspect for MapReduce computations since the overall computation depends on the results produced by each execution of the map and reduce functions. In CGL-MapReduce, we have identified three crucial components, which need to support, fault tolerance. They are: (i) MRDriver, (ii) Map Worker, and (iii) Reduce Worker. MRDriver can be made fault tolerant by using redundancy and typical checkpointing strategies. Failure of a Map Worker can easily be corrected by adopting a policy of re-executing failed map tasks. Handling the failures of reduce workers is more crucial in our implementation since a given reduce worker may have the results of many map tasks which have already completed and these intermediate results are directly transferred to the Reduce Worker without writing them to the persistence storage. We are planning to use the reliable streaming feature of NaradaBrokering [19] to implement the fault tolerance in CGL-MapReduce. We will present details of our ongoing research in fault-tolerance in subsequent articles.

4. Scientific Applications

This section describes the scientific data analysis tasks, which we implemented using both Hadoop and CGL-MapReduce and the challenges we faced in implementing them.

4.1. HEP Data Analysis

As part of an effort funded by the DoE we are working with the High Energy Physics group at Caltech with their particle physics data analysis tools. The data analysis framework used by these tools is ROOT[16], and the analysis functions are written using an interpreted language of ROOT named CINT[4].

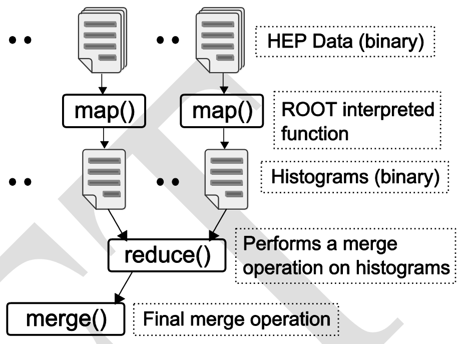

Figure 4. MapReduce for the HEP data analysis

The goal of the analysis is to execute a set of analysis functions on a collection of data files produced by high-energy physics experiments. After processing each data file, the analysis produces a histogram of identified features. These histograms are then combined to produce the final result of the overall analysis. This data analysis task is both data and compute intensive and fits very well in the class of composable applications. Figure 4 shows the program flow of this analysis once it is converted to a MapReduce implementation.

Although there are many examples for using MapReduce for textual data processing using Hadoop, we could not find any schemes for using MapReduce for these types of applications. Hadoop expects the data for the MapReduce tasks to be in its distributed file system but currently there is no support from accessing the data in HDFS using other languages such as C++. Hadoop supports map and reduce functions written in other languages via a special API called Hadoop streaming, which executes the functions as separate processes and then collects the output of the function from standard output and feeds it to the reduce tasks. However, we could not use the above approach since the output of the map task is also in the binary format (a histogram file) and the reduce function expects it as a data file. We could modify the analysis function used for the map task in such a way that it will output the histogram file name instead of the data and then let Hadoop transfer this file name to the appropriate reduce task. This approach does not work either since the outputs are created in the local file system and the reduce tasks cannot access them unless they are stored in the HDFS.

The solution we came up with this is to write wrapper functions for the map and reduce tasks in Java, and use these wrapper functions to execute the data analysis functions written in CINT. The data is placed on a high-speed/high-bandwidth network file system so that all the map tasks can access them without reading them via HDFS.

The input to the map function is the names of the data files. Each map task will process some of these files and produce a histogram file. The map wrapper function reads this histogram file and saves it using HDFS. The output of the map wrapper will be the location of this file in HDFS. Hadoop's runtime collects these locations and send them to the appropriate reduce tasks (reduce wrappers). The reduce wrapper reads these histogram files from the HDFS and copies them to the local disk, where it has been executing, and invokes the reduce task written in CINT to perform the merging of the histograms. The merged histogram is again stored in HDFS by the reduce wrapper and the location is passed to the user program, which then performs a similar operation to merge them all into a single histogram.

We adopted the same technique to implement the CGL-MapReduce version of the above data analysis. However, in CGL-MapReduce version, the output histograms are directly transferred to the appropriate reduce tasks via NaradaBrokering. The reduce wrapper save the data as local files and executes the reduce task written in CINT. The output of the reduce task is also read by the reduce wrapper and transferred directly to the user program where a similar computation to merge all the histograms to a single histogram is performed.

