DryadLINQ for Scientific Analyses

Jaliya Ekanayake[1,a], Atilla Soner Balkir[c], Thilina Gunarathne[a], Geoffrey Fox[a,b], Christophe Poulain[d], Nelson Araujo[d], Roger Barga[d]
[a]*School of Informatics and Computing, Indiana University Bloomington*
[b]*Pervasive Technology Institute, Indiana University Bloomington*
[c]*Department of Computer Science, University of Chicago*
[d]*Microsoft Research*
{jekanaya, tgunarat, gcf}@indiana.edu,
soner@uchicago.edu,{cpoulain,nelson,barga}@microsoft.com

Abstract

Applying high level parallel runtimes to data/compute intensive applications is becoming increasingly common. The simplicity of the MapReduce programming model and the availability of open source MapReduce runtimes such as Hadoop, attract more users around MapReduce programming model. Recently, Microsoft has released DryadLINQ for academic use, allowing users to experience a new programming model and a runtime that is capable of performing large scale data/compute intensive analyses. In this paper, we present our experience in applying DryadLINQ for a series of scientific data analysis applications, identify their mapping to the DryadLINQ programming model, and compare their performances with Hadoop implementations of the same applications.

1. Introduction

Among many applications benefit from cloud technologies such as DryadLINQ [1] and Hadoop [2], the data/compute intensive applications are the most important. The deluge of data and the highly compute intensive applications found in many domains such as particle physics, biology, chemistry, finance, and information retrieval, mandate the use of large computing infrastructures and parallel runtimes to achieve considerable performance gains. The support for handling large data sets, the concept of moving computation to data, and the better quality of services provided by Hadoop and DryadLINQ made them favorable choice of technologies to implement such problems.

Cloud technologies such as Hadoop and Hadoop Distributed File System (HDFS), Microsoft DryadLINQ, and CGL-MapReduce [3] adopt a more data-centered approach to parallel processing. In these frameworks, the data is staged in data/compute nodes of clusters or large-scale data centers and the computations are shipped to the data in order to perform data processing. HDFS allows Hadoop to access data via a customized distributed storage system built on top of heterogeneous compute nodes, while DryadLINQ and CGL-MapReduce access data from local disks and shared file systems. The simplicity of these programming models enables better support for quality of services such as fault tolerance and monitoring.

Although DryadLINQ comes with standard samples such as Terasort, word count, its applicability for large scale data/compute intensive scientific applications is not studied well. A comparison of these programming models and their performances would benefit many users who need to select the appropriate technology for the problem at hand.

We have developed a series of scientific applications using DryadLINQ, namely, CAP3 DNA sequence assembly program [4], High Energy Physics data analysis, CloudBurst [5] - a parallel seed-and-extend read-mapping application, and Kmeans Clustering [6]. Each of these applications has unique requirements for parallel runtimes. For example, the HEP data analysis application requires ROOT [7] data analysis framework to be available in all the compute nodes, and in CloudBurst the framework needs to process different workloads at *map/reduce/vertex* tasks. We have implemented all these applications using DryadLINQ and Hadoop, and used them to compare the performance of these two runtimes. CGL-MapReduce and MPI are used in applications where the contrast in performance needs to be highlighted.

In the sections that follow, we first present the DryadLINQ programming model and its architecture on HPC environment, and a brief introduction to

Hadoop. In section 3, we discuss the data analysis applications and the challenges we faced in implementing them along with a performance analysis of these applications. In section 4, we present the related work to this research, and in section 5 we present our conclusions and the future works.

2. DryadLINQ and Hadoop

A central goal of DryadLINQ is to provide a wide array of developers with an easy way to write applications that run on clusters of computers to process large amounts of data. The DryadLINQ environment shields the developer from many of the complexities associated with writing robust and efficient distributed applications by layering the set of technologies shown in Figure 1.

