DRYADLINQ CTP EVALUATION

Performance of Key Features and Interfaces in DryadLINQ CTP

Hui Li, Yang Ruan, Yuduo Zhou, Judy Qiu
November 15, 2011

SALSA Group, Pervasive Technology Institute, Indiana University
http://salsahpc.indiana.edu/

Table of Contents

1 Introduction

We are in the data deluge when progress in science requires the processing of large amounts of scientific data [1]. One important approach is to apply new languages and runtimes to new data-intensive applications [2] to enable the preservation, movement, access, and analysis of massive data sets. Systems such as MapReduce and Hadoop allow developers to write applications for distributing tasks to remote environments containing the desired data, which instantiates the paradigm of "moving the computation to data". The MapReduce programming model has been applied to a wide range of "big data" applications and attracts enthusiasm from distributed computing communities due to its ease of use and efficiency in processing large-scale distributed data.

MapReduce, however, has its limitations. For instance, its rigid and flat data-processing paradigm does not directly support relational operations that have multiple related inhomogeneous data sets. This causes difficulties and inefficiency when using MapReduce to simulate relational operations such as *join*, which is very common in database systems. For example, the classic implementation of PageRank is notably inefficient since the simulation of *joins* with MapReduce causes a lot of network traffic during the computation. Further optimization of PageRank requires developers to have sophisticated knowledge of web graph structure.

Dryad [3] is a general-purpose runtime for supporting data-intensive applications on a Windows platform. It models programs as a directed, acyclic graph of the data flowing between operations and addresses some limitations existing in MapReduce. DryadLINQ [4] is the declarative programming interface for Dryad, and it automatically translates LINQ programs written by the .NET language into distributed computations executing on top of the Dryad system. For some applications, writing DryadLINQ distributed programs is as simple as writing sequential programs. DryadLINQ and Dryad runtime optimize job execution planning. This optimization is handled by the runtime and is transparent to users. For example, when implementing PageRank with the *GroupAndAggregate*() operator, DryadLINQ can dynamically constructs a partial aggregation tree based on data locality to reduce network traffic over cluster nodes.

The overall performance issues of data parallel programing models like MapReduce are well understood. DryadLINQ simplifies usage by leaving the details of scheduling, communication, and data access to underlying runtime systems that hide the low-level complexity of parallel programming. However, such an abstraction may come at a price in terms of performance when applied to a wide range of applications that port to multi-core and heterogeneous systems. We have conducted extensive experiments on DryadLINQ/DryadLINQ CTP and its usage in a recent publication [5] to identify the classes of applications that fit well. It is based on our evaluation of DryadLINQ, which was published as a Community Technology Preview (CTP) in December 2010.

Let us explain how this new report fits with earlier results. This report extends significantly the results presented in our earlier DryadLINQ evaluation [2] and we have not repeated discussions given earlier. The first report in particular focused on comparing DryadLINQ with Hadoop and MPI and covered multiple pleasing parallel (essentially independent) applications. Further it covered K-means clustering as an example of an important iterative algorithm and used this to motivate the Iterative MapReduce runtime. The original report had an analysis of applications suitable for MapReduce and its iterative extensions which is still accurate but not repeated here.

In this report we use a newer version of DryadLINQ (CTP) programming models and can be applied to three different types of classic scientific applications including pleasingly parallel, hybrid distributed and shared memory, and distributed grouped aggregation. We further give a comparative analysis of DryadLINQ CTP and Hadoop. Our focus was on novel features of this runtime and particularly challenging

