Applicability of DryadLINQ to Scientific Applications

Salsa Group, Pervasive Technology Institute, Indiana University
http://salsaweb.ads.iu.edu/salsa/
Jan 30th 2010

Contents

1. Introduction 4

2. Overview 4

2.1 Microsoft DryadLINQ 4

2.2 Apache Hadoop 5

2.3 *Twister*: Iterate MapReduce 5

2.4 MPI 5

3. Performance and Usability of Applications using DryadLINQ 7

3.1 EST (Expressed Sequence Tag) sequence assembly program using DNA sequence assembly program software CAP3. 7

3.1.1 Evaluations and Findings 8

3.1.2 Inhomogeneity of data partitions and scheduling partitions to nodes 9

3.1.3 Threads vs. Processes (Issue No. 2) 11

3.2 Pairwise Alu sequence alignment using Smith Waterman GOTOH 12

3.2.1 ALU Clustering 12

3.2.2 Smith Waterman Dissimilarities 12

3.2.3 The $O(N^2)$ Factor of 2 and structure of processing algorithm 13

3.2.4 DryadLINQ Implementation 13

3.2.5 MPI Implementation 14

3.2.6 Performance of Smith Waterman Gotoh SW-G Algorithm 14

3.2.7 Apache Hadoop Implementation 15

3.2.8 Performance comparison of *DryadLINQ* and Hadoop SW-G implementations 16

3.2.9 Inhomogeneous data study 17

3.3 PhyloD Application 19

3.3.1 PhyloD Algorithm 19

3.3.2 DryadLINQ Implementation 19

3.3.3 DryadLINQ PhyloD Performance 21

3.4 HEP Processing large column of physics data using software Root and produce histogram results for data analysis. 23

3.4.1 Evaluations and Findings 24

3.5 K-means Clustering 26

3.5.1 Evaluations and Findings 26

3.5.2 Another Relevant Application - Matrix Multiplication 27
4. Analysis 29
4.1 DryadLINQ vs. Other Runtimes 29
4.1.1 Handling Data 29
4.1.2 Parallel Topologies 29
4.1.3 *Twister*: Iterate MapReduce 31
4.2 Performance and Usability of Dryad 31
4.2.1 Installation and Cluster Access 32
4.2.2 Developing and Deployment of Applications 32
4.2.3 Debugging 32
4.2.4 Fault Tolerance 33
4.2.5 Monitoring 33
5. Summary of key features of applications that suitable and not suitable for Dryad 34
References 36
Appendix A 38
Appendix B 38

1. Introduction

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.

The goal of our study is to explore the applicability of DryadLINQ to real scientific applications and compare its performance with other relevant parallel runtimes such as Hadoop. To achieve this goal we have developed a series of scientific applications using DryadLINQ, namely, CAP3 DNA sequence assembly program [1], Pairwise ALU sequence alignment, High Energy Physics(HEP)data analysis, and K-means Clustering [2]. Each of these applications has unique requirements for parallel runtimes. For example, the HEP data analysis application requires ROOT [3] data analysis framework to be available in all the compute nodes and in Pairwise ALU sequence alignment the framework must handle computing of distance matrix with hundreds of millions of points. We have implemented all these applications using DryadLINQ and Hadoop, and used them to compare the performance of these two runtimes. Twister and MPI are used in applications where the contrast in performance needs to be highlighted.

In the sections that follow, we first present an overview of the different parallel runtimes we use in this analysis followed by a detailed discussion of the data analysis applications we developed. Here we discuss the mappings of parallel algorithms to the DryadLINQ programming model and present performance comparisons with Hadoop implementations of the same applications. In section 4 we analyze DryadLINQ's programming model comparing it with other relevant technologies such as Hadoop and *Twister*. We also include a set of usability requirements for DryadLINQ. We present our conclusions in section 5.

2. Overview

This section presents a brief introduction to a set of parallel runtimes we use our evaluations.

2.1 Microsoft DryadLINQ

Dryad [4] is a distributed execution engine for coarse grain data parallel applications. Dryad considers computation tasks as directed acyclic graphs (DAG) where the vertices represent computation tasks and while the edges acting as communication channels over which the data flow from one vertex to another. In the HPC version of Dryad the data is stored in (or partitioned to) Windows shared directories in local compute nodes and a meta-data file is use to produce a description of the data distribution and replication. Dryad schedules the execution of vertices depending on the data locality. Dryad also stores the output of vertices in local disks, and the other vertices which depend on these results, access them via the shared directories. This enables Dryad to re-execute failed vertices, a step which improves the fault tolerance in the programming model.

DryadLINQ[5] is a higher level language layer for Dryad aimed at distributed data processing. DryadLINQ has the ability to translate LINQ programs written using existing .NET programming language constructs in to distributed Dryad computations. The academic release of Dryad only

exposes the DryadLINQ API for programmers. Therefore, all our implementations are written using DryadLINQ although it uses Dryad as the underlying runtime.

2.2 Apache Hadoop

Apache Hadoop [6] has a similar architecture to Google's MapReduce runtime [8], 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 across all the data/computing nodes. HDFS also replicates the data on multiple nodes so that failures of any nodes containing a portion of the data will not affect the computations which use that data. Hadoop schedules the MapReduce computation tasks depending on the data locality, improving 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 a significant communication overhead to the intermediate data transfers, especially for applications that produce small intermediate results frequently.

Apache Hadoop Pig [7] project with its Pig Latin higher level language layer for data analysis is the DryadLINQ counterpart for Hadoop. However, all our implementations are currently written only using Hadoop Map Reduce.

2.3 *Twister*: Iterate MapReduce

Twister [9][10] is a light-weight MapReduce runtime (earlier called CGL-MapReduce) that incorporates several improvements to the MapReduce programming model such as (i) faster intermediate data transfer via a pub/sub broker network; (ii) support for long running *map/reduce* tasks; and (iii) efficient support for iterative MapReduce computations. The use of streaming enables *Twister* to send the intermediate results directly from its producers to its consumers, and eliminates the overhead of the file based communication mechanisms adopted by both Hadoop and DryadLINQ. The support for long running *map/reduce* tasks enables configuring and re-using of *map/reduce* tasks in the case of iterative MapReduce computations, and eliminates the need for the re-configuring or the re-loading of static data in each iteration.

2.4 MPI

MPI [11], the de-facto standard for parallel programming, is a language-independent communications protocol that uses a message-passing paradigm to share the data and state among a set of cooperative processes running on a distributed memory system. MPI specification (F defines a set of routines to support various parallel programming models such as point-to-point communication, collective communication, derived data types, and parallel I/O operations. Most MPI runtimes are deployed in computation clusters where a set of compute nodes are connected via a high-speed network connection yielding very low communication latencies (typically in microseconds). MPI processes typically have a direct mapping to the available processors in a compute cluster or to the processor cores in the case of multi-core systems.. We use MPI as the baseline performance measure for the various algorithms that are used to evaluate the different parallel programming runtimes. Table 1 summarizes the different characteristics of Hadoop, Dryad, *Twister*, and MPI.

Table 1. Comparison of features supported by different parallel programming runtimes.

Feature	Hadoop	DryadLINQ	Twister	MPI
Programming Model	MapReduce	DAG based execution flows	MapReduce with a *Combine* phase	Variety of topologies constructed using the rich set of parallel constructs
Data Handling	HDFS	Shared directories/ Local disks	Shared file system / Local disks	Shared file systems
Intermediate Data Communication	HDFS/ Point-to-point via HTTP	Files/TCP pipes/ Shared memory FIFO	Content Distribution Network (NaradaBrokering (Pallickara and Fox 2003))	Low latency communication channels
Scheduling	Data locality/ Rack aware	Data locality/ Network topology based run time graph optimizations	Data locality	Available processing capabilities
Failure Handling	Persistence via HDFS Re-execution of map and reduce tasks	Re-execution of vertices	Currently not implemented (Re-executing map tasks, redundant reduce tasks)	Program level Check pointing OpenMPI , FT MPI
Monitoring	Monitoring support of HDFS, Monitoring MapReduce computations	Monitoring support for execution graphs	Programming interface to monitor the progress of jobs	Minimal support for task level monitoring
Language Support	Implemented using Java. Other languages are supported via Hadoop Streaming	Programmable via C# DryadLINQ provides LINQ programming API for Dryad	Implemented using Java Other languages are supported via Java wrappers	C, C++, Fortran, Java, C#

3. Performance and Usability of Applications using DryadLINQ

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 our benchmarks, we used three clusters with almost identical hardware configurations with 256 CPU cores in each and a large cluster with 768 cores as shown in Table 2.

Table 2. Different computation clusters used for this analysis.

Feature	Linux Cluster (Ref A)	Windows Cluster (Ref B)	Windows Cluster (Ref C)	Windows Cluster (Ref D)
CPU	Intel(R) Xeon(R) CPU L5420 2.50GHz	Intel(R) Xeon(R) CPU L5420 2.50GHz	Intel(R) Xeon(R) CPU L5420 2.40GHz	Intel(R) Xeon(R) CPU L5420 2.50GHz
# CPU # Cores	2 8	2 8	4 6	2 8
Memory	32GB	16 GB	48 GB	32 GB
# Disk	1	2	1	1
Network	Giga bit Ethernet	Giga bit Ethernet	Giga bit Ethernet	Giga bit Ethernet
Operating System	Red Hat Enterprise Linux Server release 5.3 -64 bit	Microsoft Window HPC Server 2008 (Service Pack 1) - 64 bit	Microsoft Window HPC Server 2008 (Service Pack 1) - 64 bit	Microsoft Window HPC Server 2008 (Service Pack 1) - 64 bit
# Cores	256	256	768	256

3.1 EST (Expressed Sequence Tag) sequence assembly using CAP3 DNA sequence assembly software

CAP3[1] 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.)