4.2. Kmeans Clustering

The HEP data analysis task discussed in the previous section represents a class of MapReduce computations where the entire computation is performed in a single pass of data through the map and the reduce functions. Kmeans clustering is within the class of applications where multiple iterations of MapReduce computations are necessary for the overall computation. For this purpose, we used the Kmeans clustering algorithm to cluster a collection of 2D data points.

In Kmeans clustering, the target is to cluster a set of data points to a predefined number of clusters. An iteration of the algorithm produces a set of cluster centers where it is compared with the set of cluster centers produced during the previous iteration. The total error is the difference between the cluster centers produced at n^{th} iteration and the cluster centers produced at $(n-1)^{th}$ iteration. The iterations continue until the error reduces to a predefined threshold value. Figure 5 shows the MapReduce version of the Kmeans algorithm that we developed.

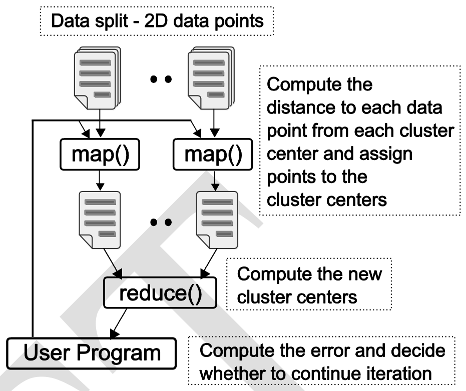
Data split - 2D data points
Compute the distance to each data point from each cluster center and assign points to the cluster centers
map()
map()
reduce()
Compute the new cluster centers
User Program
Compute the error and decide whether to continue iteration

Figure 5. MapReduce for Kmeans clustering

In Kmeans clustering, each map function gets a portion of the data, and it needs to access this data split in each iteration. These data items do not change over the iterations, and it is loaded once for the entire set of iterations. The variable data is the current cluster centers calculated during the previous iteration and hence used as the input value for the map function.

Hadoop's MapReduce API does not support configuring and using a map task over multiple iterations and hence the Hadoop version of the Kmeans algorithm, the map task loads the data at each iteration.

As mentioned in section 3, CGL-MapReduce allows the map task to be configured and used for multiple iterations. This gives a CGL-MapReduce a performance advantage over Hadoop in addition to the performance gain obtained by the use of streaming.

The output of the map task is a set of partial cluster centers. Hadoop handles the transfer of these partial centers to the reduce tasks via its distributed file system. In CGL-MapReduce these outputs are directly transferred to the reduce task by the runtime using the content dissemination network.

Once the reduce task receives all the partial cluster centers it computes new cluster centers. In the Hadoop version of this algorithm, the new cluster centers are written to HDFS and then read by the user program, which calculates the difference (error) between the new cluster centers and the previous cluster centers. If the difference is greater than a predefined threshold, the user program starts a new iteration of MapReduce using this new cluster centers as the input data. CGL-MapReduce version performs the same computations as the Hadoop version. However, the data transfer happens much faster because it uses streaming instead of a file system.

4.3. Experimental Setup

The amount of data we have for the HEP data analysis is about 1 Terabytes (TB). The data is stored in IU Data Capacitor: a high-speed and high-bandwidth storage system running the Lustre File System. For HEP data, we processed them using a cluster of 12 computing nodes. For Kmeans clustering, which uses a small data set of around 2GB, we used a cluster of 5 nodes.

All machines involved in the benchmarks had Dual Quad Core Intel Xeon processors and 8GB of memory and were running Red Hat Enterprise Linux operating system version 4. The JVM version 1.6.0_07 was used for the benchmarks and the gcc version 3.4.6 compiler was used for the C++ code. LAM MPI version 7.1.4 was used for the MPI implementations.

5. Evaluation

To evaluate the MapReduce technique for the HEP data analysis we first measured the total execution time it takes to process the data under different implementations by increasing the amount of data. As we increase the amount of data, we also increase the number of map tasks so that each map task processes almost the same amount of data in every run. Figure 6 depicts our results.