applications. We cover an essentially pleasing parallel application consisting of Map and Reduce steps, the Smith Waterman Gotoh (SWG) [6] algorithm for dissimilarity computation in bioinformatics. In this case, we study in detail load balancing with inhomogeneity in cluster and application characteristics. We implement SWG with *ApplyPerPartition* operator, which can be considered as a distributed version of "Apply" in SQL. We cover the use of hybrid programming to combine inter-node distributed memory with intra-node shared memory parallelization, using multicore threading and DryadLINQ for the case of matrix multiplication which was covered briefly in the first report. We port multicore technologies including PLINQ and TPL into a user-defined function within DryadLINQ queries. Our new discussion is much more comprehensive than the first paper [5] and has an extensive discussion of the performance of different parallel algorithms on different programming models for threads. The other major application we look at is Pagerank which is sparse matrix vector multiplication, implemented as an iterative algorithm using power method. Here we compare several of the sophisticated LINQ models for data access. PageRank is a communication-intensive application that requires joining two input data streams and performing the grouped aggregation over partial results. We implemented the PageRank application with three distributed grouped aggregation approaches. The new paper has comments on usability and use of DryadLINQ in education, which were not in the original report [2].

Now we finish the introduction with the highlights of following sections as Table 1.

Table 1. Highlights of the DryadLINQ CTP Evaluation

	Key Features	Applications	Selected Findings
1	Task scheduling	Smith-Waterman Gotoh (SWG)	Compared with DryadLINQ (2009,11), DryadLINQ CTP provides better task scheduling strategy, data model, and interface to solve the workload balance issue for pleasingly parallel applications. (Section 3.4)
2	Hybrid Parallel programming models	Matrix multiplication	Porting multi-core technologies like PLINQ and TPL to DryadLINQ tasks can increase system utilization. (Section 4.5)
3	Distributed grouped aggregation	PageRank	The choice of distributed grouped aggregation with DryadLINQ CTP has a substantial impact on the performance of data aggregation/reduction applications. (Section 5.4)

Additional observations of DryadLINQ CTP:

1) **We found a bug in *AsDistributed()* interface, namely a mismatch between partitions and compute nodes in the default setting of DryadLINQ CTP**. (section 3.2.1)
2) DryadLINQ supports iterative tasks by chaining the execution of LINQ queries. However, **for BSP-style applications that need to explicitly evaluate LINQ query in each synchronization step, DryadLINQ requires resubmission of a Dryad job to the HPC scheduler at each synchronization step**, which limits its overall performance. (section 5.3.3)
3) When Dryad tasks invoke a third party executable binary file as process, **Dryad process is not aware of the class path that the Dryad vertex maintains, and it throws out an error : "required file cannot be found."** (section 6.1)
4) When applying late evaluation in chained queries, **DryadLINQ only evaluates the iterations parameter at the last iteration and uses that value for further execution of all the queries including previous iterations**. This imposes an ambiguous variable scope issue. (section 6.2)

5) When using a two dimensional array, objects in matrix multiplication, and PageRank applications, **DryadLINQ program will throw out an error message when a Dryad task tries to access unserializedilized two dimensional array objects on remote compute nodes.** (section 6.3)
6) DryadLINQ CTP is able to tolerate up to 50% compute node failure. **The job manager node failure is a single point failure that has no fault tolerance support from DryadLINQ**. (section 6.4.1)
7) **It is critical to run multiple DryadLINQ jobs simultaneously on a HPC cluster**. However, this feature is not mentioned in either Programming or Guides. **Every Dryad job requires an extra node acting as a job manager causing low CPU usage on this particular node**. (section 7.2)

Hadoop is a popular open source implementation of the Google's MapReduce model for Big Data applications. For instance, Hadoop is used by Yahoo to process hundreds of terabytes of data on at least 10,000 cores. Facebook run Hadoop jobs on 15 terabytes of new data per day. LINQ programing model is more general than Hadoop MapReduce for most applications that process semi-structure or un-structure data applications. For .NET platform, LINQ should be the best choice to express flow of data processing. However, **DryadLINQ has a jump start cost which is higher than that of Hadoop mainly due to the fact that many users are familiar with Hadoop and Linux environment**.