3.1.1 Evaluations and Findings

We developed CAP3 data analysis applications for Hadoop using only the *map* stage of the MapReduce programming model. In these implementations, the *map* function simply calls the CAP3 executable passing the input data file names. We evaluated DryadLINQ and Hadoop for the CAP3 application and evaluated its scalability by measuring the program execution times varying the number of data files. Figure 1 shows comparisons of performance and the scalability of the DryadLINQ CAP3 application, with the Hadoop CAP3 application. For these evaluations we ran DryadLINQ applications in cluster ref D and Hadoop applications in cluster ref A, which essentially are the same cluster booted to different operating systems. It should be noted that the standalone CAP3 program ran approximately 12.5% slower on the Linux environment than on the windows environment

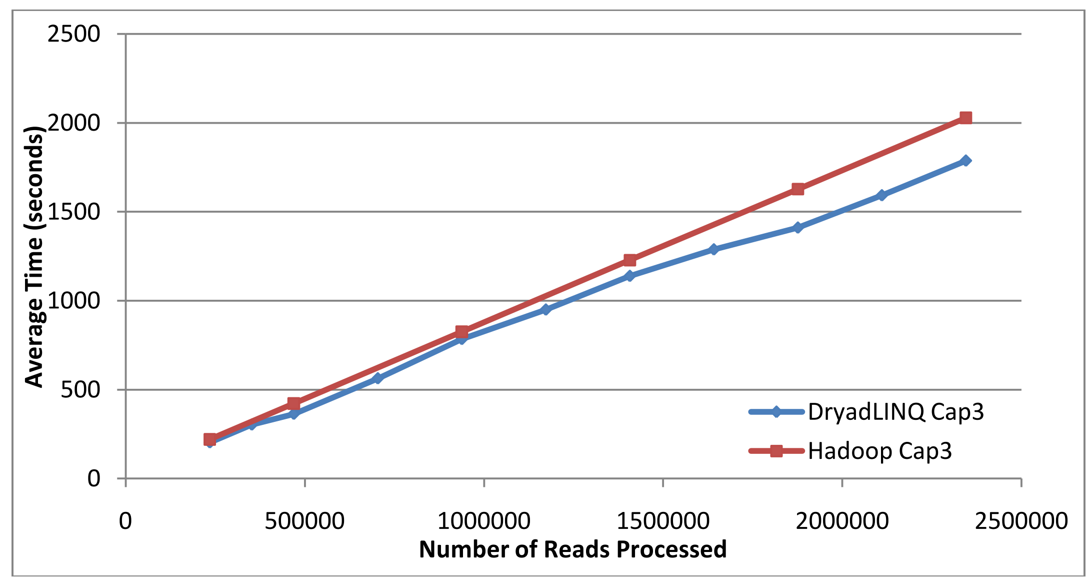

Figure 1. Performance of different implementations of CAP3 application.[35]

The performance and the scalability graphs shows that both 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. Except for the manual data partitioning requirement, implementing this type of applications using DryadLINQ is extremely simple and straightforward.

During this analysis we identified two issues related to DryadLINQ and the software it uses, which we will discuss in the coming sections. They are:

Issue No. 1 DryadLINQ schedule jobs to nodes rather than cores – idle cores when the data is inhomogeneous.

Issue No. 2 Performance of threads is extremely low for memory intensive operations compared to processes.

3.1.2 Inhomogeneity of data partitions and scheduling partitions to nodes

DryadLINQ schedules vertices of the DAG (corresponding to data partitions) to compute nodes rather than individual CPU cores **(Issue No. 1).** This may also produce suboptimal CPU utilizations of Dryad programs depending on the data partition strategy. As in MapReduce programming model, Dryad also assumes that the vertices corresponding to a given phase of the computation partitions data so that the data is distributed evenly across the computation nodes. Although this is possible in some computations such as sorting and histogramming where the data can be divisible arbitrary, it is not always possible when there are inhomogeneous data products at the lowest level of the data items such as gene sequences, binary data files etc.. For example, CAP3 process sequence data as a collection of FASTA files and the number of sequences containing in each of these files may differ significantly causing imbalanced workloads.

Since DryadLINQ schedules vertices to nodes, it is possible that a vertex which processes few large FASTA files using few CPU cores of a compute node will keep all the other CPU cores of that machine idle. In Hadoop, the map/reduce tasks are scheduled to individual CPU cores (customized by the user) and hence it is able to utilize all the CPU cores to execute map/reduce tasks in a given time.

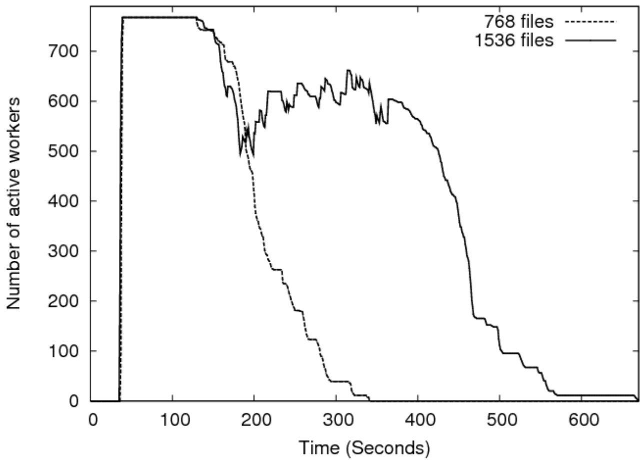

Figure 2. Number of active tasks/CPU cores along the running times of two runs of CAP3.

The collection of input files we used for the benchmarks contained different number of gene sequences in each, and hence it did not represent a uniform workload across the concurrent vertices of the DryadLINQ application, because the time the CAP3 takes to process an input file varies depending on the number of sequences available in the input file. The above characteristics of the data produces lower efficiencies at higher number of CPU cores as more CPU cores become idle towards the end of the computation waiting for vertices that takes longer time to complete.

To verify the above observation we measured the utilization of vertices during two runs of the CAP3 program. In our first run we used 768 input files so that Dryad schedules 768 vertices on 768 CPU cores, while in the second Dryad schedules 1536 vertices on 768 CPU cores. The result of this benchmark is shown in figure 2. The first graph in figure 3 corresponding to 768 files indicates that although DryadLINQ starts all the 768 vertices at the same time they finish at different times with long running tasks taking roughly 40% of the overall time. The second graph (1536 files) shows that the above effect has caused lower utilization of vertices when Dryad schedules 1536 vertices to 768 CPU cores.

For our next experiment, we created data sets with different standard deviations to further study the effects of inhomogeneous data to our CAP3 applications. CAP3 program execution time depends mainly on the content of the sequences. When generating the data sets, we first calculated the standalone CAP3 execution time for each of the files in our real data set. Then, based on those timings, we created data sets that have approximately similar mean times, while the standard deviation of the standalone running times is different in each data set. We performed the performance testing for randomly distributed as well as skewed distributed (sorted according to individual file running time) data sets. The speedup is taken by dividing the sum of standalone running times of the files in the data set on the respective environments by the parallel implementation running time.

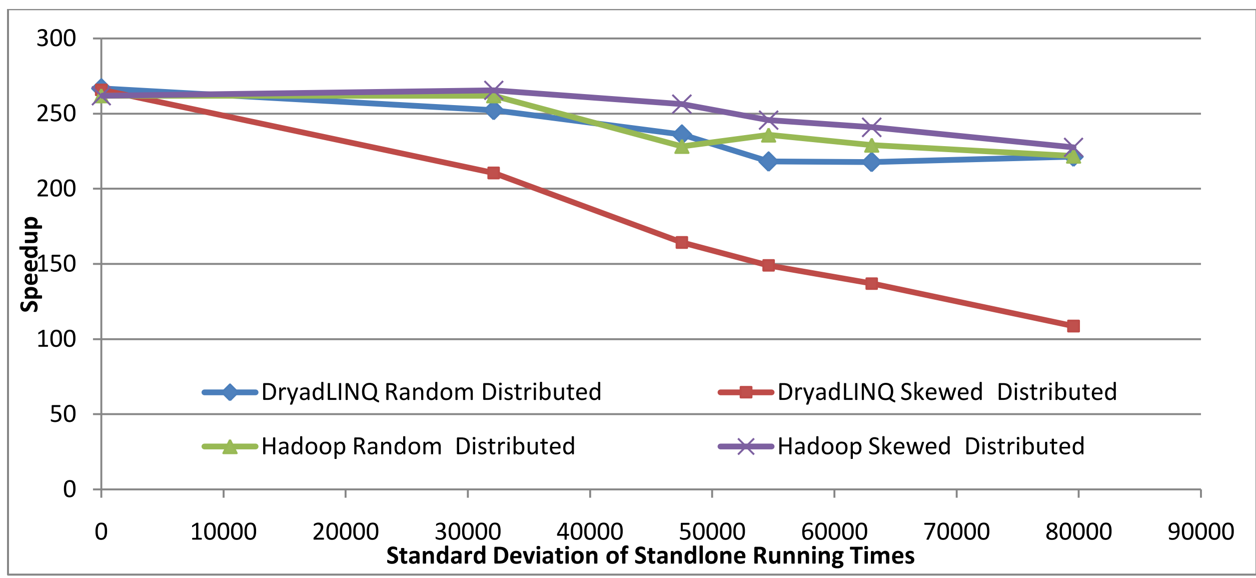

Figure 3. Cap3 DryadLINQ implementation performance against inhomogeneous data.[35]

In the figure Figure 3 we can notice that both the applications perform well when the files in a data set are randomly distributed. The reason for the above behavior is, when a data set is randomly distributed it provides a natural load balancing among the data partitions.

The DryadLINQ application performance degrades when the files in the data set are skew distributed, while the Hadoop CAP3 application performance is much better. Hadoop performs better as a result of its' dynamic global pipeline scheduling of map tasks providing a run time global load balancing. The DryadLINQ skew distributed results further confirm the issue no 1. One possible remedy for this issue is to distribute the data among the partitions randomly without following any order.

3.1.3 **Threads vs. Processes (Issue No. 2)**

When we develop CAP3 and similar applications, we noticed that the applications perform far better when the functions/programs which are executed using `Select` or `Apply` constructs are executed as processes than just as functions in the same program. i.e. executed using threads via PLINQ. Consider the following simple PLINQ program segment.

```
IEnumerable<int> inputs = indices.AsEnumerable();
IEnumerable<int> outputs =
      ParallelEnumerable.Select(inputs.AsParallel(), x => Func_X (x));
```

Variations of `Func_X` are:

1. `Func_ComputeIntensive()`
2. `Func_ComputeIntensiveProcesses()`
3. `Func_MemoryIntensive()`
4. `Func_MemoryIntensiveProcesses()`

The difference between the `Func_ComputeIntensive()` and `Func_ComputeIntensiveProcesses()` is that the second function calls the first function as a separate executable (process). Similarly the `Func_MemoryIntensiveProcesses ()` calls `Func_MemoryIntensive()` as a separate process.

The `Func_ComputeIntensive()` simply multiply double value `pi` in a loop to produce an artificial compute intensive task. The `Func_MemoryIntensive()` function allocates and de-allocates small 2D arrays (about 300 by 300 elements) with floating point computations in-between resembling a function in many gene analyses such as Smith Waterman or CAP3. The `Func_MemoryIntensive()` does not try to utilize all the memory or let the computer in to the thrashing mode. (Note: these functions are shown in Appendix A and Appendix B of this document.)

We ran the above simple program with four different functions mentioned above to understand the effect of threads vs. processes for compute intensive and memory intensive functions. In this analysis we directly used PLINQ without using DryadLINQ to perform the above query in a multi-core computer with 24 CPU cores. This helped us to isolate the performance issue in threads vs. processes better.

We made the following observations:

1. For compute intensive workloads, threads and processes did not show any significant performance difference.

2. For memory intensive workloads, processes perform about 20 times faster than threads.

The main reason for the extremely poor performance of threads is due to the large number of context switches occur when a memory intensive operation is used with threads. (**Note: We verified this behavior with both the latest version of PLINQ and the previous version of PLINQ**). Following table (Table 3) shows the results.

Table 3. Performance of threads and processes.