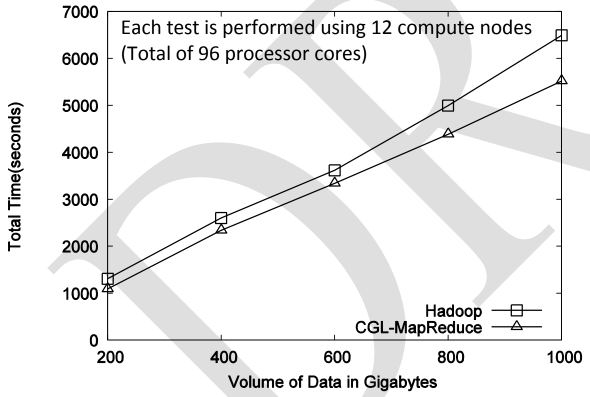

Figure 6. HEP data analysis, execution time vs. the volume of data (fixed compute resources)

Hadoop and CGL-MapReduce both show similar performance. The amount of data accessed in each analysis is extremely large and hence the performance is limited by the I/O bandwidth of a given node rather than the total processor cores. The overhead induced by the MapReduce implementations has negligible effect on the overall computation.

We performed another benchmark to see how the two MapReduce implementations scale as the number of processing units increases. We fixed the volume of the data at 100 GB, and measured the execution time by varying the number of nodes in the cluster. Since the overall performance is limited by the I/O bandwidth, we use only one processor core in each node for this evaluation. We also measured the time it takes to process the same 100GB of data using a sequential program and calculated the speedups achieved by Hadoop and CGL-MapReduce. The results shown in Figure 7 and 8 highlight the scalability of the MapReduce technique itself and the two implementations. The results also indicate how the speed gain diminish after a certain number of parallel processing units (after around 10 units) for the data set that we used. This is because after this threshold the overhead associated with the parallelization technique negates the effect of increased concurrency.

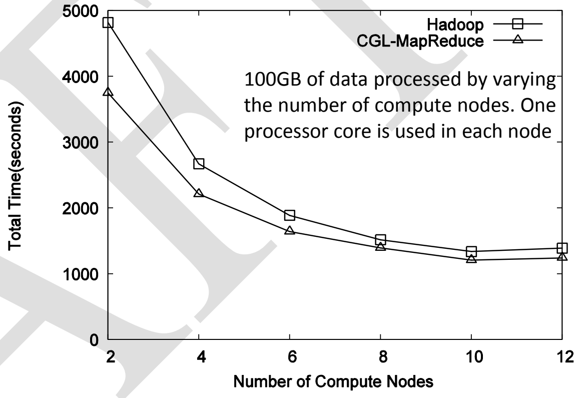

Figure 7. Total time vs. the number of compute nodes (fixed data)

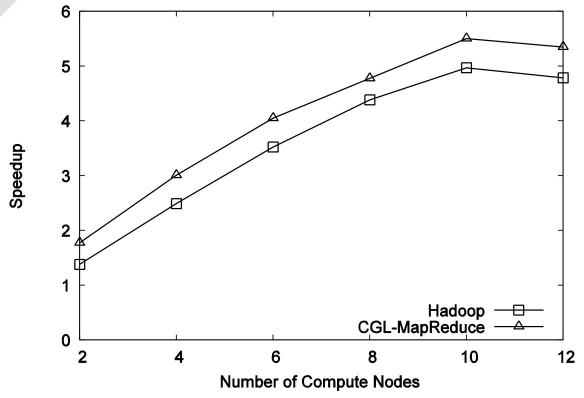

Figure 8. Speedup for 100GB of HEP data

For the Kmeans clustering, we first evaluate the overall performance of Hadoop and CGL-MapReduce by measuring the execution time as we increase the number of data points for clustering. We also evaluate the performance of an MPI [12] version of the same Kmeans clustering algorithm implemented in C++. The results are shown in Figure 9.