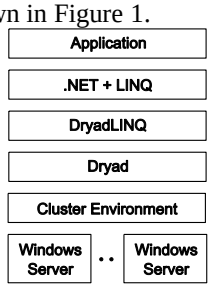

Figure 1. DryadLINQ software stack

Working at his workstation, the programmer writes code in one of the managed languages of the .NET Framework using Language Integrated Queries. The LINQ operators are mixed with imperative code to process data held in collection of strongly typed objects. A single collection can span multiple computers thereby allowing for scalable storage and efficient execution. The code produced by a DryadLINQ programmer looks like the code for a sequential LINQ application. Behind the scene, however, DryadLINQ translates LINQ queries into Dryad computations (Directed Acyclic Graph (DAG) based execution flows). While the Dryad engine executes the distributed computation, the DryadLINQ client application typically waits for the results to continue with further processing. The DryadLINQ system and its programming model are described in details in [1].

This paper describes results obtained with the so-called Academic Release of Dryad and DryadLINQ, which is publically available [8]. This newer version includes changes in the DryadLINQ API since the original paper. Most notably, all DryadLINQ collections are represented by the `PartitionedTable<T>` type. Hence, the example computation cited in Section 3.2 of [1] is now expressed as:

```
var input =
PartitionedTable.Get<LineRecord>("file://in.tbl");
var result = MainProgram(input, …);
var output =
result.ToPartitionedTable("file://out.tbl");
```

As noted in Figure 1, the Dryad execution engine operates on top of an environment which provides certain cluster services. In [1][9] Dryad is used in conjunction with the proprietary Cosmos environment. In the Academic Release, Dryad operates in the context of a cluster running Windows High-Performance Computing (HPC) Server 2008. While the core of Dryad and DryadLINQ does not change, the bindings to a specific execution environment are different and may lead to differences in performance.

Table 1. Comparison of features supported by Dryad and Hadoop

Feature	Hadoop	Dryad
Programming Model	MapReduce	DAG based execution flows
Data Handling	HDFS	Shared directories/ Local disks
Intermediate Data Communication	HDFS/ Point-to-point via HTTP	Files/TCP pipes/ Shared memory FIFO
Scheduling	Data locality/ Rack aware	Data locality/ Network topology based run time graph optimizations
Failure Handling	Persistence via HDFS Re-execution of map and reduce tasks	Re-execution of vertices
Monitoring	Monitoring support of HDFS, and MapReduce computations	Monitoring support for execution graphs
Language Support	Implemented using Java Other languages are supported via Hadoop Streaming	Programmable via C# DryadLINQ Provides LINQ programming API for Dryad

Apache Hadoop has a similar architecture to Google's MapReduce runtime, where it accesses data via HDFS, which maps all the local disks of the compute nodes to a single file system hierarchy allowing the data to be dispersed to all the data/computing nodes. Hadoop schedules the MapReduce computation tasks depending on the data locality to improve the overall I/O bandwidth. The outputs of the map tasks are first stored in local disks

until later, when the reduce tasks access them (pull) via HTTP connections. Although this approach simplifies the fault handling mechanism in Hadoop, it adds significant communication overhead to the intermediate data transfers, especially for applications that produce small intermediate results frequently. The current release of DryadLINQ also communicates using files, and hence we expect similar overheads in DryadLINQ as well. Table 1 presents a comparison of DryadLINQ and Hadoop on various features supported by these technologies.

3. Scientific Applications

In this section, we present the details of the DryadLINQ applications that we developed, the techniques we adopted in optimizing the applications, and their performance characteristics compared with Hadoop implementations. For all our benchmarks, we used two clusters with almost identical hardware configurations as shown in Table 2.

Table 2. Different computation clusters used for the analyses

Feature	Linux Cluster (Ref A)	Windows Cluster (Ref B)
CPU	Intel(R) Xeon(R) CPU L5420 2.50GHz	Intel(R) Xeon(R) CPU L5420 2.50GHz
# CPU # Cores	2 8	2 8
Memory	32GB	16 GB
# Disk	1	2
Network	Giga bit Ethernet	Giga bit Ethernet
Operating System	Red Hat Enterprise Linux Server -64 bit	Windows Server Enterprise - 64 bit
# Nodes	32	32

2.1. CAP3

CAP3 is a DNA sequence assembly program developed by X. Huang and A. Madan [4] that performs several major assembly steps such as computation of overlaps, construction of contigs, construction of multiple sequence alignments and generation of consensus sequences, to a given set of gene sequences. The program reads a collection of gene sequences from an input file (FASTA file format) and writes its output to several output files and to the standard output as shown below. The input data is contained in a collection of files, each of which needs to be processed by the CAP3 program separately.