Test Type	Total Time (Seconds)	Context Switches	Hard Page Faults	CPU utilization
Func_MemoryIntensive()	133.62	100000-110000	2000-3000	76%
Func_MemoryIntensiveProcesses()	5.93	5000-6000	100-300	100%
Func_ComputeIntensive()	15.7	<6000	<110	100%
Func_ComputeIntensiveProcesses()	15.73	<6000	<110	100%

From table 3, it is evident that although we noticed a 76% CPU utilization in the case of `Func_MemoryIntensive()`, most of the time the program is doing context switches rather than useful work. On the other hand, when the same function is executed as separate processes; all the CPU cores were used to perform the real application.

We observed these lower CPU utilizations in most of the applications we developed, and hence we made the functions that perform scientific analysis into separate programs and executed as processes using DryadLINQ.

3.2 Pairwise Alu sequence alignment using Smith Waterman GOTOH

3.2.1 ALU Clustering

The ALU clustering problem [26] is one of the most challenging problems for sequencing clustering because ALUs represent the largest repeat families in human genome. There are about 1 million copies of ALU sequences in human genome, in which most insertions can be found in other primates and only a small fraction (~ 7000) are human-specific. This indicates that the classification of ALU repeats can be deduced solely from the 1 million human ALU elements. Notable, ALU clustering can be viewed as a classical case study for the capacity of computational infrastructures because it is not only of great intrinsic biological interests, but also a problem of a scale that will remain as the upper limit of many other clustering problem in bioinformatics for the next few years, e.g. the automated protein family classification for a few millions of proteins predicted from large metagenomics projects.

3.2.2 Smith Waterman Dissimilarities

We identified samples of the human and Chimpanzee ALU gene sequences using Repeatmasker [27] with Repbase Update [28]. We have been gradually increasing the size of our projects with the current largest samples having 35339 and 50000 sequences and these require a modest cluster such as ref C in table 2 (768 cores) for processing in a reasonable time (a few hours as shown in

table 4). We are aiming at supporting problems with a million sequences -- quite practical today on TeraGrid and equivalent facilities given basic analysis steps scale like $O(N^2)$.

We used open source version NAligner [29] of the Smith Waterman – Gotoh algorithm SW-G [30][31] modified to ensure low start up effects by each thread/processing large numbers (above a few hundred) at a time. Memory bandwidth needed was reduced by storing data items in as few bytes as possible.

3.2.3 The $O(N^2)$ Factor of 2 and structure of processing algorithm

The ALU sequencing problem shows a well known factor of 2 issue present in many $O(N^2)$ parallel algorithms such as those in direct simulations of astrophysical stems. We initially calculate in parallel the Distance D(i,j) between points (sequences) i and j. This is done in parallel over all processor nodes selecting criteria $i < j$ (or $j > i$ for upper triangular case) to avoid calculating both D(i,j) and the identical D(j,i). This can require substantial file transfer as it is unlikely that nodes requiring D(i,j) in a later step will find that it was calculated on nodes where it is needed.

For example the MDS and PW (PairWise) Clustering algorithms described in [10], require a parallel decomposition where each of N processes (MPI processes, threads) has 1/N of sequences and for this subset {i} of sequences stores in memory D({i},j) for all sequences j and the subset {i} of sequences for which this node is responsible. This implies that we need D (i,j) and D (j,i) (which are equal) stored in different processors/disks. This is a well known collective operation in MPI called either gather or scatter.

3.2.4 DryadLINQ Implementation

We developed a DryadLINQ application to perform the calculation of pairwise SW-G distances for a given set of genes by adopting a coarse grain task decomposition approach which requires minimum inter-process communicational requirements to ameliorate the higher communication and synchronization costs of the parallel runtime. To clarify our algorithm, let's consider an example where N gene sequences produces a pairwise distance matrix of size NxN. We decompose the computation task by considering the resultant matrix and groups the overall computation into a block matrix of size DxD where D is a multiple (>2) of the available computation nodes. Due to the symmetry of the distances D(i,j) and D(j,i) we only calculate the distances in the blocks of the upper triangle of the block matrix as shown in figure 5 (left). The blocks in the upper triangle are partitioned (assigned) to the available compute nodes and an "Apply" operation is used to execute a function to calculate (N/D)x(N/D) distances in each block. After computing the distances in each block, the function calculates the transpose matrix of the result matrix which corresponds to a block in the lower triangle, and writes both these matrices into two output files in the local file system. The names of these files and their block numbers are communicated back to the main program. The main program sort the files based on their block number s and perform another "Apply" operation to combine the files corresponding to a row of blocks in a single large row block as shown in the figure 4 (right).

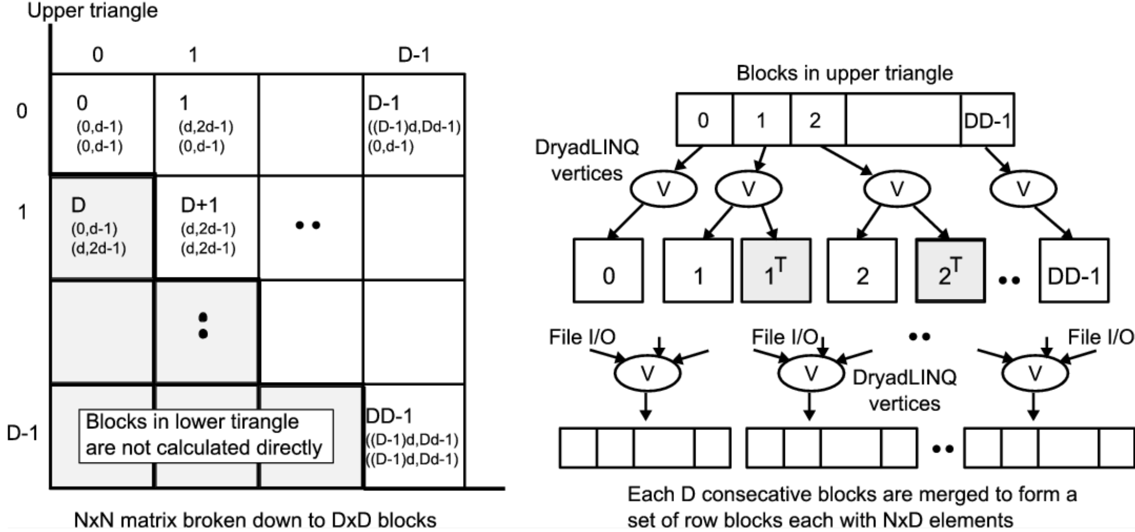

Figure 4. Task decomposition (left) and the Dryad vertex hierarchy (right) of the DryadLINQ implementation of SW-G pairwise distance calculation application.

3.2.5 **MPI Implementation**

The MPI version of SW-G calculates pairwise distances using a set of either single or multi-threaded processes. For N gene sequences, we need to compute half of the values (in the lower triangular matrix), which is a total of M = N x (N-1) /2 distances. At a high level, computation tasks are evenly divided among P processes and execute in parallel. Namely, computation workload per process is M/P. At a low level, each computation task can be further divided into subgroups and run in T concurrent threads. Our implementation is designed for flexible use of shared memory multicore system and distributed memory clusters (tight to medium tight coupled communication technologies such threading and MPI).

3.2.6 **Performance of Smith Waterman Gotoh SW-G Algorithm**

We performed the DryadLINQ and MPI implementations of ALU SW-G distance calculations on two large data sets and obtained the following results. Both these tests were performed in cluster *ref C*.

Table 4. Comparison of DryadLINQ and MPI technologies on ALU sequencing application with SW-G algorithm

Technology		Total Time (seconds)	Time per Pair (milliseconds)	Partition Data (seconds)	Calculate and Output Distance(seconds)	Merge files (seconds)
DryadLINQ	50,000 sequences	17200.413	0.0069	2.118	17104.979	93.316
	35,339 sequences	8510.475	0.0068	2.716	8429.429	78.33
MPI	50,000 sequences	16588.741	0.0066	N/A	13997.681	2591.06
	35,339 sequences	8138.314	0.0065	N/A	6909.214	1229.10

There is a short partitioning phase for *DryadLINQ* application and then both approaches calculate the distances and write out these to intermediate files as discussed in section 3.2.4. We note that merge time is currently much longer for MPI than *DryadLINQ* while the initial steps are significantly faster for MPI. However the total times in table 4 indicates that both MPI and *DryadLINQ* implementations perform well for this application with MPI a few percent faster with current implementations. As expected, the times scale proportionally to the square of the number of distances. On 744 cores the average time of 0.0067 milliseconds per pair that corresponds to roughly 5 milliseconds per pair calculated per core used. The coarse grained *DryadLINQ* application performs competitively with the tightly synchronized MPI application.

3.2.7 **Apache Hadoop Implementation**

We developed an Apache Hadoop version of the pairwise distance calculation program based on the JAligner[26] program, the java implementation of the NAligner. Similar to the other implementations, the computation is partitioned in to blocks based on the resultant matrix. Each of the blocks would get computed as a map task. The block size (D) can be specified via an argument to the program. The block size needs to specified in such a way that there will be much more map tasks than the map task capacity of the system, so that the Apache Hadoop scheduling will happen as a pipeline of map tasks resulting in global load balancing of the application. The input data is distributed to the worker nodes through the Hadoop distributed cache, which makes them available in the local disk of each compute node.

A load balanced task partitioning strategy according to the following rules is used to identify the blocks that need to be computed (green) through map tasks as shown in the figure 5(a). In addition all the blocks in the diagonal (blue) are computed. Even though the task partitioning mechanisms are different, both Dryad-SWG and Hadoop SW-G ends up with essentially identical computation blocks, if the same block size is given to both the programs.

If $\beta >= \alpha$*, we only calculate* $D(\alpha,\beta)$ *if* $\alpha+\beta$ *is even,*

If $\beta < \alpha$*, we only calculate* $D(\alpha,\beta)$ *if* $\alpha+\beta$ *is odd.*

The figure 5 (b) depicts the run time behavior of the Hadoop SW-G program. In the given example the map task capacity of the system is “k” and the number of blocks is “N”. The solid black lines represent the starting state, where “k” map tasks (blocks) will get scheduled in the compute nodes. The solid red lines represent the state at t_1 , when 2 map tasks, m_2 & m_6, get completed and two map tasks from the pipeline gets scheduled for the placeholders emptied by the completed map tasks. The dotted lines represent the future.