We analyze the key features of DryadLINQ CTP and Hadoop, especially those in which DryadLINQ outperforms Hadoop in programming interface, task scheduling, performance, and applications. We also studied performance-related issues between DryadLINQ CTP and Hadoop by identifying fast communication, data-locality-aware scheduling, and pipelining between jobs. In summary, we make the following observations:

8) **DryadLINQ provides a data model and better language support by interfacing with .NET and LINQ** that is more attractive than Hadoop's interface for some applications. (section 8.1)
9) **DryadLINQ performs better than Pig** when processing relational queries and Iterative MapReduce tasks; Note that DAG model in Dryad can be implemented as Workflow and **Workflow plus Hadoop is an interesting approach**. (section 8.1)
10) **DryadLINQ supports advanced inter-task communication technologies** such as files, TCP Pipe, and shared-memory FIFO. Hadoop transfers intermediate data via files and http. (section 8.1)
11) **DryadLINQ can maintain data locality at both Map and Reduce phases** while Hadoop only supports Map-side data locality. (section 8.1)
12) DryadLINQ supports pipelining execution stages for high performance but Hadoop doesn't. (section 8.1)
13) **DryadLINQ provides a rich set of distributed group aggregation strategies to reduce data movement,** but Hadoop has limited support. (section 8.1)
14) Although DAG supports dataflow though vertices, iterations defined via graphs is more like workflow. **DAG is not a suitable parallel runtime model for scientific computation where there're fine grained synchronizations between iterations** that is directly supported by Iterative MapReduce. (section 8.2)
15) **Neither Hadoop nor DryadLINQ support Iterative MapReduce properly**. (section 8.2)

The report is organized as follows. Section 1 introduces key features of DryadLINQ CTP. Section 2 studies the task scheduling in DryadLINQ CTP with a SWG application. Section 3 explores hybrid parallel programing models with Matrix Multiplication. Section 4 introduces distributed grouped aggregation exemplified by PageRank. Section 5 investigates the programming issues of DryadLINQ CTP. Section 6 discusses programming issues in DryadLINQ CTP. Section 7 illustrates how Dryad/DryadLINQ has been used in class projects for computer science graduate students of Professor Qiu's courses at Indiana University. Section 8 analyzes the key features that DryadLINQ CTP outperformances Hadoop.

Note that in the report: “DryadLINQ CTP” refers to the DryadLINQ community technical preview released in 2010.12; "LINQ to HPC" refers to the newly released LINQ to HPC Beta 2 in 2011.07; “Dryad/DryadLINQ (2009)” refers to the version released in 2009.11.11; “Dryad/DryadLINQ” refers to all Dryad/DryadLINQ versions. Experiments are conducted on four Windows HPC clusters: STORM, TEMPEST, MADRID and iDataPlex [Appendix A, B, C and D]. STORM consists of heterogeneous multicore nodes while TEMPEST, MADRID and iDataPlex are homogeneous production systems of 768,128 and 256 cores each.

2 Overview

Dryad, DryadLINQ, and the Distributed Storage Catalog (DSC) [7] are sets of technologies that support the processing of data-intensive applications on a Windows HPC cluster. The software stack of these technologies is shown in Figure 1. Dryad is a general-purpose distributed runtime designed to execute data-intensive applications on a Windows cluster. A Dryad job is represented as a directed acyclic graph (DAG), which is called a “Dryad” graph. The Dryad graph consists of vertices and channels. A vertex in the graph represents an independent instance of the data processing for a particular stage. Graph edges represent channels transferring data between vertices. A DSC component works with the NTFS to provide the data management functionalities, such as file replication and load balancing for Dryad and DryadLINQ.

DryadLINQ is a library for translating .NET written Language-Integrated Query (LINQ) programs into distributed computations executing on top of the Dryad system. The DryadLINQ API takes advantage of standard query operators and adds query extensions specific to Dryad. Developers can apply LINQ operators such as *join* and *groupby* to a set of .NET objects. Specifically, DryadLINQ supports a unified data and programming model in the representation and processing of data. DryadLINQ data objects are collections of.NET type objects, which can be split into partitions and distributed across the computer nodes of a cluster. These DryadLINQ data objects are represented as either DistributedQuery <T> or DistributedData <T> objects and can be used by LINQ operators. In summary, DryadLINQ greatly simplifies the development of data parallel applications.