Input.fsa -> Cap3.exe -> Stdout + Other output files

We developed a DryadLINQ application to perform the above data analysis in parallel. The DryadLINQ application executes the CAP3 executable as an external program, passing an input data file name, and the other necessary program parameters to it. Since DryadLINQ executes CAP3 as an external executable, it must only know the input file names and their locations. We achieve the above functionality as follows: (i) the input data files are partitioned among the nodes of the cluster so that each node of the cluster stores roughly the same number of input data files; (ii) a "data-partition" (A text file for this application) is created in each node containing the names of the original data files available in that node; (iii) Dryad "partitioned-file" (a meta-data file for DryadLINQ) is created to point to the individual data-partitions located in the nodes of the cluster.

Following the above steps, a DryadLINQ program can be developed to read the data file names from the provided partitioned-file, and execute the CAP3 program using the following two DryadLINQ queries.

```
IQueryable<Line Record> filenames =
PartitionedTable.Get<LineRecord>(uri);
IQueryable<int> exitCodes= filenames.Select(s
=> ExecuteCAP3(s.line));
```

Although we use this program specifically for the CAP3 application, the same pattern can be used to execute other programs, scripts, and analysis functions written using the frameworks such as R and Matlab, on a collection of data files. (Note: In this application, we assumed that DryadLINQ would process the input data files on the same nodes where they are located. Although this is not guaranteed by the DryadLINQ runtime, if the nodes containing the data are free during the execution of the program, it will schedule the parallel tasks to the appropriate nodes; otherwise, the data will be accessed via the shared directories.)

When we first deployed the application on the cluster, we noticed a sub-optimal CPU core utilization by the application, which is highly unlikely for a compute intensive program such as CAP3. A trace of job scheduling in the HPC cluster revealed that the scheduling of individual CAP3 executables in a given node is not optimal in terms of the utilization of CPU cores. In a node with 8 CPU cores, we would expect DryadLINQ to execute 8 `ExecuteCAP3()` functions simultaneously. However, we noticed that the number of parallel executions varies from 8 to 1 with the DryadLINQ's scheduling mechanism. For example, with 16 data file names in a data-partition; we noticed the following scheduling pattern (of concurrent `ExecuteCAP3()`s) from PLINQ, 8->4->4, which should ideally be 8 ->8.

When an application is scheduled, DryadLINQ uses the number of data partitions as a guideline to

determine the number of vertices to run. Then DryadLINQ schedules these vertices to the nodes rather than individual CPU cores assuming that the underlying PLINQ runtime would handle the further parallelism available at each vertex and utilize all the CPU cores. The PLINQ runtime, which is intended to optimize processing of fine grained tasks in multi-core nodes, performs optimizations by chunking the input data. Since our input for DraydLINQ is only the names of the original data files, it has no way to determine how much time the `ExecuteCAP3()` take to process a file, and hence the chunking of records at PLINQ results sub-optimal scheduling of tasks.

We found a workaround to this problem by changing the way we partition the data. Instead of partitioning input data to a single data-partition per node, we create data-partitions containing at most 8 (=number of CPU cores) line records (actual input file names). This way, we used DryadLINQ's scheduler to schedule series of vertices corresponding to different data-partitions in nodes while PLINQ always schedules 8 tasks at once, which gave us 100% CPU utilization. Figure 2 and 3 show comparisons of performance and the scalability of the DryadLINQ application, with the Hadoop and CGL-MapReduce versions of the CAP3 application.