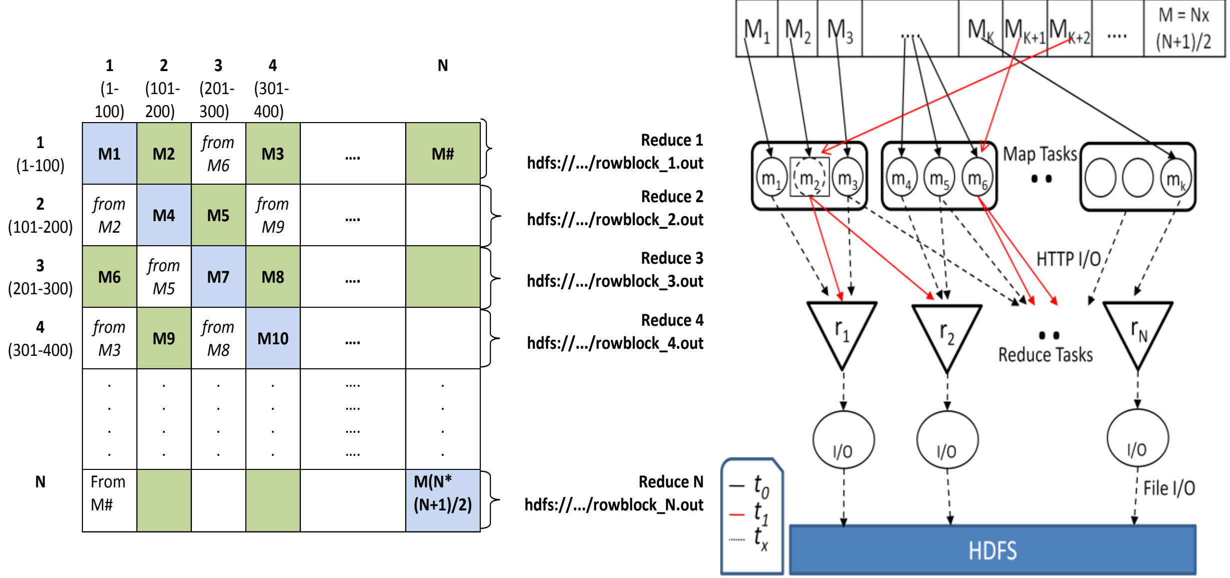

Figure 5. (a)Task (Map) decomposition and the reduce task data collection (b) Application run time

Map tasks use custom Hadoop writable objects as the map task output values to store the calculated pairwise distance matrices for the respective blocks. In addition, non-diagonal map tasks output the inverse distances matrix as a separate output value. Hadoop uses local files and http transfers underneath to transfer the map task output key value pairs to the reduce tasks.

The outputs of the map tasks are collected by the reduce tasks. Since the reduce tasks start collecting the outputs as soon as the first map task finishes and continue to do so while other map tasks are executing, the data transfers from the map tasks to reduce tasks do not present a significant performance overhead to the program. The program currently creates a single reduce task per each row block resulting in total of (no. of sequences/block size) Reduce tasks. Each reduce task to accumulate the output distances for a row block and writes the collected output to a single file in Hadoop Distributed File System (HDFS). This results in N number of output files corresponding to each row block, similar to the output we produce in the *DryadLINQ* version.

3.2.8 **Performance comparison of *DryadLINQ* and Hadoop SW-G implementations**

We compared the *DryadLINQ* and Hadoop implementations on the same data sets we used for the *DryadLINQ* and MPI comparisons, but on a different cluster. These tests were run on cluster *ref A* for Hadoop SW-G and on *ref D* for DryadLINQ SW-G, which are two identical Linux and Windows clusters. The DryadLINQ-adjusted results row represents the performance timings adjusted for the performance difference of the base programs, NAligner and the JAligner. In here we do not present separate times for the merge step as the Hadoop implementation performs the merging with reduce tasks even when the map tasks are running. Table 5 shows the results of this comparison.

Table 5. Comparison of *DryadLINQ* and Hadoop technologies on ALU sequencing application with SW-G algorithm

Technology	No. of Sequences	Total Time (seconds)	Time per Pair (ms)	No of actual Alignments	Sequential Time (seconds)	Speedup	Speedup per core
DryadLINQ	50,000	30881.74	0.0124	1259765625	4884111.33	158.16	61.78%
	35,339	14654.41	0.0117	634179061	2458712.22	167.78	65.54%
DryadLINQ –adjusted	50,000	24202.4	0.0097	1259765625	3827736.66	158.16	61.78%
	35,339	11484.84	0.0092	634179061	1926922.27	167.78	65.54%
Hadoop	50,000	17798.59	0.0071	1262500000	4260937.50	239.40	93.51%
	35,339	8974.638	0.0072	629716021	2125291.57	236.81	92.50%

We can notice that the Hadoop implementation shows more speedup per core than the *DryadLINQ* implementation. In an alternate ongoing testing we are noticing that the block size plays a larger role with regards to the *DryadLINQ* implementation performance, where relatively smaller block sizes are performing better. This led us to speculate that the lower speedup of *DryadLINQ* implementation is related to the memory usage. We are currently pursuing this issue more deeply to understand the reasons for this behavior.

3.2.9 **Inhomogeneous data study**

The time complexity to align and obtain distances for two genome sequences with lengths 'm' and 'n' using SW-G algorithm is proportional to the product of the lengths of two sequences, O(mn). This makes the sequence length distribution of a block to determine the execution time for that particular execution block. Frameworks like Dryad and Hadoop work optimally when the work is equally partitioned among the tasks, striving for equal length sequences in the case of pairwise distance calculations. Depending on the scheduling strategy of the framework, blocks with different execution times can have an adverse effect on the performance of the applications, unless proper load balancing measures have been taken in the task partitioning steps. For an example, in DryadLINQ vertices are scheduled at the node level, making it possible for a node to have blocks with varying execution times. In this case if a single block inside a vertex takes a larger amount of time than other blocks to execute, then the whole node have to wait till the large task completes, which utilizes only a fraction of the node resources.

Sequence sets that we encounter in the real data sets are inhomogeneous in length. In this section we study the effect of inhomogeneous gene sequence lengths for our pairwise distance calculation applications. The data sets used were randomly generated with a given mean sequence length (400) with varying standard deviations following a normal distribution of the sequence lengths. Each data set contained a set of 10000 sequences, 100 million pairwise distance calculations to perform. We performed this experiment by distributing the sequences of varying lengths randomly across the data set as well as by distributing them in a sorted order based on the sequence length.

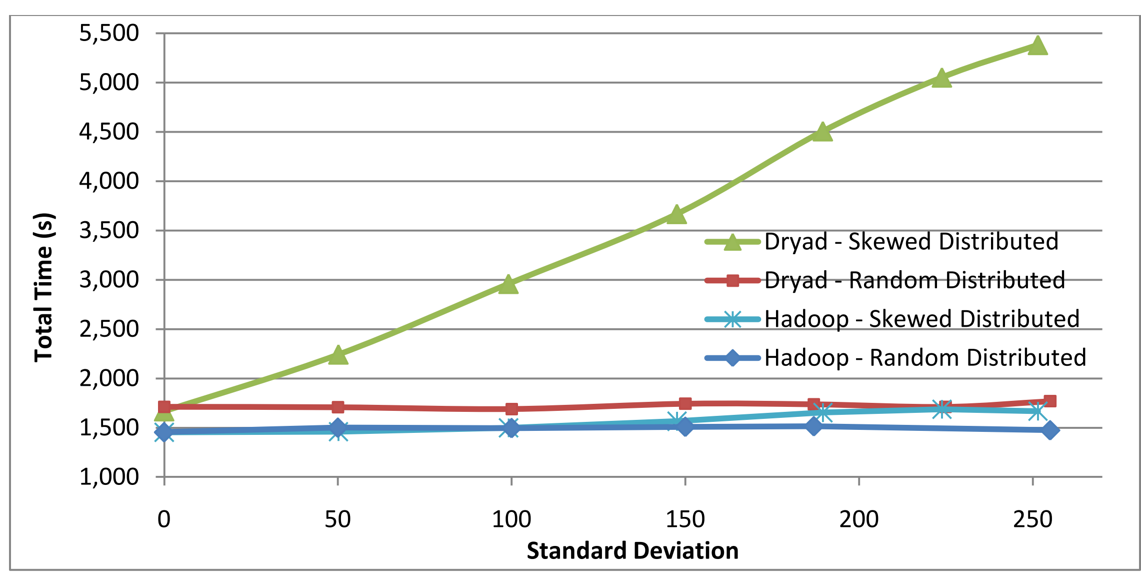

Figure 6. Performance of SW-G pairwise distance calculation application for inhomogeneous data.[35]

The DryadLINQ-adjusted results depict the raw DryadLINQ results adjusted for the performance difference of the NAligner and JAligner base programs. As we notice from the figure 6, both DryadLINQ implementation as well as the Hadoop implementation performed satisfactorily for the randomly distributed data, without showing significant performance degradations. In fact Hadoop implementation showed minor improvements in the execution times. The acceptable performance can be attributed to the fact that the sequences with varying lengths are randomly distributed across the data set, giving a natural load balancing to the sequence blocks. Similar to the results we noticed in the DryadLINQ CAP3 application, the DryadLINQ SW-G performance degrades when the data is skew distributed, due to a result of Dryad static scheduling of data partitions.

The Hadoop implementations' better performance can be attributed to the global pipeline scheduling of map tasks that Hadoop performs. In Hadoop administrator can specify the map task capacity of a particular worker node and then Hadoop global scheduler schedules the map tasks directly on to those placeholders in a much finer granularity than in Dryad as and when individual tasks finish. This allows the Hadoop implementation to perform natural global level load balancing. In this case it might even be advantageous to have varying task execution times to iron out the effect of any trailing map tasks towards the end.

3.3 PhyloD Application

The **H**uman **L**eukocyte **A**ntigen can help to eliminate the HIV virus. However, the HIV virus can avoid the elimination by evolution of escape mutation. HIV mutations can be considered as HIV codons changing or evolution. The PhyloD[33] application uses statistical method to identify HLA-associated viral evolution from the sample data of HIV-infected individuals.

3.3.1 PhyloD Algorithm

PhyloD is a new statistical package to derive the association among HLA and HIV by counting given sample data. The PhyloD package have three kinds of input data: (i) the phylogenic tree information of the codons, (ii) the information about HLA alleles, and (iii) the information about HIV codons. A run of PhyloD job have three main steps. First, it computes a cross product of input files to produce all allele-codon pairs. Second, it computes the p-value for each pair, which is used to measure the association between allele-codon pair. Third, it computes a q-value per p-value, which is an indicative measure of the significance of the p-value.

The running time of PhyloD algorithm is a function of the number of different HLA alleles -|X|, the number of different HIV codons -|Y|, and the number of individuals in the study -N, which is equal to the number of leaves of the phylogenic tree. To calculate p-value of one allele-codon pair, it will cost O(NlogN) and there are |X|*|Y| allele-codon pairs. So the PhyloD algorithm runs in time O(|X|*|Y|*NlogN). The computation of the p-value of one pair can be done independently of other p-value computations. This makes it easy to implement a parallel version of PhyloD using DryadLINQ.