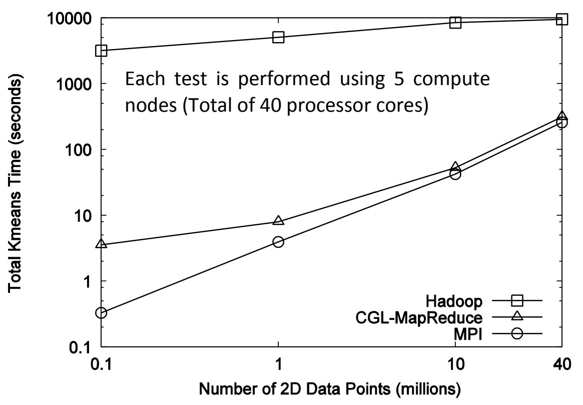

Figure 9. Total Kmeans time against the number of data points (Both axes are in log scale)

The lack of support for iterative MapReduce tasks and the large overhead caused by the file system based communication have largely affected the overall performance of Hadoop. CGL-MapReduce shows a performance close to the MPI implementation for higher number of data points.

To verify the above observation we calculated the overhead associated in each approach using the following formula:

$$\text{Overhead } f(P) = [P\ T_P - T_1]/T_1$$

Where *P* denotes the number of hardware processing units we used and the T_P is the total time as a function of the *P*. T_1 is the total time when a sequential program is used. T_1 is measured using sequential programs, implemented in Java for Hadoop and CGL-MapReduce and C++ for the MPI version, on a single node of the same cluster. The result of this analysis is shown in Figure 10.

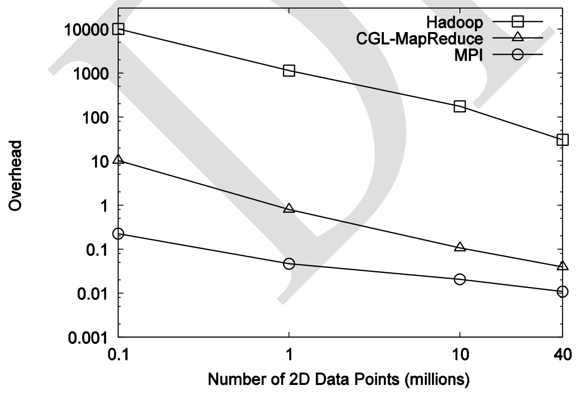

Figure 10. Overheads associated with Hadoop, CGL-MapReduce and MPI for iterative MapReduce

The results in Figures 9 and Figure 10 show how the approach of configuring once and re-using of map/reduce tasks for multiple iterations and the use of streaming have improved the performance of CGL-MapReduce for iterative MapReduce tasks making it almost comparable to the results of MPI for higher number of data points. The communication overhead and the loading of data multiple times have caused the Hadoop results to be almost hundred times more than that of CGL-MapReduce.

Cheng-Tao et al. [2] and Colby Ranger et al. [3] both used Kmeans clustering algorithm to evaluate their MapReduce implementations for multi-core and multi-processor systems. However, other clustering algorithms such as Deterministic Annealing [10], which we are investigating in our current research, will have much higher computation requirements and hence for such algorithms we expect that Hadoop's overhead will be smaller than the above.

6. Related Work

The SPMD programming style introduced by Frederica Darema has been a core technique in parallelizing applications since most applications can be considered as a collection of SPMD programs in different granularities. Parallel Virtual Machine [13] became the first standard of the SPMD programming and currently the MPI is the de-facto standard in developing SPMD programs.

MapReduce was first introduced in the Lisp programming language where the programmer is allowed to use a function to map a data set into another data set, and then use a function to reduce (combine) the results [6].

Swazall is an interpreted programming language for developing MapReduce programs based on Google's MapReduce implementation. R. Pike et al. present its semantics and its usability in their paper [15]. The language is geared towards processing large document collections, which are typical operations for Google. However, it is not clear how to use such a language for scientific data processing.

M. Isard et al. present Dryad - a distributed execution engine for coarse grain data parallel applications [11]. It combines the MapReduce programming style with dataflow graphs to solve the computation tasks. Dryad's computation task is a set of vertices connected by communication channels, and hence it processes the graph to solve the problem.

Hung-chin Yang et al. [7] adds another phase "merge" to MapReduce style programming, mainly to handle the various join operations in database queries. In CGL-MapReduce, we also support the merge operation so that the outputs of the all reduce tasks can be merged to produce a final result. This feature is especially useful for the iterative MapReduce where the user program needs to calculate some value depending on all the reduce outputs.