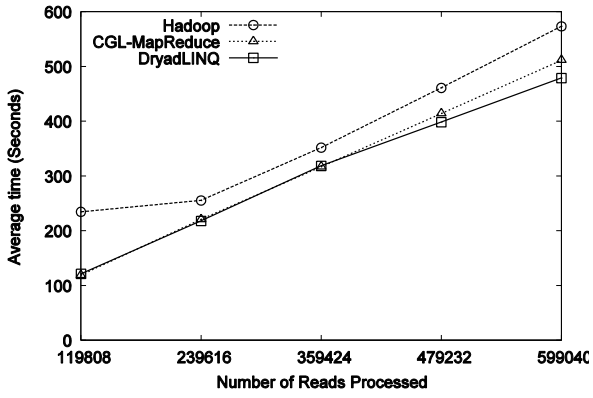

Figure 2. Performance of different implementations of CAP3 application

Although the above approach (partitioning data) works perfectly fine with this application, it does not solve the underlying problem completely. DryadLINQ does not schedule multiple concurrent vertices to a given node, but one vertex at a time. Therefore, a vertex which uses PLINQ to schedule some non-homogeneous parallel tasks would have a running time equal to the task which takes the longest time to complete. For example, in our application, if one of a input data file takes longer time to process than the others, which are assigned to a given vertex (assume a total of 8 files per vertex), the remaining 7 CPU cores will be idling while the task that takes longest completes. In contrast, in Hadoop, the user can set the maximum and minimum number of *map* and *reduce* tasks to execute concurrently on a given node so that it will utilize all the CPU cores.

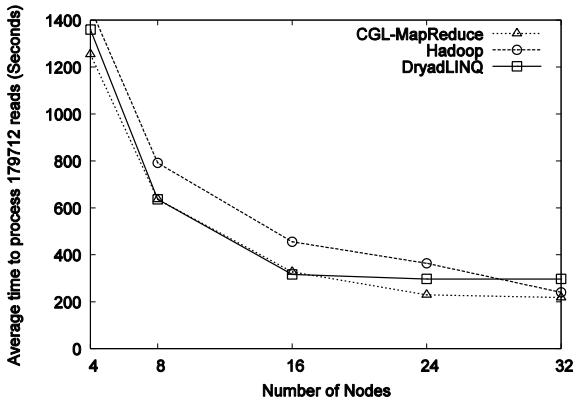

Figure 3. Scalability of different implementations of CAP3

The performance and the scalability graphs shows that all three runtimes work almost equally well for the CAP3 program, and we would expect them to behave in the same way for similar applications with simple parallel topologies.

2.2. High Energy Physics

Next, we developed a high energy physics (HEP) data analysis application and compared it with the previous implementations of Hadoop and CGL-MapReduce versions. As in CAP3, in this application the input is also available as a collection of large number of binary files, each with roughly 33MB of data, which will not be directly accessed by the DryadLINQ program. We manually partition the input data to the compute nodes of the cluster and generated data-partitions containing only the file names available in a given node. The first step of the analysis requires applying a function coded in ROOT script to all the input files. The analysis script we used can process multiple input files at once, therefore we used a `homomorphic Apply` (shown below) operation in Dryad to perform the first stage (corresponds to the *map()* stage in MapReduce) of the analysis.

```
[Homomorphic]
ApplyROOT(string fileName){..}

IQueryable<HistoFile> histograms =
dataFileNames.Apply(s => ApplyROOT (s));
```

Unlike the `Select` operation that processes records one by one, the `Apply` operation allows a function to be applied to an entire data set, and produce multiple output values. Therefore, in each vertex the program can access a data partition available in that node

(provided that the node is available for executing this application – please refer to the "Note" under CAP3 section). Inside the `ApplyROOT()` method, the program iterates over the data set and groups the input data files, and execute the ROOT script passing these files names along with other necessary parameters. The output of this operation is a binary file containing a histogram of identified features of the input data. The `ApplyROOT()` method saves the output histograms in a predefined shared directory and produces its location as the return value.

In the next step of the program, we perform a combining operation to these partial histograms. Again, we used a `homomorphic Apply` operation to combine partial histograms. Inside the function that is applied to the collection of histograms, we use another ROOT script to combine collections of histograms in a given data partition. (Before this step, the main program generates the data-partitions containing the histogram file names). The output partial histograms produced by the previous step will be combined by the main program to produce the final histogram of identified features.

We measure the performance of this application with different input sizes up to 1TB of data and compared the results with Hadoop and CGL-MapReduce implantations that we have developed previously. The results of this analysis are shown in Figure 4.