PhyloD executable allows user to divide the PhyloD job into a set of tasks each of which works on the assigned part of the HLA allele file and the HIV codon file. Assuming the set of HLA alleles is {A[0],A[1],..A[|X|-1]}, and the set of HIV codons is {C[0],C[1],…C[|Y|-1]}, then the set of HLA and HIV pairs is stored in the order of {(A[0],C[0]),(A[1],C[0]),(A2,C[0]),..(A[|X|-1],C[0]); (A[0],C[1]), (A[1],C[1]),(A[2],C[1]),.. (A[|X|-2],C[|Y|-1]),(A[|X|-1],C[|Y|-1])}. Accordingly, PhyloD executable divides the PhyloD job into N tasks (divide the set of all pairs into N partitions) in the same order. The index bounds of set of pairs of the Kth task can be calculated by following formulas.

```
Set B = (|X|*|Y|+N-1)/N;
If 0<=K<=N-2
    Start index: (Ai,Ci)    A[i] = A[K*B%|X|];              C[i] = C[K*B/|X|]
    End index:   (Aj,Cj)    A[j] = A[((K+1)*B – 1)%|X|];    C[j] = C[(K+1)*B/|X|]
If K=N-1
    Start index: (Ai,Ci)    A[i] = A[K*B%|X|];              C[i] = C[K*B/|X|];
    End index:   (Aj,Cj)    A[j] = A[|X|-1];                C[j] = C[|Y|-1];
```

3.3.2 DryadLINQ Implementation

We implemented a parallel version of the PhyloD application using DryadLINQ and the standalone PhyloD runtime available from Microsoft Research [33][34]. As mentioned above the first phase of the PhyloD computation requires calculation of p-values for each HLA alleles and HIV codons. To increase the granularity of the parallel tasks, we group the individual computations into computation blocks containing a number of HLA alleles and HIV codons. Next these groups of computations are performed as a set of independent computations using DryadLINQ's "Select"

construct. As the number of patients samples in each pair are quite different, the PhyloD tasks are inhomogeneous in running time. To ameliorate this effect we partitioned the data (computation blocks) randomly so that the assignment of blocks to nodes will happen randomly.

After completion of the first step of PhyloD, we get one output file for each task (computation block). The second step will merge the K*M output files together to get one final output file with the q-values of all pairs. The DryadLINQ PhyloD task decomposition and Dryad vertex hierarchy of the DryadLINQ PhyloD are shown in the Figure 7.

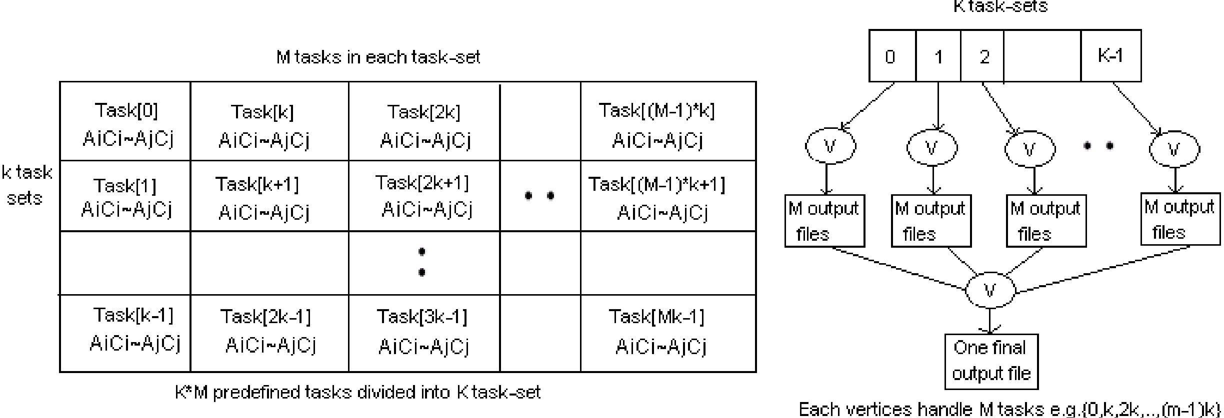

Figure 7. PhyloD task decompostion (left) and the Dryad vertex hierarchy (right) of the DryadLINQ implementation of PhyloD application

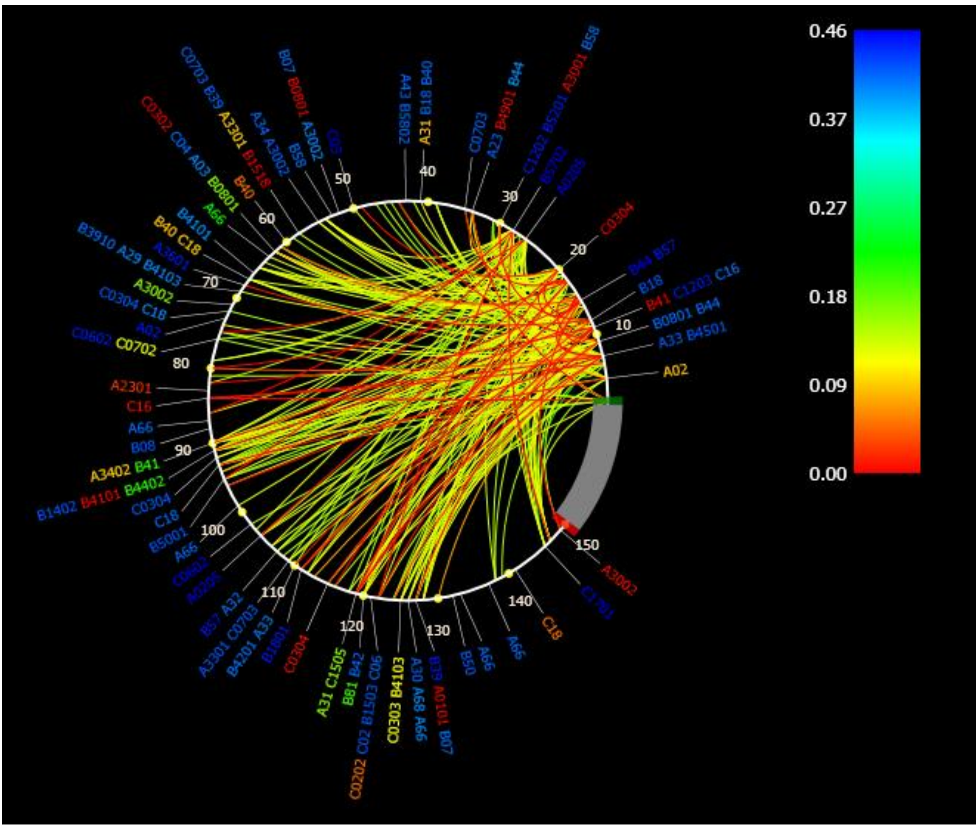

Figure 8. Part PhyloD DryadLINQ result about HIV Gag p17 and p24 protein codons

To explore the results of HLA-codon and codon-codon associations, Microsoft Research developed PhyloD viewer. Figure 8 is a PhyloDv[36] picture of part of PhyloD results for HIV Gag p17 and p24 protein codons with the DryadLINQ implementation. HLA-codon associations are drawn as external edges, whereas codon-codon associations are drawn as arcs within the circle. Colors indicate p-values of the associations. Some associations showed on this figure have already been well-studied by scientists before. For example, the B57 allele has been proved to be strongly associated with effective HIV control [32].

3.3.3 DryadLINQ PhyloD Performance

We investigated the scalability and speed up of DrayLINQ PhyloD implementation. The data set includes 136 distinct HLA alleles and 841 distinct HIV codons, resulting in 114376 HIL-HIV pairs. The cluster *Ref C* is used for these studies. Figure 9 depicts the speedup of running 114376 pairs on increasing number of cores. As the number of cores increase from 192 to 384, the speed up is not as good as cases with smaller number of cores. This increasing of overhead is due to the granularity becoming smaller with the increase of number of cores. The speed up would have been better on a larger data set.

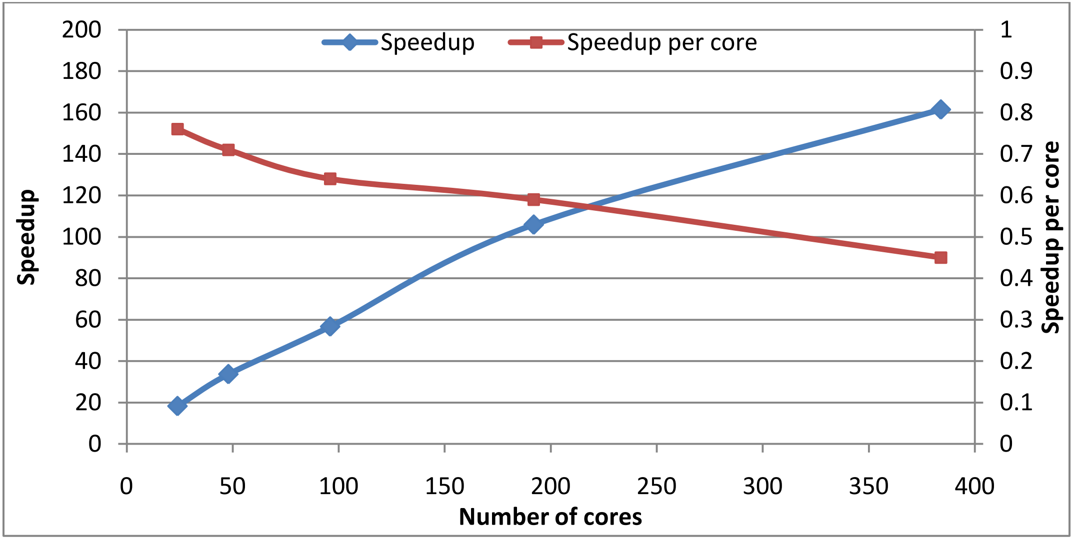

Figure 9. DryadLINQ PhyloD speedup on different number of cores for 114376 pairs of computation

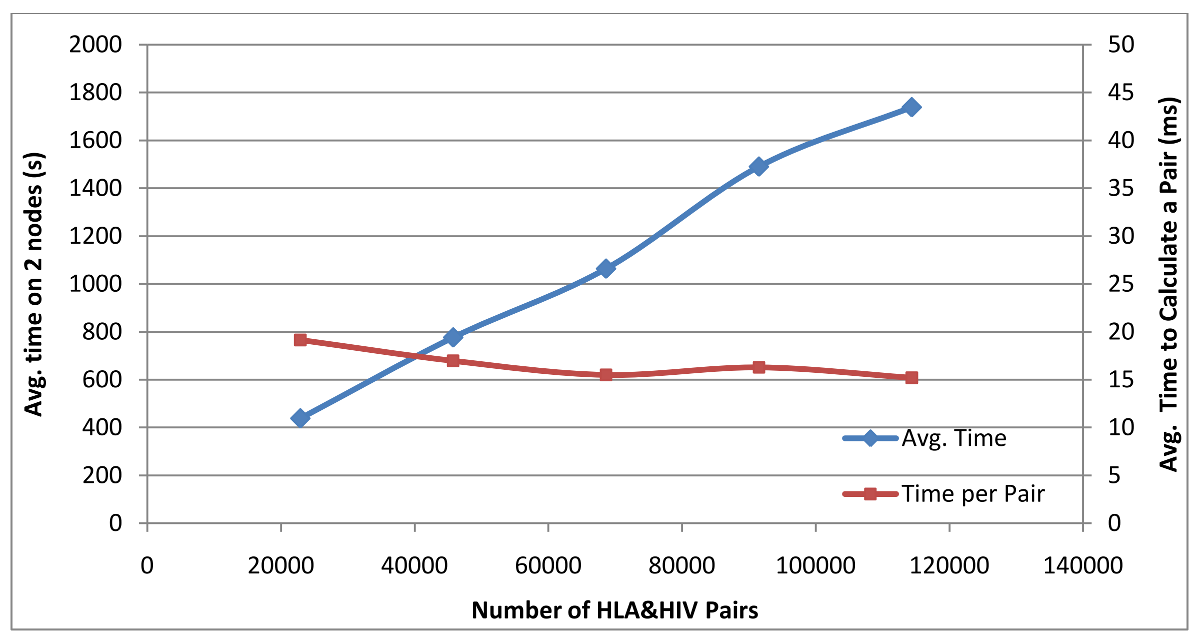

Figure 10. DryadLINQ PhyloD scalability with increase of dataset size

In the scalability experiment, we used data sets with increasing number of similar length tasks on a fixed number of cores (2 nodes* 24 cores). As shown in figure 10 the DryadLINQ PhyloD implementation scales well with the increase of data size.