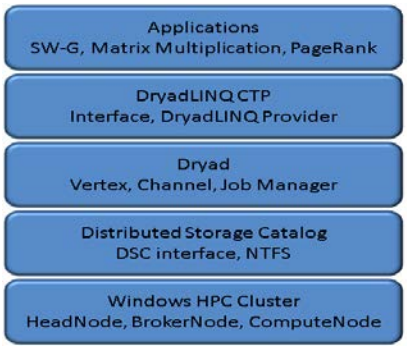


Figure1: Software Stack for DryadLINQ CTP

2.1 Task Scheduling

Task scheduling for DryadLINQ CTP is a key feature investigated in this report. A DryadLINQ provider translates LINQ queries into distributed computation and automatically dispatches tasks to a cluster. This process is handled by the runtime and is transparent to users. The task scheduling component also automatically handles fault tolerance and workload balance issues.

We have studied DryadLINQ CTP’s load balance issue and investigated its relationship to task granularity along with its impact on performance. In batch job scheduling systems, like PBS, programmers manually

group/ungroup (or partition/combine) input and output data for the purpose of controlling task granularity. Hadoop provides a user interface to define task granularity as the size of input records in HDFS. Similarly, DryadLINQ (2009) allows developers to create a partition file. DryadLINQ CTP has a simplified data model and flexible interface in which *AsDistributed*, *Select*, and *ApplyPerPartition* operators (which can be considered as the distributed versions of *Select* and *Apply* in SQL) enable developers to tune the granularity of data partitions and run pleasingly parallel applications like sequential ones.

2.2 Parallel Programming Model

Dryad is designed to process coarse granularity tasks for large-scale distributed data and schedules tasks to computing resources over compute nodes rather than cores. To achieve high utilization of the multi-core resources of a HPC cluster for DryadLINQ jobs, one approach is to explore inner-node parallelism using PLINQ since DryadLINQ can automatically transfer a PLINQ query to parallel computations. Another approach is to apply multi-core technologies in .NET, such as Task Parallel Library (TPL) or thread pool for user-defined functions within the lambda expression of DryadLINQ query.

In a hybrid parallel programming model, Dryad handles inter-node parallelism while PLINQ, TPL, and thread pool technologies leverage inner-node parallelism on multi-cores. Dryad/DryadLINQ has been successful in executing as a hybrid model and applied to data clustering applications, such as General Topographical Mapping (GTM) interpolation and Multi-Dimensional Scaling (MDS) interpolation [8]. Most of the pleasingly parallel applications can be implemented in a straightforward fashion using this model with increased overall utilization of cluster resources. However, more compelling machine learning or data analysis applications usually have either squared or quadratic computation complexity, which has high requirements of system design for scalability.

2.3 Distributed Grouped Aggregation

The *groupby* operator in parallel databases is often followed by aggregate functions, which groups input records into partitions by keys and merges the records for each group by certain attribute values; this computing pattern is called Distributed Grouped Aggregation. Example applications include sales data summarizations, log data analysis, and social network influence analysis.

MapReduce and SQL for databases are two programming models to perform distributed grouped aggregation. MapReduce has been applied to the process of a wide range of flat distributed data, but is inefficient in processing relational operations, which have multiple inhomogeneous input data stream such as *join*. However, a full-featured SQL database has extra overhead and constraints that prevent it from processing large-scale input data.