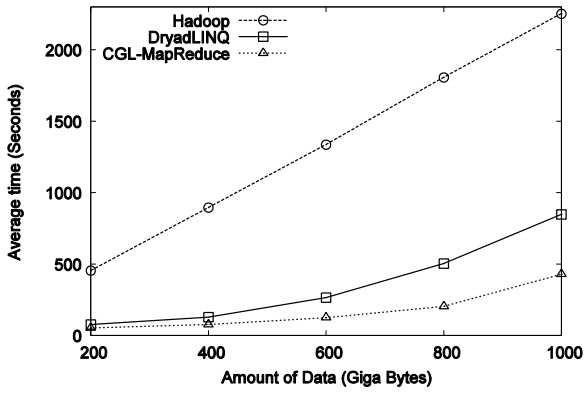

Figure 4. Performance of different implementations of HEP data analysis applications

The results in Figure 4 highlight that Hadoop implementation has a considerable overhead compared to DraydLINQ and CGL-MapReduce implementations. This is mainly due to differences in the storage mechanisms used in these frameworks. DryadLINQ and CGL-MapReduce access the input from local disks where the data is partitioned and distributed before the computation. Currently, HDFS can only be accessed using Java or C++ clients, and the ROOT – data analysis framework is not capable of accessing the input from HDFS. Therefore, we placed the input data in IU Data Capacitor – a high performance parallel file system based on Lustre file system, and allowed each map task in Hadoop to directly access the input from this file system. This dynamic data movement in the Hadoop implementation incurred considerable overhead to the computation. In contrast, the ability of reading input from the local disks gives significant performance improvements to both Dryad and CGL-MapReduce implementations.

As in CAP3 program, we noticed sub-optimal utilization of CPU cores by the HEP application as well. With heterogeneous processing times of different input files, we were able to correct this partially by carefully selecting the number of data partitions and the amount of records accessed at once by the `ApplyROOT()` function. Additionally, in the DryadLINQ implementation, we stored the intermediate partial histograms in a shared directory and combined them during the second phase as a separate analysis. In Hadoop and CGL-MapReduce implementations, the partial histograms are directly transferred to the *reducers* where they are saved in local file systems and combined. These differences can explain the performance difference between the CGL-MapReduce version and the DryadLINQ version of the program.

2.3. CloudBurst

CloudBurst is an open source Hadoop application that performs a parallel seed-and-extend read-mapping algorithm optimized for mapping next generation sequence data to the human genome and other reference genomes. It reports all alignments for each read with up to a user specified number of differences including mismatches and indels [5].

It parallelizes execution by seed, so that the reference and query sequences sharing the same seed are grouped together and sent to a reducer for further analysis. It is composed of a two stage MapReduce workflow: The first stage is to compute the alignments for each read with at most *k* differences where *k* is a user specified input. The second stage is optional, and it is used as a filter to report only the best unambiguous alignment for each read rather than the full catalog of all alignments. The execution time is typically dominated by the *reduce* phase.

An important characteristic of the application is that the variable amount of time it spent in the reduction phase. Seeds composed of a single DNA character occur a disproportionate number of times in the input data and therefore *reducers* assigned to these "low complexity" seeds spend considerably more time than the others. CloudBurst tries to minimize this effect by

emitting redundant copies of each "low complexity" seed in the reference and assigning them to multiple reducers to re-balance the workload. However, calculating the alignments for a "low complexity" seed in a reducer still takes more time compared to the others. This characteristic can be a limiting factor to scale, depending on the scheduling policies of the framework running the algorithm.

We developed a DryadLINQ application based on the available source code written for Hadoop. The Hadoop workflow can be expressed as:

```
Map -> Shuffle -> Reduce -> Identity Map -> Shuffle -> Reduce
```

The identity *map* at the second stage is used for grouping the alignments together and sending them to a *reducer*. In DryadLINQ, the same workflow is expressed as follows:

```
Map -> GroupBy -> Reduce -> GroupBy -> Reduce
```

Notice that we omit the identity map by doing an on-the-fly `GroupBy` right after the *reduce* step. Although these two workflows are identical in terms of functionality, DryadLINQ runs the whole computation as one large query rather than two separate MapReduce jobs followed by one another.