The current data set we have is too small for a definitive study. We intend further study of the DryadLINQ PhyloD application behavior with larger data sets.

3.4 HEP Processing large column of physics data using software Root and produce histogram results for data analysis.

HEP data analysis application has a typical MapReduce application in which the *map* phase is used process a large collection of input files containing events (features) generated by HEP experiments. The output of the *map* phase is a collection of partial histograms containing identified features. During the reduction phase these partial histograms are merged to produce a single histogram representing the overall data analysis. Figure 11 shows the data flow of the HEP data analysis application.

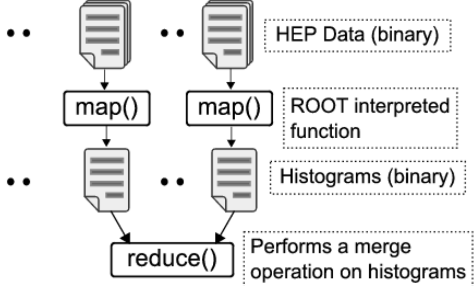

Figure 11. Program/data flow of the HEP data analysis application

Although the structure of this application is simple and fits perfectly well with the MapReduce programming model, it has a set of highly specific requirements such as:

1. All data processing functions are written using an interpreted language supported by ROOT [3] data analysis framework
2. All the data products are in binary format and passed as files to the processing scripts
3. Large input data sets (Large Hadron Collider will produce 15 petabytes of data per year).

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. 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.

3.4.1 Evaluations and Findings

The first task we had to tackle in the DryadLINQ implementation of this application is the distribution of data across the computation cluster. We used a data set of one terabytes (1TB) and hence storing and distributing this data set poses challenges. Typically these large data sets are stored in shared file systems and then get distributed to the computation nodes before the analysis. In this application the input data is organized in a large number of binary files each of which roughly occupy 33MB of disk space. Distributing a collection of data files across a computation cluster is a typical requirement in many scientific applications and we have already experienced this in CAP3 data analysis as well.

Current release of DryadLINQ does not provide any tools to do such data distribution. However, it provides two partitioning constructs which can be use to develop an application to perform this data distribution. One possible approach is to develop a DryadLINQ application to copy input files from its shared repository to individual computation units. This may saturate the shared repository infrastructure as all the compute nodes try to copy data from this shared location. We developed a standalone application to perform the above distribution as it can be used for many similar situations.

Hadoop provides and optimized solution to distributing data across computation nodes of a cluster via HDFS [6] and a client tool. The above data distribution reduces to the following simple command in the Hadoop environment.

```
bin/Hadoop –dfs put shared_repository_path destination_in_hdfs
```

We think that a similar tool for DryadLINQ would help users to partition data (available in files) more easily than developing custom solution for each application.

The second challenge we faced in implementing the above application is the use of ROOT data analysis framework to process data. This is also a common requirement in scientific analysis as many data analysis functions are written using specific analysis software such as ROOT, R, Matlab etc. To use these specific software at DryadLINQ vertices, they need to be installed in each and every compute node of the cluster. Some of these applications only require copying a collection of libraries to the compute nodes while some requires complete installations. `Clusrun` is a possible solution to handle both types of installations, however providing another simple tool to perform the first type of installations would benefit the users. (Note: we could ship few shared libraries or other necessary resources using `DryadLINQ.Resources.Add(resource_name)` method. However, this does not allow user to add a folder of libraries or a collection of folders. The ROOT installation requires copying few folders to every compute node)

After tackling the above two problems we were able to develop a DryadLINQ application for the HEP data analysis.

We measure the performance of this application with different input sizes up to 1TB of data and compare the results with Hadoop and *Twister* implementations that we have developed previously. The results of this analysis are shown in Figure 12.

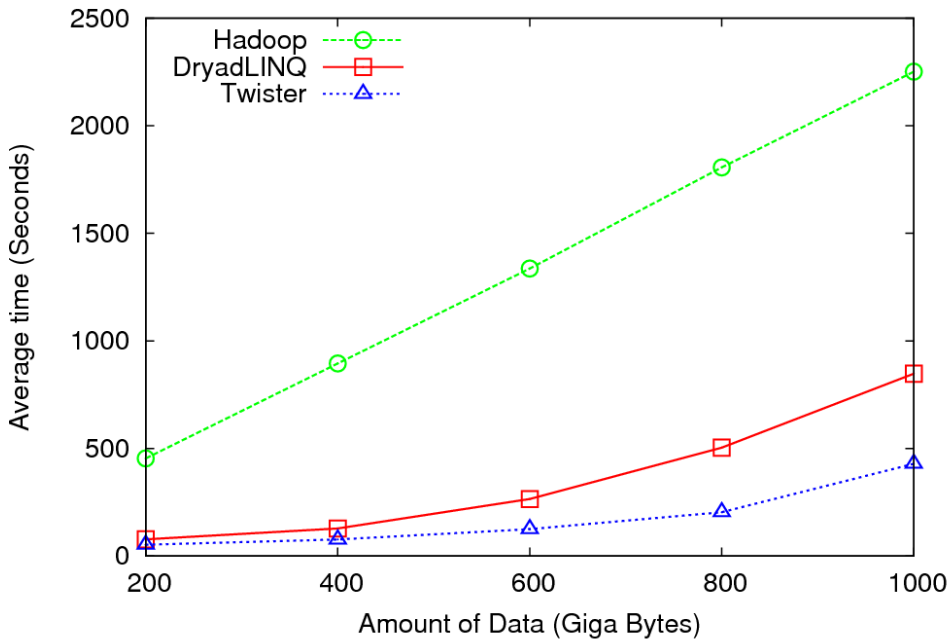

Figure 12. Performance of different implementations of HEP data analysis applications.

The results in Figure 12 highlight that Hadoop implementation has a considerable overhead compared to DraydLINQ and *Twister* implementations. This is mainly due to differences in the storage mechanisms used in these frameworks. DryadLINQ and *Twister* 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 *Twister* implementations.

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 *Twister* 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 *Twister* version and the DryadLINQ version of the program. We are planning to develop a better version of this application for DryadLINQ in the future.

3.5 K-means Clustering

We implemented a K-means Clustering [2] application using DryadLINQ to evaluate its performance under iterative computations. Algorithms such as clustering, matrix multiplication, Multi Dimensional Scaling [12] are some examples that performs 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 ith vector, $C_{n,j}$ refers to the jth 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 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 nth 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.

3.5.1 Evaluations and Findings

When implementing K-means algorithm using DryadLINQ we noticed that the trivial MapReduce style implementation of this algorithm perform extremely slow. We had to make several optimizations to the data structures and how we perform the calculations. One of the key changes is the use of `Apply` operation instead of `Select` to compare each data point with the current set of cluster centers. This enables DryadLINQ to consume an entire data partition at once and perform the comparisons. Figure 13 shows a comparison of performances of different implementations of K-means clustering.

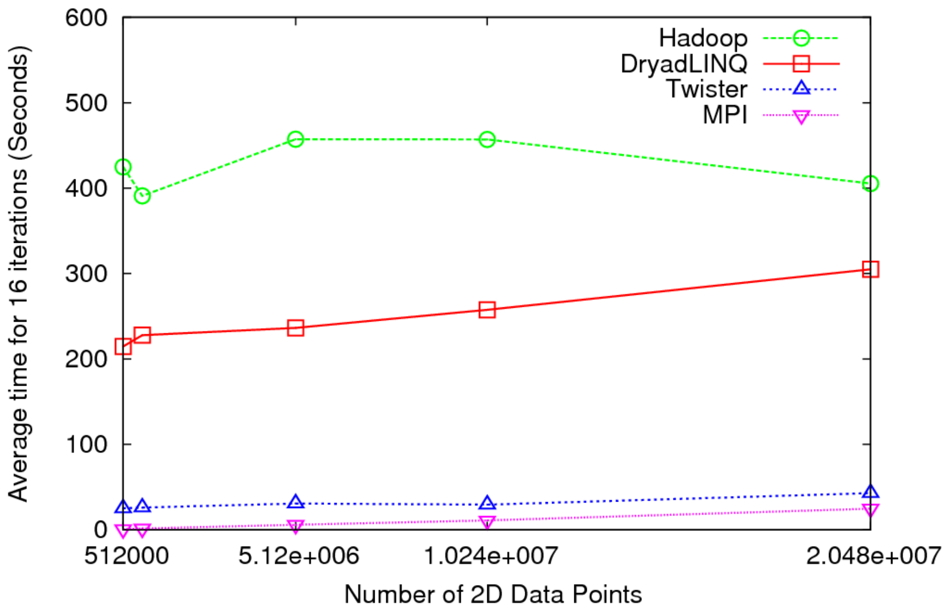

Figure 13. Performance of different implementations of clustering algorithm.

The performance graph shows that although DryadLINQ performs better than Hadoop for K-means application, still the average time taken by DryadLINQ and Hadoop implementations is extremely large compared to the MPI and the *Twister* implementations.

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 Figure 13 mainly shows the overhead of the different runtimes. With its loop unrolling feature, DryadLINQ does not need to materialize the outputs of the queries used in the program in every iteration. In the Hadoop implementation each iteration produces a new MapReduce computation increasing the total overhead of the implementation. 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 *Twister* 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. Currently the academic release uses file system based communication mechanism. However, according to the architecture discussed in Dryad paper [1], Dryad is capable of communicating via TCP pipes and therefore we expect better performances for this type of applications once it is supported by DryadLINQ as well.

