DryadLINQ for Scientific Analyses

Jaliya Ekanayake[a], Thilina Gunarathne[a], Geoffrey Fox[a,b]
[a]School of Informatics and Computing, [b]Pervasive Technology Institute
Indiana University Bloomington
{jekanaya, tgunarat, gcf}@indiana.edu

Atilla Soner Balkir
Department of Computer Science, University of Chicago
soner@uchicago.edu

Christophe Poulain, Nelson Araujo, Roger Barga
Microsoft Research
{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, are attracting more users to the 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.**

Keywords-Cloud, MapReduce, DryadLINQ, Hadoop

I. INTRODUCTION

Among the benefits offered by cloud technologies, the ability to tackle data/compute intensive problems with frameworks such as DryadLINQ [1] and Hadoop [2] is one of 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 make them favorable choice of technologies to solve 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 the academic version of 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 sort of large data sets, word count, etc., its applicability for large scale data/compute intensive scientific applications is not well studied. 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 K-means 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 must handle worker tasks with very heterogeneous workloads at identical stages of the computation. 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 Windows HPC Server 2008 platform, and a brief introduction to Hadoop. In section 3, we discuss the data analysis applications, our experience implementing them, and a performance analysis of these applications. In section 4, we present work related to this research, and in section 5 we present our conclusions and future work.

II. 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 Fig. 1.

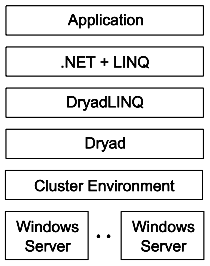

Figure 1. DryadLINQ software stack

Working at his workstation, the programmer writes code in one of the managed languages of the .NET Framework using Language Integrated Query [8]. 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 [9]. 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].

Our paper describes results obtained with the so-called Academic Release of Dryad and DryadLINQ, which is publically available [10]. 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 Fig. 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. In addition, not all features available in internal versions of Dryad are exposed in the external release. Please note that in the remainder of this paper, we use the terms Dryad or DryadLINQ to refer to the academic release of these systems.

Apache Hadoop has a similar architecture to Google's MapReduce [11] 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

TABLE I. COMPARISON OF FEATURES SUPPORTED BY DRYAD AND HADOOP

Feature	Hadoop	Dryad/DryadLINQ
Programming Model & Language Support	MapReduce Implemented using Java Other languages are supported via Hadoop Streaming	DAG based execution flows. DryadLINQ provides LINQ programming API for Dryad using a managed language (e.g. C#)
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

dispersed to all the data/compute 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.

III. 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 II. 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 release 5.3 -64 bit	Windows Server 2008 Enterprise (Service Pack 1) - 64 bit
# Nodes	32	32

A. CAP3

CAP3 is a DNA sequence assembly program, developed by Huang and Madan [4], which 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. During an actual analysis, the CAP3 program is invoked repeatedly to process a large collection of input FASTA file.