The paper presented by Cheng-Tao et al. discusses their experience in developing a MapReduce implementation for multi-core machines [2]. Phoenix, presented by Colby Ranger et al., is a MapReduce implementation for multi-core systems and multiprocessor systems [3]. 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. As we have shown under HEP data analysis, in data intensive applications, the overall performance depends greatly on the I/O bandwidth and hence a MapReduce implementation on multi-core system may not yield significant performance improvements. However, for compute intensive applications such as machine learning algorithms, the MapReduce implementation on multi-core would utilize the computing power available in processor cores better.

7. Conclusion

In this paper, we have presented our experience in applying the map-reduce technique for scientific applications. The HEP data analysis represents a large-scale data analysis task that can be implemented in MapReduce style to gain scalability. We have used our implementation to analyze up to 1 Terabytes of data. The Kmeans clustering represents an iterative map-reduce computation, and we have used it to cluster up to 40 million data points requiring around nearly 250 MapReduce iterations.

We performed the above two data analyses using Hadoop and CGL-MapReduce and compared the results. Our results confirm the observations: (i) Most scientific data analyses which has some form of SMPD algorithm can benefit from the MapReduce technique to achieve speedup and scalability; (ii) As the amount of data and the amount of computation increases, the overhead induced by a particular runtime diminishes; and (iii) Even tightly coupled applications can benefit from the MapReduce technique if the appropriate size of data and an efficient runtime is used.

In our future works, we are planning to improve the CGL-MapReduce in two key areas: (i) the fault tolerance support; (ii) integration of a distributed file system so that it can be used in a cluster of commodity computers where there is no shared file system. We are also studying the applicability of the MapReduce technique in cloud computing environments.

References

[1] Apache Hadoop, http://hadoop.apache.org/core/

[2] C. Chu, S. K. Kim, Y. Lin, Y.Y. Yu, G.R. Bradski, A. Y. Ng, K. Olukotun, “Map-Reduce for Machine Learning on Multicore,” NIPS 2006, pp. 281-288,

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. “Evaluating MapReduce for Multi-core and Multiprocessor Systems”, HPCA 2007, pp. 13-24.

[4] CINT - The CINT C/C++ Interpreter, http://root.cern.ch/twiki/bin/view/ROOT/CINT

[5] F. Darema, “SPMD model: past, present and future”, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, 2001.

[6] G. L. Steel, Jr. “Parallelism in Lisp”, SIGPLAN Lisp Pointers, VIII(2):1-14,1995.

[7] H. chih, A. Dasdan, R. L. Hsiao, and D. S. Parker, “Map-reducemerge: Simplified relational data processing on large clusters,” SIGMOD, 2007.

[8] J. B. MacQueen , “Some Methods for classification and Analysis of Multivariate Observations,” Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, University of California Press, vol. 1, pp. 281-297.

[9] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” OSDI’04: Sixth Symposium on Operating System Design and Implementation, Dec 2004.

[10] K. Rose, E. Gurewwitz, and G. Fox, “A deterministic annealing approach to clustering,” Pattern Recogn. Lett, vol. 11, no. 9, pp. 589-594,1995.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad:Distributed data-parallel programs from sequential building blocks,” European Conference on Computer Systems , March 2007.

[12] MPI (Message Passing Interface), http://www-unix.mcs.anl.gov/mpi/

[13] PVM (Parallel Virtual Machine), http://www.csm.ornl.gov/pvm/

[14] R. Buyya, D. Abramson, and J. Giddy, “Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid,” HPC Asia 2000, IEEE CS Press, USA, 2000.

[15] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, “Interpreting the data: Parallel analysis with sawzall,”Scientific Programming Journal Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure, vol. 13, no. 4, pp. 227–298, 2005.

[16] ROOT - An Object Oriented Data Analysis Framework, http://root.cern.ch/

[17] S. Ghemawat, H. Gobioff, and S. Leung, “The Google file system”, Symposium on Op-erating Systems Principles, pp 29–43, 2003.

[18] S. Pallickara and G. Fox, “NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids,” Middleware 2003, pp. 41-61.

[19] S. Pallickara, H. Bulut, and G. Fox, “Fault-Tolerant Reliable Delivery of Messages in Distributed Publish/Subscribe Systems,” 4th IEEE International Conference on Autonomic Computing, 2007.