3.5.2 Another Relevant Application - Matrix Multiplication

Parallel applications that are implemented using message passing runtimes can utilize various communication constructs to build diverse communication topologies. For example, a matrix multiplication application that implements Fox's Algorithm [13] and Cannon's Algorithm [14] assumes parallel processes to be in a rectangular grid. Each parallel process in the grid communicates with its left and top neighbors as shown in figure 14 (left). The current cloud runtimes, which are based on data flow models such as MapReduce and Dryad, do not support this

behavior, in which the peer nodes communicate with each other. Therefore, implementing the above type of parallel applications using MapReduce or Dryad requires adopting different algorithms.

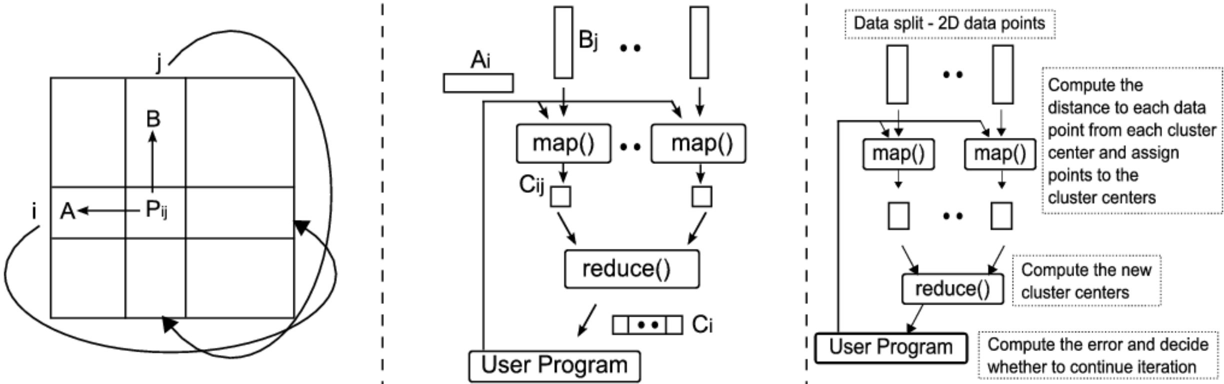

Figure 14. (Left) The communication topology of Cannon's Algorithm implemented using MPI, (middle) Communication topology of matrix multiplication application based on MapReduce, and (right) Communication topology of K-means Clustering implemented as a MapReduce application.

We have implemented matrix multiplication applications using Hadoop and *Twister* by adopting a row/column decomposition approach to split the matrices. To clarify our algorithm, let's consider an example where two input matrices, A and B, produce matrix C, as the result of the multiplication process. We split the matrix B into a set of column blocks and the matrix A into a set of row blocks. In each iteration, all the map tasks process two inputs: (i) a column block of matrix B, and (ii) a row block of matrix A; collectively, they produce a row block of the resultant matrix C. The column block associated with a particular map task is fixed throughout the computation, while the row blocks are changed in each iteration. However, in Hadoop's programming model (a typical MapReduce model), there is no way to specify this behavior. Hence, it loads both the column block and the row block in each iteration of the computation. *Twister* supports the notion of long running map/reduce tasks where these tasks are allowed to retain static data in the memory across invocations, yielding better performance for "Iterative MapReduce" computations. The communication pattern of this application is shown in figure 14 (middle). We haven't implemented a Matrix multiplication application using DryadLINQ yet and plan to do so in the future.

4. Analysis

4.1 DryadLINQ vs. Other Runtimes

4.1.1 Handling Data

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. DryadLINQ starts its computation from a partition table adapting the same data centered approach and try to schedule computations where the data is available.

In DryadLINQ the data is partitioned to the shared directories of the computation nodes of the HPC cluster where all the nodes have access to these common directories. With the support from a *partitioned file* DryadLINQ builds the necessary meta-data to access these data partitions and it also supports replicated data partitions to improve the fault tolerance. As we have discussed under sections 3.1 and 3.3.1 with the current release of DryadLINQ the partitioning of the existing data (either in individual files or in large data items) needs to be handled by the user manually. Comparatively, Apache Hadoop comes with a distributed file system that can be deployed on top of a set of heterogeneous resources, and a set of client tools to perform necessary file system operations. With this the user is completely shielded from the locations where the data is stored and its fault tolerance functionalities. *Twister* also adopts a DryadLINQ style meta-data model to handle data partitions and currently supports file based data types.

Although the use of a distributed file system in Hadoop makes the data partitioning and managing much easier, not all the applications benefit from this approach. For example, in HEP data analysis, the data is processed via a specialized software framework named ROOT which needs to access data files directly from the file system, but Hadoop provides only Java and C++ API to access HDFS. We used a shared parallel file system (Lustre) deployed at Indiana University to store HEP data and this resulted higher overheads in the Hadoop implementation. Apache subprojects such as FUSE [15] allows HDFS to be mounted as a shared file system but these approaches do not support the concept of "moving computation to data" rather use HDFS as a just another distributed file system. Sector/Sphere [16] is a parallel runtime developed by Y. Gu, and R. L. Grossman that can be used to implement MapReduce style applications. Sphere uses Sector distributed files system resembling an architecture similar to Hadoop.

4.1.2 Parallel Topologies

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 [17] and SWARM [18] 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. DryadLINQ, Hadoop, and *Twister* can all handle this class of applications well. Except for the manual data partitioning requirement, programming such problems using DryadLINQ is considerably easier than Hadoop or *Twister* implementations. With the debugging support from visual studio and the automatic deployment mechanism, the users can develop applications faster

with DryadLINQ. The CAP3 program we developed using DryadLINQ can be used as a model for many similar problems which has the simple parallel topology of collection of independent tasks.

MapReduce programming model provides more parallel topologies than the simple independent tasks 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. In DryadLINQ we can use the programming flows of `Apply -> GroupBy -> Apply` or `Select -> GroupBy -> Apply` to simulate MapReduce style computations by using an appropriate `GroupBy` function.

Among other parallel runtimes that support individual tasks and MapReduce style applications, Sphere[16] 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 [19] 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 section 3.2. 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 in 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 [7] 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 to be more suitable for applications that process structured data that can be fit into tabular structures.

4.1.3 *Twister*: Iterate MapReduce

Our work on *Twister* (previously known as CGL-MapReduce) extends capabilities of the MapReduce programming to applications that perform iterative MapReduce computations. We differentiate the variable and fixed data items used in MapReduce computation and allow cacheable map/reduce tasks to hold static data in memory to support faster iterative MapReduce computations. The use of streaming for communication enables *Twister* to operate with minimum overheads. Currently *Twister* 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. The architecture of *Twister* and a comparison of synchronization and intercommunication mechanisms used by the parallel runtimes are shown in figure 12.

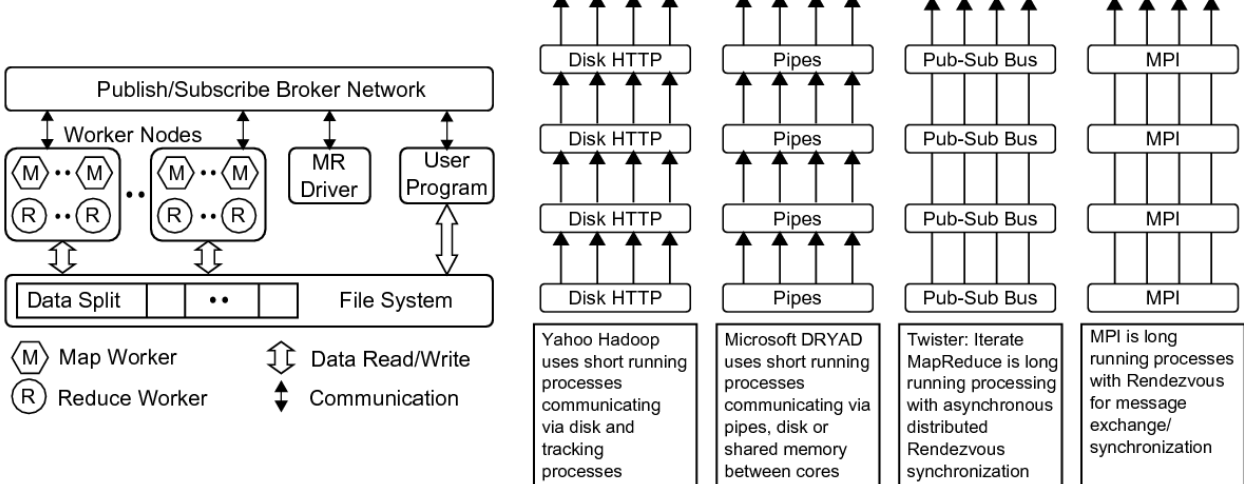

Figure 15. (Left) Components of the *Twister*. (Right) Different synchronization and intercommunication mechanisms used by the parallel runtimes.

4.2 Performance and Usability of Dryad

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 MapReduce jobs in HEP data analysis 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. 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 *Twister*.

Apart from those we would like to highlight the following usability characteristics of DryadLINQ comparing it with other similar runtimes.

4.2.1 Installation and Cluster Access

We note a technical issue we encountered using DryadLINQ within our Windows HPC environment. The HPC clusters at our institution are setup using a network configuration that has the headnode connected directly to the enterprise network (ADS domain access) and the compute nodes behind the headnode on a private network. Enterprise network access is provided to the compute nodes via DHCP and NAT(network address translation) services running on the headnode. This is our preferred configuration as it isolates the compute nodes from extraneous network traffic, places the compute nodes on a more secure private network and minimizes the attack surface of our clusters.

Using this configuration with DryadLINQ applications has been somewhat cumbersome as this configuration does not allow direct access to the compute node's private network from the enterprise network, while DryadLINQ applications require to access the compute nodes periodically. In other words, unless we run our DryadLINQ applications directly on the headnode, DryadLINQ is unable to access the compute node file systems as only the headnode is aware of the private network.

4.2.2 Developing and Deployment of Applications

Enabling DryadLINQ for an application simply requires adding DryadLINQ.dll to the project and pointing to the correct `DryadLinqConfig.xml`. After this step, the user can develop applications using Visual Studio and use it to deploy and run DryadLINQ applications directly on the cluster. With the appropriate cluster configurations, the development teams can test DryadLINQ applications directly from their workstations. In Hadoop, the user can add Hadoop jar files to the class path and start developing Hadoop applications using a Java development environment, but to deploy and run those applications the user need to create jar files packaging all the necessary programs and then copy them to a particular directory that Hadoop can find. Tools such as IBM's eclipse plugin for MapReduce [24] add more flexibility to create MapReduce computations using Hadoop.

4.2.3 Debugging

DryadLINQ supports debugging applications via visual studio by setting the property DryadLinq.LocalDebug=`true`. This is a significant improvement of usability compared to the other parallel runtimes such as Hadoop. The user can simply develop the entire application logic in his workstation and move to the cluster to do the actual data processing. Hadoop also supports single

machine deployments but the user needs to do manual configuration and debugging to test applications.