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

We developed a DryadLINQ application to perform the above data analysis in parallel. This application takes as input a *PartitionedTable* defining the complete list of FASTA files to process. For each file, the CAP3 executable is invoked by starting a process. The input collection of file locations is built as follows: (i) the input data files are distributed 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 file paths of the original data files available in that node; (iii) a DryadLINQ "partitioned file" (a meta-data file understood by 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 paths from the provided partitioned-file, and execute the CAP3 program using the following two lines of code.

```
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 rely on DryadLINQ to process the input data files on the same compute nodes where they are located. If the nodes containing the data are free during the execution of the program, the DryadLINQ runtime will schedule the parallel tasks to the appropriate nodes to ensure co-location of process and data; 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 utilization, which seemed 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 was not always utilizing all CPU cores. We traced this behavior to the use of an early version of the PLINQ [12] library (June 2008 Community Technology Preview), which DryadLINQ uses to achieve core level parallelism on a single machine.

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 partitions as vertices to the nodes (rather than individual CPU cores) and, uses the PLINQ runtime to achieve further parallelism within each vertex by taking full advantage of all the cores. The academic release of DryadLINQ uses the June 2008 preview version of PLINQ and this version of PLINQ does not always handle the scheduling of coarse grained parallel tasks well. We verified that this issue has been fixed in the current version of PLINQ and future releases of DryadLINQ will benefit from these improvements.

While using the preview version of PLINQ (which is publically available), we were able to reach full CPU utilization using the Academic release of DryadLINQ by changing the way we partition the data. Instead of partitioning input data to a single data-partition per node, we created 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. For the DryadLINQ application, note that the partitioning workaround will not be necessary to achieve these results once a version of DryadLINQ taking advantage of the improved version of the PLINQ library becomes publically available.

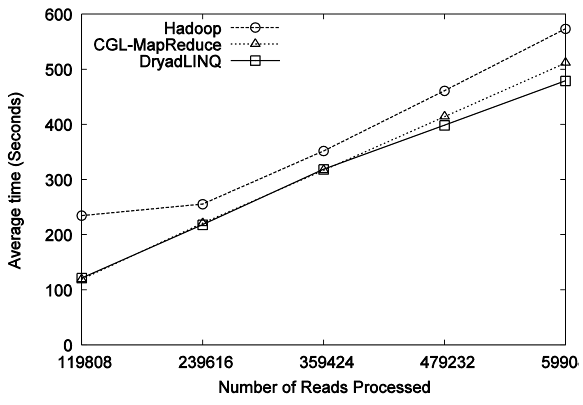

Figure 2. Performance of different implementations of CAP3 application.

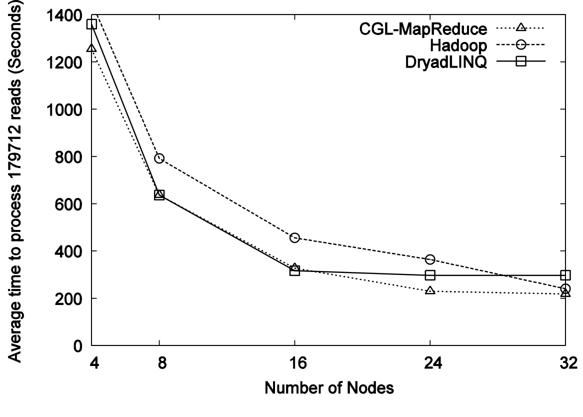

Figure 3. Scalability of different implementations of CAP3.

Fig. 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.

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.

B. 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 partitioned 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 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 DryadLINQ to perform the first stage (corresponding 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.

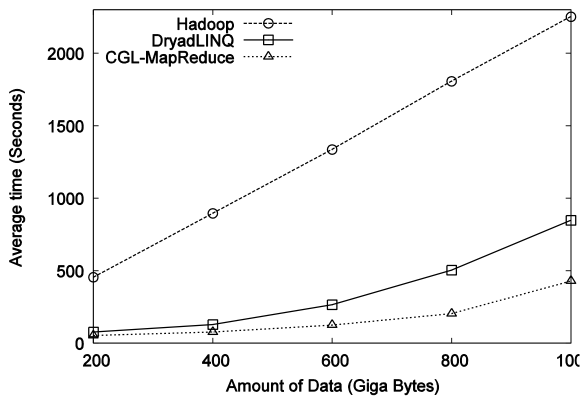

Figure 4. Performance of different implementations of HEP data analysis applications.

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 of these partial histograms. Again, we use 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 compare the results with Hadoop and CGL-MapReduce implementations that we have developed previously. The results of this analysis are shown in Fig. 4.

The results in Fig. 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 due to the above mention problem in the early version of PLINQ (June 2008 CTP). 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. We are planning to develop a better version of this application for DryadLINQ in the future.

C. *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 time spent by each worker process in the reduce phase varies considerably. 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.