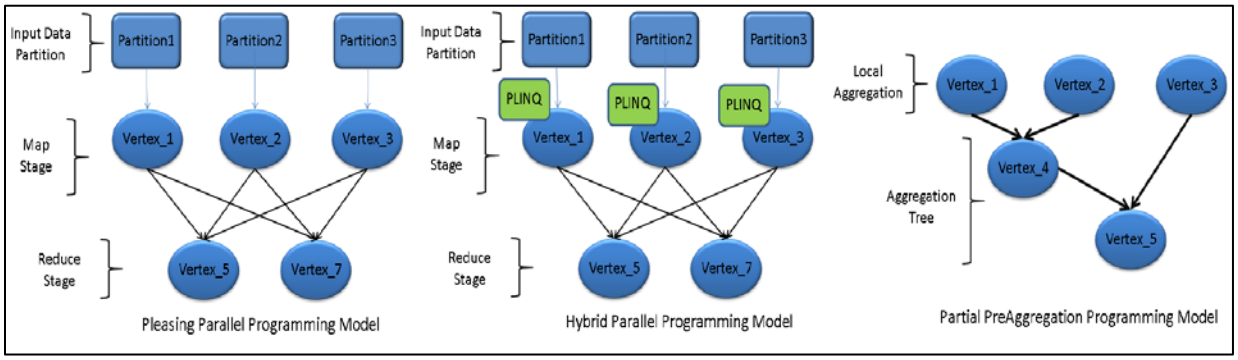


Figure 2: Three Programming Models for Scientific Applications in DryadLINQ CTP

DryadLINQ is between SQL and MapReduce and addresses some of their limitations. DryadLINQ provides SQL-like queries for processing efficient aggregation for homogenous input data streams and multiple inhomogeneous input streams and does not have sufficient overhead since SQL eliminates some of the functionalities of a database (transactions, data lockers, etc.). Further, DryadLINQ can build an aggregation tree (some databases also provides this kind of optimization) to decrease data transformation in the hash partitioning stage. In this report, we investigated the usability and performance of three programming models using Dyrad/DryadLINQ as illustrated in Figure 2: a) the pleasingly parallel mode, b) the hybrid programming model, and d) distributed grouped aggregation.

3 Pleasingly Parallel Application in DryadLINQ CTP

3.1 Introduction

A pleasingly parallel application can be partitioned into parallel tasks since there is neither essential data dependency nor communication between those parallel tasks. Task scheduling and granularity have a great impact on performance and are evaluated in DryadLINQ CTP using the Pairwise Alu Sequence Alignment application. Furthermore, many pleasingly parallel applications share a similar execution pattern. The observation and conclusion drawn from this work applies to a large class of similar applications.

3.1.1 Pairwise Alu Sequence Alignment Using Smith Waterman Gotoh

The Alu clustering problem [9] is one of the most challenging problems for sequencing clustering because Alus represent the largest repeat families in human genome. There are approximately 1 million copies of Alu sequences in the 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. Notably, Alu clustering can be viewed as a classic case study for the capacity of computational infrastructure 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 problems in bioinformatics for the next few years, e.g. the automated protein family classification for a few millions proteins predicted from large meta-genomics projects.

An open source version, NAligner [10], of the Smith Waterman-Gotoh algorithm (SWG) [11] was used to ensure low start-up effects by each task process for large numbers (more than a few hundred) at a time. The needed memory bandwidth is reduced by storing half of the data items for symmetric features.

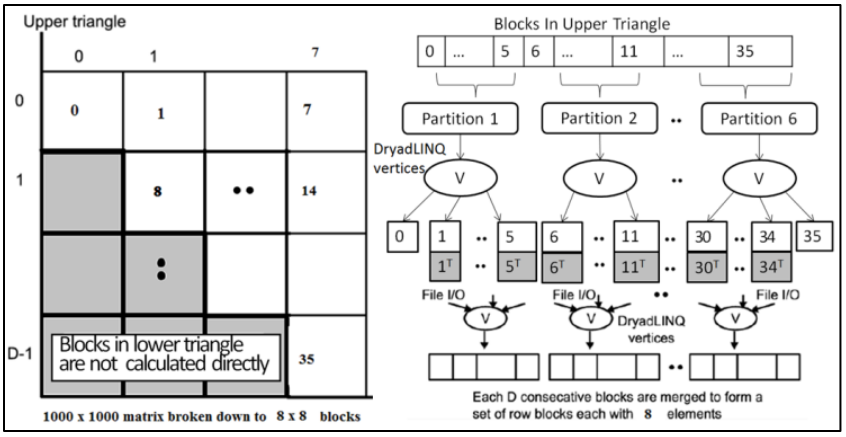


Figure 3: Task Decomposition (left) and the Dryad Vertex Hierarchy (right) of the DryadLINQ Implementation of SWG Pairwise Distance Calculation Application