The *reduce* function takes a set of reference and query seeds sharing the same key as input, and produces one or more alignments as output. For each input record, query seeds are grouped in batches, and each batch is sent to an alignment function sequentially to reduce the memory limitations. We developed another DryadLINQ implementation that can process each batch in parallel assigning them as separate threads running at the same time using .NET Parallel Extensions.

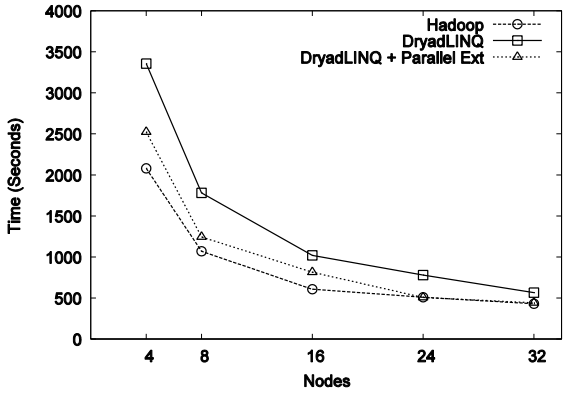

Figure 5. Figure5. Scalability of CloudBurst with different implementations.

We compared the scalability of these three implementations by mapping 7 million publicly available Illumina/Solexa sequencing reads [10] to the full human genome chromosome1.

The results in Figure 5 show that all three implementations follow a similar pattern although DryadLINQ is not fast enough especially with small number of nodes. As we mentioned in the previous section, DryadLINQ assigns vertices to nodes rather than cores and PLINQ handles the core level parallelism automatically by assigning records to separate threads running concurrently. However, we observed that the cores were not utilized completely, and the total CPU utilization per node varied continuously between 12.5% and 90% during the *reduce* step due to the inefficiencies with the PLINQ scheduler. Conversely, in Hadoop, we started 8 *reduce* tasks per node and each task ran independently by doing a fairly equal amount of work.

Another difference between DryadLINQ and Hadoop implementations is the number of partitions created before the *reduce* step. For example, with 32 nodes (with 8 cores each), Hadoop creates 256 partitions, since there are 256 *reduce* tasks. DryadLINQ, on the other hand, creates 32 vertices and assigns each vertex to a node (if we start the computation with 32 partitions initially). Notice that the partitions are created using a hash function, and they are not necessarily the same size. Creating more partitions results in having smaller groups and thus decreases the overall variance in the group size. The total time spent in the reduce function is equal to the maximum time spent in the longest reduce task. Since Hadoop creates more partitions, it balances the workload among reducers more equally. If the PLINQ scheduler worked as expected, this would not be a problem as it would keep the cores busy and thus yield a similar load balance to Hadoop.

As a work around to this problem, we tried starting the computation with more partitions. In the 32 node example, we created 256 partitions aiming to schedule 8 vertices per node. However, DryadLINQ runs the tasks in order, so it is impossible to start 8 vertices at once in one node with the current academic release. Instead, DryadLINQ waits for one vertex to finish before scheduling the second vertex, but the first vertex may be busy with only one record, and thus holding the rest of the cores idle. We observed that scheduling too many vertices (of the same type) to a node is not efficient for this application due to its heterogeneous record structure.

Our main motivation behind using the .NET parallel extensions was to reduce this gap by fully utilizing the idle cores, although it is not identical to Hadoop's level of parallelism.

Figure 6 shows the performance comparison of DryadLINQ and Hadoop with increasing data size. Both implementations scale linearly, and the time gap is mainly related to the current limitations with PLINQ

and DryadLINQ's job scheduling policies explained above. Hadoop shows a non linear behavior with the last data set and we will do further investigations with larger data sets to better understand the difference in the shapes.