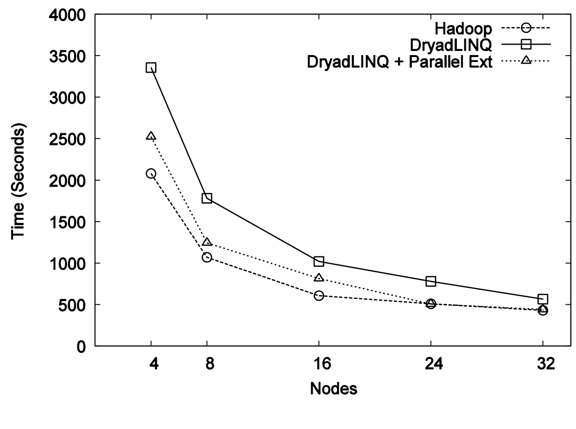

Figure 5. Scalability of CloudBurst with different implementations.

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.

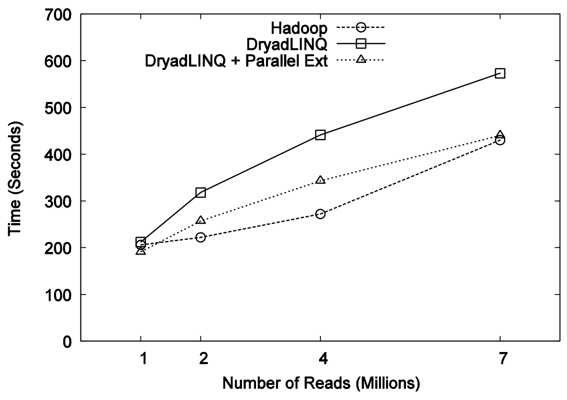

Figure 6. Performance comparison of DryadLINQ and Hadoop for CloudBurst.

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.

We compared the scalability of these three implementations by mapping 7 million publicly available Illumina/Solexa sequencing reads [13] to the full human genome chromosome1.

The results in Fig. 5 show that all three implementations follow a similar pattern although the DryadLINQ implementation is not as fast 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. Conversely, in Hadoop, we start multiple *reduce* tasks per node and each task runs is a separate process.

In order to better understand the performance difference, we isolated the reduce function as a standalone program and ran it on complex records with two different implementations. In the first implementation, we processed the records launching multiple reduce processes running independently. In the second one, we launched a single process and created multiple concurrent reduce tasks inside, each working on a separate record. Although both implementations were identical in functionality, we observed that the second implementation was slower. Since DryadLINQ creates multiple tasks using PLINQ in each node, this likely explains the performance reduction in the DryadLINQ implementation of CloudBurst. The root of the problem is still under inspection; it may be originating from several reasons such as excessive memory allocation in the code, garbage collection issues and complications with thread scheduling.

Another difference between DryadLINQ and Hadoop implementations is the number of partitions created before the *reduce* step. DryadLINQ creates vertices based on the initial number of partitions given as input. If we start the computation with 32 partitions, DryadLINQ creates 32 groups using a hash function and assigns each group to a vertex for the reduce operation. In Hadoop, the number of partitions is equal to the number of reduce tasks, which is specified as an input parameter. For example, with 32 nodes (8 cores each), Hadoop creates 256 partitions when we set the number of reduce tasks to 256. Having more partitions results in smaller groups and thus decreases the overall variance in the group size. Since Hadoop creates more partitions, it balances the workload among reducers more equally.

In the case of the DryadLINQ implementation, we can also start the computation with more partitions. However, DryadLINQ waits for one vertex to finish before scheduling the second vertex on the same node, 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.

Fig. 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 differences in job scheduling policies explained above. However, 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.

D. K-means Clustering

We implemented a K-means Clustering application using DryadLINQ to evaluate its performance under iterative computations. We used K-means 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.

The DryadLINQ implementation uses an `Apply` operation, which executes in parallel in terms of the data vectors, to calculate the partial cluster centers. Another `Apply` operation, which runs 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 "loop unrolling", using which multiple iterations of the computation can be performed as a single DryadLINQ query. Deferred query evaluation is a feature of LINQ, whereby a query is not evaluated until the program accesses the query results.. Thus, in the K-means program, we accumulate the computations performed in several iterations (we used 4 as our unrolling factor) into one query and only "materialize" the value of the new cluster centers every 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. Fig. 7 shows a comparison of performances of different implementations of K-means clustering.

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 ($Diff < THRESHOLD$)
```