3.1.2 DryadLINQ Implementation

The SWG program runs in two steps. In the map stage input data is divided into partitions being assigned to vertices. A vertex calls external pair-wise distance calculations on each block and runs independently. In the reduce stage, this vertex starts a few merge threads to collect output from the map stage, merges them into one file, and then sends meta data of the file back to the head node. To clarify our algorithm, let's consider an example of 10,000 gene sequences that produces a pairwise distance matrix of size 10,000 × 10,000. The computation is partitioned into 8 × 8 blocks as a resultant matrix D, where each sub-block contains 1250 × 1250 sequences. Due to the symmetry feature of pairwise distance matrix D(i, j) and D(j, i), only 36 blocks need to be calculated as shown in the upper triangle matrix of Figure 3 (left).

Dryad divides the total workload of 36 blocks into 6 partitions, each of which contains 6 blocks. After the partitions are distributed to available compute nodes an *ApplyPerPartition()* operation is executed on each vertex. A user-defined *PerformAlignments()* function processes multiple SWG blocks within a partition, where concurrent threads utilize multicore internal to a compute node. Each thread launches an operating system process to calculate a SWG block in order. Finally, a function calculates the transpose matrix corresponding to the lower triangle matrix and writes both matrices into two output files on local file system. The main program performs another *ApplyPerPartition()* operation to combine the metadata of files as shown in Figure 3. The pseudo code for our implementation is provided as below:

Map stage:

```
DistributedQuery<OutputInfo> outputInfo = swgInputBlocks.AsDistributedFromPartitions()
ApplyPerPartition(blocks => PerformAlignments(blocks, swgInputFile, swgSharePath,
outputFilePrefix, outFileExtension, seqAlignerExecName, swgExecName));
```

Reduce stage:

```
var finalOutputFiles = swgOutputFiles.AsDistributed().ApplyPerPartition(files =>
PerformMerge(files, dimOfBlockMatrix, sharepath, mergedFilePrefix, outFileExtension));
```

3.2 Task Granularity Study

This section examines the performance of different task granularities. As mentioned above, SWG is a pleasingly parallel application for dividing the input data into partitions. The task granularity was tuned by saving all SWG blocks into two-dimensional arrays and converting to distributed partitions using *AsDistributedFromPartitions* operator.

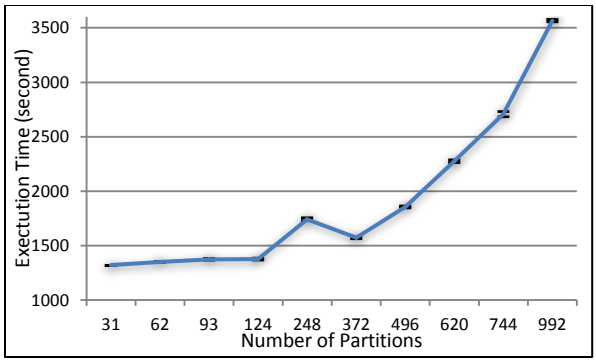


Figure 4: Execution Time for Various SWG Partitions
Executed on Tempest Cluster, with input of 10,000 sequences, and a 128×128 block matrix