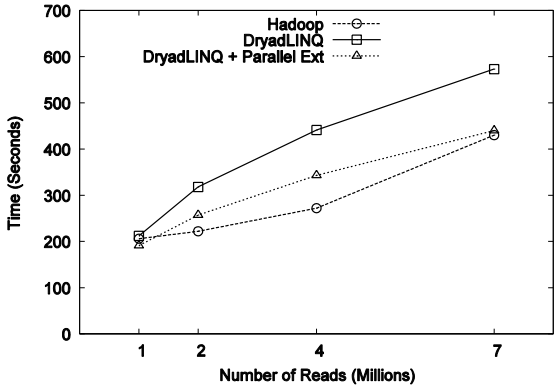

Figure 6. Performance comparison of DryadLINQ and Hadoop for CloudBurst.

2.4. Kmeans Clustering

We implemented a Kmeans Clustering application using DryadLINQ to evaluate its performance under iterative computations. We used Kmeans clustering to cluster a collection of 2D data points (vectors) to a given number of cluster centers. The MapReduce algorithm we used is shown below. (Assume that the input is already partitioned and available in the compute nodes). In this algorithm, V_i refers to the i^{th} vector, $C_{n,j}$ refers to the j^{th} cluster center in n^{th} iteration, D_{ij} refers to the Euclidian distance between i^{th} vector and j^{th} cluster center, and K is the number of cluster centers.

K-means Clustering Algorithm for MapReduce

do
  Broadcast C_n
  [Perform in parallel] –the map() operation
  for each V_i
    for each $C_{n,j}$
      D_{ij} <= *Euclidian* (V_i,$C_{n,j}$)
  Assign point V_i to $C_{n,j}$ with minimum D_{ij}
  for each $C_{n,j}$
    $C_{n,j}$ <=$C_{n,j}/K$
  [Perform Sequentially] –the reduce() operation
  Collect all C_n
  Calculate new cluster centers C_{n+1}
  Diff<= *Euclidian* (C_n, C_{n+1})
while (*Diff* <*THRESHOLD*)

The DryadLINQ implementation uses an `Apply` operation which will be performed in parallel in terms of the data vectors, to calculate the partial cluster centers. Another `Apply` operation, which works sequentially, calculates the new cluster centers for the n^{th} iteration. Finally, we calculate the distance between the previous cluster centers and the new cluster centers using a `Join` operation to compute the Euclidian distance between the corresponding cluster centers. DryadLINQ support "unrolling loops", using which multiple iterations of the computation can be performed as a single DryadLINQ query. In DryadLINQ programming model, the queries are not evaluated until a program accesses their values. Therefore, in the Kmeans program, we accumulate the computations performed in several iterations (we used 4 as our unrolling factor) as queries and "materialize" the value of the new cluster centers only at each 4th iteration. In Hadoop's MapReduce model, each iteration is represented as a separate MapReduce computation. Notice that without the loop unrolling feature in DryadLINQ, each iteration would be represented by a separate execution graph as well. Figure 7 shows a comparison of performances of different implementations of Kmeans clustering.

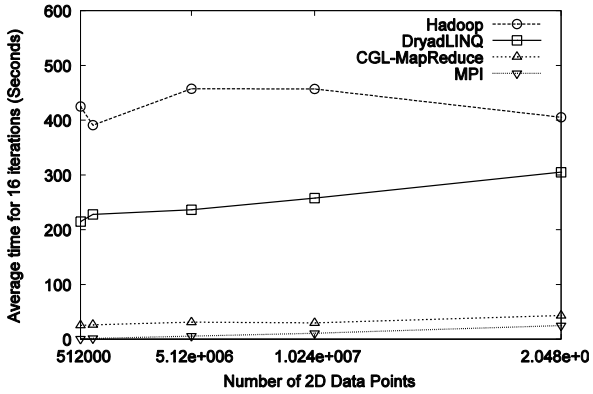

Figure 7. Performance of different implementations of Kmeans clustering algorithm

Although we used a fixed number of iterations, we changed the number of data points from 500k to 20 millions. Increase in the number of data points triggers the amount of computation. However, it was not sufficient to load the 32 node cluster we used. As a result, the graph in Figure 7 mainly shows the overhead of different runtimes. The use of file system based communication mechanisms and the loading of static input data at each iteration (in Hadoop) and in each unrolled loop (in Dryad) has resulted in higher overheads compared to CGL-MapReduce and MPI. Iterative applications which perform more computations or access large volumes of data may produce better results for Hadoop and Dryad, and ameliorate the higher overhead induced by these runtimes.