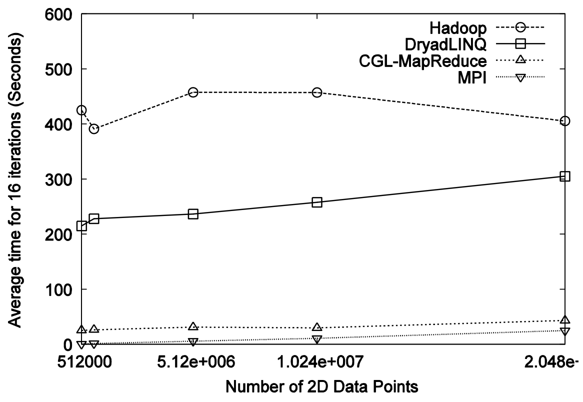

Figure 7. Performance of different implementations of clustering algorithm.

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 ameliorate the overheads introduced by Hadoop and DryadLINQ runtimes. As a result, the graph in Fig. 7 mainly shows the overhead of the 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 DryadLINQ) results in higher overheads compared to CGL-MapReduce and MPI. Iterative applications which perform more computations or access larger volumes of data may produce better results for Hadoop and DryadLINQ as the higher overhead induced by these runtimes becomes relatively less significant.

IV. Related Work

Cloud technologies adopts a more data centered approach to parallel programming compared to the traditional parallel runtimes such as MPI, Workflow runtimes, and individual job scheduling runtimes in which the scheduling decisions are made mainly by the availability of the computation resources. In cloud technologies the computations move to the locations of data to process them and are specifically designed to handle large volumes of data.

Parallel topologies supported by various parallel runtimes and the problems that can be implemented using these parallel topologies determine the applicability of many parallel runtimes to the problems in hand. For example, many job scheduling infrastructures such as TORQUE [14] and SWARM [15] can be used to execute parallel applications such as CAP3 consisting of a simple parallel topology of a collection of large number of independent tasks. Applications that perform parametric sweeps, document conversions, and brute-force searches are few other examples of this category. MapReduce programming model provides more parallel topologies than the simple job scheduling infrastructures with its support for the "reduction" phase. In typical MapReduce model, the outputs of the map tasks are partitioned using a hash function and assigned to a collection of reduce tasks. With the support of overloaded "key selectors" or hashes and by selecting the appropriate key selector function, this simple process can be extended to support additional models producing customized topologies under the umbrella of MapReduce model. For example, in the MapReduce version of tera-sort [16] application, Hadoop uses a customized hashing function to model the bucket sort algorithm.

Sector/Sphere [17] is a parallel runtime developed by Y. Gu, and R. L. Grossman that can be used to implement MapReduce style applications. Sphere adopts a streaming based computation model used in GPUs which can be used to develop applications with parallel topologies as a collection of MapReduce style applications. All Pairs [18] solves the specific problems of comparing elements in two data sets with each other and several other specific parallel topologies. We have used DryadLINQ to perform a similar computation to calculate pair-wise distances of a large collection of genes and our algorithm is explained in details in [19]. Swift [20] provides a scripting language and a execution and management runtime for developing parallel applications with the added support for defining typed data products via schemas. DryadLINQ allows user to define data types as C# structures or classes allowing users to handle various data types seamlessly with the runtime with the advantage of strong typing. Hadoop allows user to define "record readers" depending on the data that needs to be processed.

Parallel runtimes that support DAG based execution flows provide more parallel topologies compared to the mere MapReduce programming model or the models that support scheduling of large number of individual jobs. Condor DAGMan [21] is a well-known parallel runtime that supports applications expressible as DAGs and many workflow runtimes supports DAG based execution flows. However, the granularity of tasks handled at the vertices of Dryad/DryadLINQ and the tasks handled at map/reduce tasks in MapReduce is more fine grained than the tasks handled in Condor DAGMan and other workflow runtimes. This distinction become blurred when it comes to the parallel applications such as CAP3 where the entire application can be viewed as a collection of independent jobs, but for many other applications the parallel tasks of cloud technologies such as Hadoop and Dryad are more fine grained than the ones in workflow runtimes. For example, during the processing of the *GroupBy* operation used in DryadLINQ, which can be used to group a collection of records using a user defined key field, a vertex of the DAG generated for this operation may only process few records. In contrary the vertices DAGMan may be a complete programs performing considerable amount of processing.