4.2.4 Fault Tolerance

The Dryad publication [4] discusses the fault tolerance features such as re-execution of failed vertices and duplicate execution of slower running tasks. We expected good fault tolerance support from Dryad, since better fault tolerance support is noted as major advantage in the new parallel frameworks like Dryad and Hadoop map reduce over the traditional parallel frameworks, enabling them to perform reliable computations on commodity unreliable hardware.

On the contrary, recently we encountered couple of issues regarding Dryad fault tolerance with respect to duplicate executions and failed vertices. First issue is the failures related to duplicate task executions. We wanted to perform a larger computation on a fewer number of nodes for scalability testing purposes. Due to unbalanced task sizes and the longer running times of vertices, Dryad executed duplicate tasks for the slower running tasks. Eventually the original tasks succeeded and the duplicated tasks got killed. But upon seen the killed tasks, the window HPC scheduler terminated the job as a failure. In this case we assume that Dryad behaved as expected by scheduling the duplicate tasks, but the Dryad windows HPC scheduler integration caused the failure without understanding the Dryad semantics. We've been informed that this issue is fixed in the Nov 2009 release of DryadLINQ, where DryadLINQ application will terminate correctly with the correct output, even though the HPC job is marked as failed.

Second issue happened recently when a misbehaving node joined the windows HPC cluster unexpectedly. A task from a Dryad job got scheduled in this node and that particular task failed due to the misbehavior of the node. We expected Dryad to schedule the failed task on a different node and to recover the job, but instead the whole job got terminated as a failed job. We have encountered both of the above issues in our Hadoop clusters many times and Hadoop was able to recover all of them successfully.

4.2.5 Monitoring

DryadLINQ depends on the HCP Cluster Manager and HPC Job Manager's monitoring capabilities to monitor the progress and problems of the jobs. Although the HPC Cluster Manager and Job Manager give better view of the hardware utilization and locations where the job getting executed, there is no direct way to find the progress of the DryadLINQ applications. Finding an error that happens only in a cluster deployment is even harder with the current release of DryadLINQ. For example, the user need to follow the steps below to find the standard output (stdout) and standard error (stderr) streams related to a particular vertex of the DryadLINQ application.

1. Find the job's ID using Job Manager
2. Find which vertex (sub job has failed) and find its task number
3. Find where that task was running using Job Manager
4. Navigate to the shared directory where the job outputs are created
5. Open the stdout and stderr files to find any problems.

Note: When the vertex is using an `Apply` operation won't give any information because then the standard outputs printed by the program does not get saved in stdout or stderr files.

Hadoop provides a simple web interface to monitor the progress of the computations and to locate these standard output and error files. A simple view of how many map/reduce tasks

completed so far gives a better understanding of the progress of the program in Hadoop. We think that a simple approach like this would help new users to develop applications easily without frustration using DryadLINQ.

5. Summary of key features of applications that suitable and not suitable for Dryad

In the past Fox has discussed the mapping applications to different hardware and software in terms of 5 "Application Architectures" [22]. These 5 categories are listed in Table 6.

Table 6. Application classification

1	Synchronous	The problem can be implemented with instruction level Lockstep Operation as in SIMD architectures
2	Loosely Synchronous	These problems exhibit iterative Compute-Communication stages with independent compute (map) operations for each CPU that are synchronized with a communication step. This problem class covers many successful MPI applications including partial differential equation solution and particle dynamics applications.
3	Asynchronous	Compute Chess and Integer Programming; Combinatorial Search often supported by dynamic threads. This is rarely important in scientific computing but at heart of operating systems and concurrency in consumer applications such as Microsoft Word.
4	Pleasingly Parallel	Each component is independent. In 1988, Fox estimated this at 20% of the total number of applications but that percentage has grown with the use of Grids and data analysis applications as seen here and for example in the LHC analysis for particle physics [23].
5	Metaproblems	These are coarse grain (asynchronous or dataflow) combinations of classes 1)-4). This area has also grown in importance and is well supported by Grids and described by workflow.
6	*Twister*	It describes file(database) to file(database) operations which has three subcategories given below and in table 7. 6a) Pleasingly Parallel Map Only 6b) Map followed by reductions 6c) Iterative "Map followed by reductions" – Extension of Current Technologies that supports much linear algebra and data mining

The above classification 1 to 5 largely described simulations and was not aimed directly at data processing. Now we can use the introduction of MapReduce as a new class which subsumes aspects of classes 2, 4, 5 above. We generalize MapReduce to include iterative computations and term it *Twister*. We have developed a prototype of this extended model and term it currently *Twister* [9][10]. Then this new category is summarized as:

Note overheads in categories 1, 2, 6c go like Communication Time/Calculation Time and basic MapReduce pays file read/write costs while MPI overhead is measured in microseconds. In *Twister* we use data streaming to reduce overheads while retaining the flexibility and fault-tolerance of MapReduce. *Twister* supports the Broadcast and Reduce operations in MPI which are all that is needed for much linear algebra and datamining including the clustering and MDS approaches described earlier.

Table 7. Comparison of *Twister* subcategories and Loosely Synchronous category

Map-only	**Classic Map-reduce**	**Iterative Reductions *Twister***	**Loosely Synchronous**
input map() output	input map() reduce() output	input map() reduce() output	j B i A P$_i$

– Document conversion (PDF->HTML)
– Brute force searches in cryptography
– Parametric sweeps
– CAP3 Gene assembly
– PolarGrid Matlab data analysis
	– High Energy Physics (HEP) Histograms
	– Distributed search
	– Distributed sort
	– Information retrieval
	– Calculation of Pairwise Distances for ALU sequences
		– Expectation maximization algorithms
		– Linear Algebra
		– Datamining including
		– Clustering
		  – K-means
		  – Deterministic Annealing clustering
		– Multidimensional Scaling (MDS)
			– Many MPI scientific applications utilizing wide variety of communication constructs including local interactions
			– Solving differential equations and
			– Particle dynamics with short range forces

← **Domain of MapReduce and Iterative Extensions** →			**MPI**

From the applications we developed it is trivial that the DryadLINQ can be applied to real scientific analyses. DryadLINQ performs competitively well with Hadoop for both pleasingly parallel and MapReduce style applications. However, applicability of DryadLINQ (also Hadoop) for iterative MapReduce applications is questionable. The file based communication mechanism and loading of static data again and again causes higher overheads in this class of applications. However, we expect that these overheads may reduce if DryadLINQ support in memory communication mechanism such as TCP pipes.

Additional support for partitioning data (few tools to perform various data partitioning strategies) and a mechanism to monitoring the progress of applications are two areas that DryadLINQ needs improvements.

References

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

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

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

[4] 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.

[5] 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.

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

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

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

[9] J. Ekanayake, S. Pallickara, and G. Fox, “MapReduce for Data Intensive Scientific Analysis,” Fourth IEEE International Conference on eScience, 2008, pp.277-284.

[10] Geoffrey Fox, Seung-Hee Bae, Jaliya Ekanayake, Xiaohong Qiu, and Huapeng Yuan: Parallel Data Mining from Multicore to Cloudy Grids. Proceedings of HPC 2008 High Performance Computing and Grids workshop Cetraro Italy July 3 2008

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

[12] J. B. Kruskal and M.Wish. *Multidimensional Scaling*. Sage Publications Inc., Beverly Hills, CA, U.S.A., 1978.

[13] Fox, G. C., Hey, A. and Otto, S., Matrix Algorithms on the Hypercube I: Matrix Multiplication, *Parallel Computing*, 4, 17 (1987)

[14] Johnsson, S. L., T. Harris, et al. 1989. Matrix multiplication on the connection machine. Proc of the *1989 ACM/IEEE conference on Supercomputing*. Reno, Nevada, United States, ACM.

[15] Mountable HDFS, http://wiki.apache.org/hadoop/MountableHDFS

[16] 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.

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

[18] 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

[19] 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.

[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] Geoffrey C. Fox, Roy D. Williams, Paul C. Messina, *Parallel Computing Works!* Morgan Kaufmann (1994).

[23] Enabling Grids for E-science (EGEE): http://www.eu-egee.org/

[24] IBM Eclipse plugin for MapReduce, http://www.alphaworks.ibm.com/tech/mapreducetools

[25] J. Ekanayake, A. S. Balkir, T. Gunarathne, G. Fox, C. Poulain, N. Araujo, R. Barga. "DryadLINQ for Scientific Analyses", Technical report, Accepted for publication in eScience 2009

[26] M.A. Batzer, P.L. Deininger, 2002. "Alu Repeats And Human Genomic Diversity." Nature Reviews Genetics 3, no. 5: 370-379. 2002

[27] A. F. A. Smit, R. Hubley, P. Green, 2004. Repeatmasker. http://www.repeatmasker.org

[28] J. Jurka, 2000. Repbase Update: a database and an electronic journal of repetitive elements. Trends Genet. 9:418-420 (2000).

[29] Source Code. Smith Waterman Software. http://jaligner.sourceforge.net

[30] T.F. Smith, M.S.Waterman,. Identification of common molecular subsequences. Journal of Molecular Biology 147:195-197, 1981.

[31] O. Gotoh, An improved algorithm for matching biological sequences. Journal of Molecular Biology 162:705-708 1982.

[32] M, A. M. Altfeld (2003). Influence of HLA-B57 on clinical presentation and viral control during acute HIV-1 infection. AIDS .

[33] J.M. Carlson, (2008). Phylogenetic Dependency Networks: Inferring Patterns of CTL Escape and Codon Covariation in HIV-1 Gag. PLoS Comput Biol .

[34] PhyloD, http://research.microsoft.com/en-us/um/redmond/projects/MSCompBio/.

[35] J. Ekanayake, T. Gunarathne, J. Qiu, Cloud Technologies for Bioinformatics Applications, Technical report. January 4, 2010. (submitted to the Journal of IEEE Transactions on Parallel and Distributed Systems)

[36] PhyloDView,http://research.microsoft.com/en-us/um/redmond/projects/MSCompBio/PhyloDViewer/

Appendix A

```
//
//Compute intensive function described in section 3.1.3
//
public static int Func_ComputeIntensive(int index) {
	double val = 0;
	for (int i = 0; i < mat_size; i++)
	{
		for (int j = 0; j < mat_size; j++)
		{
			for (int k = 0; k < mat_size; k++)
			{
				val = pi * pi;
			}
		}
	}
	return index;
}
```

Appendix B

```
//
//Memory intensive function described in section 3.1.3
//

public static int ExecuteHighMemory(int index)
        {
            Random rand = new Random();
            double val = 0;

            for (int i = 0; i < num_repititions; i++)
            {
                double[] data1 = new double[array_size];
                for (int j = 0; j < array_size; j++)
                {
                    data1[j] = pi * rand.Next();
                }

                double[] data2 = new double[array_size];
                for (int j = 0; j < array_size; j++)
                {
                    data2[j] = pi * rand.Next();
                }

                for (int j = 0; j < num_compute_loops; j++)
                {
                    val = data1[rand.Next(array_size)] *
data2[rand.Next(array_size)];
                }
            }
            return index;
        }
```