4. Related Work

Various scientific applications have been previously adapted to the MapReduce model for the past few years and Hadoop gained significant attention from the scientific research community. Kang et al. studied [11] efficient map reduce algorithms for finding the diameter of very large graphs and applied their algorithm to real web graphs. Dyer et al. described [12] map reduce implementations of parameter estimation algorithms to use in word alignment models and a phrase based translation model. Michael Schatz introduced CloudBurst [5] for mapping short reads from sequences to a reference genome. In our previous works [3][13], we have discussed the usability of MapReduce programming model for data/compute intensive scientific applications and the possible improvements to the programming model and the architectures of the runtimes. Our experience suggests that most composable applications can be implemented using MapReduce programming model either by directly exploiting their data/task parallelism or by adopting different algorithms compared to the algorithms used in traditional parallel implementations.

5. Conclusions and Future Works

We have applied DryadLINQ to a series of data/compute intensive applications with unique requirements. The applications range from simple map-only operations such as CAP3 to multiple stages of MapReduce jobs in CloudBurst and iterative MapReduce in Kmeans clustering. We showed that all these applications can be implemented using the DAG based programming model of DryadLINQ, and their performances are comparable to the MapReduce implementations of the same applications developed using Hadoop.

We also observed that cloud technologies such as DryadLINQ and Hadoop work well for many applications with simple communication topologies. The rich set of programming constructs available in DryadLINQ allows the users to develop such applications with minimum programming effort. However, we noticed that higher level of abstractions in DryadLINQ model sometimes makes fine-tuning the applications more challenging.

The current scheduling behavior of PLINQ hinders the performance of DryadLINQ applications. We would expect a fix to this problem would simply increase performance of DryadLINQ applications.

The features such as loop unrolling let DryadLINQ perform iterative applications faster, but still the amount of overheads in DryadLINQ and Hadoop is extremely large for this type of applications compared to other runtimes such as MPI and CGL-MapReduce.

As our future work, we plan to investigate the use of DryadLINQ and Hadoop on commercial cloud infrastructures.

6. Acknowledgements

We would like to thank the Dryad team at Microsoft Research, including Michael Isard, Mihai Budiu, and Derek Murray for their support in DryadLINQ applications, and Joe Rinkovsky from IU UITS for his dedicated support in setting up the compute clusters.

References

[1] Y.Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language," Symposium on Operating System Design and Implementation (OSDI), CA, December 8-10, 2008.

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

[3] J. Ekanayake and S. Pallickara, "MapReduce for Data Intensive Scientific Analysis," Fourth IEEE International Conference on eScience, 2008, pp.277-284.

[4] X. Huang and A. Madan, "CAP3: A DNA Sequence Assembly Program," Genome Research, vol. 9, no. 9, pp. 868-877, 1999.

[5] M. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce", Bioinformatics. 2009 June 1; 25(11): 1363–1369.

[6] J. Hartigan. Clustering Algorithms. Wiley, 1975.

[7] ROOT Data Analysis Framework, http://root.cern.ch/drupal/

[8] http://research.microsoft.com/en-us/downloads/03960cab-bb92-4c5c-be23-ce51aee0792c/

[9] Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1265-1276.

[10] The 1000 Genomes Project, "A Deep Catalog of Human Genetic Variation", January 2008, _http://www.1000genomes.org/page.php

[11] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, J. Leskovec, "HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop", CMU ML Tech Report CMU-ML-08-117, 2008.

[12] C. Dyer, A. Cordova, A. Mont, J. Lin, "Fast, Easy, and Cheap: Construction of Statistical Machine Translation Models with MapReduce", Proceedings of the Third Workshop on Statistical Machine Translation at ACL 2008, Columbus, Ohio.

[13] G. Fox, S. Bae, J. Ekanayake, X. Qiu, and H. Yuan, "Parallel Data Mining from Multicore to Cloudy Grids," High Performance Computing and Grids workshop, 2008.