Although in our analysis we compared DryadLINQ with Hadoop, DryadLINQ provides higher level language support for data processing than Hadoop. Hadoop's sub project Pig [22] is a more natural comparison to DryadLINQ. Our experience suggests that the scientific applications we used maps more naturally to Hadoop and Dryad (currently not available for public use) programming models than the high level runtimes such as Pig and DryadLINQ. However, we expect the high level programming models provided by the runtimes such as DryadLINQ and Pig are more suitable for applications that process structured data that can be fit into tabular structures.

Our work on CGL-MapReduce (we called it MapReduce++) extends capabilities of the MapReduce programming to applications that perform iterative MapReduce computations with minimum overheads. The use of streaming for communication and the support for cacheable map/reduce tasks enable MapReduce++ to operate with minimum overheads. Currently CGL-MapReduce does not provide any fault tolerance support for applications and we are investigating the mechanisms to support fault tolerance with the streaming based communication mechanisms we use.

Various scientific applications have been adapted to the MapReduce model in the past few years and Hadoop gained significant attention from the scientific research community. Kang et al. studied [23] efficient map reduce algorithms for finding the diameter of very large graphs and applied their algorithm to real web graphs. Dyer et al. described [24] map reduce implementations of parameter estimation algorithms to use in word alignment models and a phrase based translation model. Michael Schatz introduced CloudBurst for mapping short reads from sequences to a reference genome. In our previous works [3][25], 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 pleasingly parallel 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.

V. 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 K-means 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 make fine-tuning the applications more challenging.

Hadoop and DryadLINQ differ in their approach to fully utilize the many cores available on today's compute nodes. Hadoop allows scheduling of a worker process per core. On the other hand, DryadLINQ assigns vertices (i.e. worker processes) to nodes and achieves multi-core parallelism with PLINQ. The simplicity and flexibility of the Hadoop model proved effective for some of our benchmarks. The coarser granularity of scheduling offered by DryadLINQ performed equally well once we got a version DryadLINQ working with a newer build of the PLINQ library. Future releases of DryadLINQ and PLINQ will make those improvements available to the wider community. They will remove current needs for manual fine-tuning, which could also be alleviated by adding a tuning option that would allow a DryadLINQ user to choose the scheduling mode that best fits their workload.

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.

VI. Acknowledgements

We would like to thank the Dryad/DryadLINQ team at Microsoft Research, including Michael Isard, Yuan Yu, Mihai Budiu, and Derek Murray for their support in DryadLINQ applications, and Ed Essey, Igor Ostrovsky and Stephen Toub from the PLINQ team. We would also like to thank 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] Language Integrated Query (LINQ), http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

[9] 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.

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

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

[12] Parallel LINQ : Running Queries On Multi-Core Processors,http://msdn.microsoft.com/en-us/magazine/cc163329.aspx

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

[14] Torque Resource Manager, http://www.clusterresources.com/products/torque-resource-manager.php

[15] S. Pallickara, and M. Pierce. 2008. SWARM: Scheduling Large-Scale Jobs over the Loosely-Coupled HPC Clusters. Proc of *IEEE Fourth International Conference on eScience '08(eScience, 2008)*.Indianapolis, USA

[16] Tera-sort benchmark, http://sortbenchmark.org/

[17] Y. Gu, and R. L. Grossman. 2009. Sector and Sphere: the design and implementation of a high-performance data cloud. *Philos Transact A Math Phys Eng Sci* **367**(1897): 2429-45.

[18] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, "All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids," IEEE Transactions on Parallel and Distributed Systems, 13 Mar. 2009.

[19] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, G. Fox High Performance Parallel Computing with Clouds and Cloud Technologies Technical Report August 25 2009 to appear as Book Chapter

[20] Zhao Y., Hategan, M., Clifford, B., Foster, I., vonLaszewski, G., Raicu, I., Stef-Praun, T. and Wilde, M Swift: Fast, Reliable, Loosely Coupled Parallel Computation IEEE International Workshop on Scientific Workflows 2007

[21] Codor DAGMan, http://www.cs.wisc.edu/condor/dagman/.

[22] Apache Pig project, http://hadoop.apache.org/pig/

[23] 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.

[24] 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.

[25] 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.