The experiment was performed on a 768 core (32 nodes with 24 cores per node) Windows cluster called "TEMPEST" [Appendix B]. The input data of SWG has a length of 8192, which requires about 67 million distance calculations. The sub-block matrix size is set to 128 × 128 while we used *AsDistributedFromPartitions()* to divide input data into various partition sets {31, 62, 93, 124, 248, 372, 496, 620, 744, 992}. The mean sequence length of input data is 200 with a standard deviation as 10, which gives essentially homogeneous distribution ensuring a good load balance. On a cluster of 32 compute nodes, Dryad job manager takes one node for its dedicated usage and leaves 31 nodes for actual computations. As shown in Figure 4 and Table 1 (Appendix G), smaller number of partitions delivered better performance. Further, the best overall performance is achieved at the least scheduling cost derived from 31 partitions for this experiment. The job turnaround time increases as the number of partition increases for two reasons: 1) scheduling cost increases as the number of tasks increases, 2) partition granularity becomes finer with increasing number of partitions. When the number of partitions reaches over 372, each partition has less than 24 blocks making resources underutilized on a compute node of 24 cores. For pleasingly parallel applications, partition granularity and data homogeneity are major factors that impact performance.

3.2.1 Workload Balancing

The SWG application handled input data by gathering sequences into block partitions. Although gene sequences were evenly partitioned in sub-blocks, the length of each sequence may vary. This causes imbalanced workload distribution among computing tasks. Dryad (2009.11) used a static scheduling strategy binding its task execution plan with partition files, which gave poor performance for skewed/imbalanced input data [2]. We studied the scheduling issue in DryadLINQ CTP using the same application.

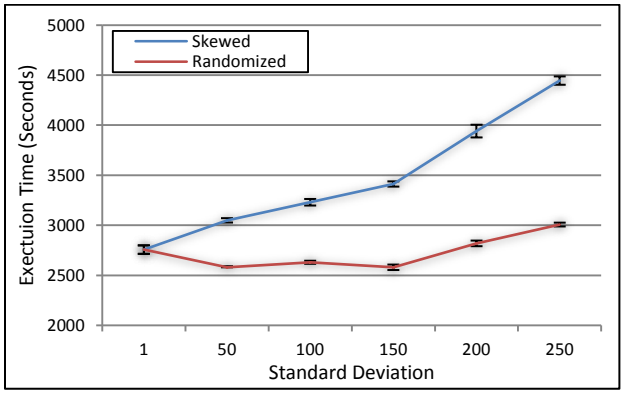


Figure 5: SWG Execution Time for Skewed and

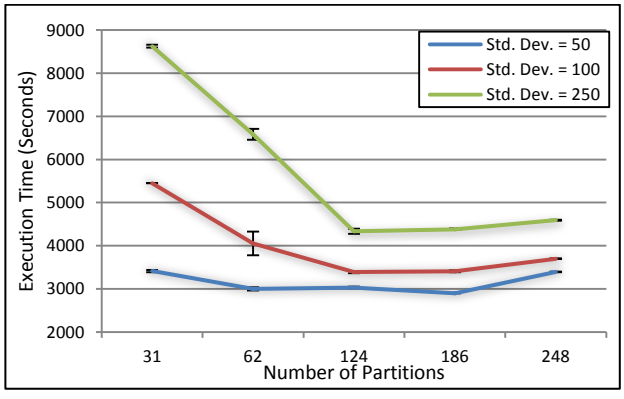


Figure 6: SWG Execution Time for Skewed Data with

Randomized Distributed Input Data Different Partition Amount

A set of SWG jobs was executed on the TEMPEST cluster with input size of 10000 sequences. The data were randomly generated with an average sequence length of 400, corresponding to a normal distribution with varied standard deviations. We constructed the SWG sub-blocks by randomly selecting sequences from the above data set in contrast to selecting sorted sequences based on their length. Figure 5 has line charts labeled with error bars, where randomized data shows better performance than skewed input data. Similar results were presented in the Dryad (2009) report as well. Since sequences were sorted by length for a skewed sample, computational workload in each sub-block was hugely variable, especially when the standard deviation was large. On the other hand, randomized sequences gave a balanced distribution of workload that contributed to better overall performance. DryadLINQ CTP provides an interface for developers to tune partition granularity. The load imbalance issue can be addressed by splitting the skewed distributed input data into many finer partitions. Figure 6 shows the relationship between number of partitions and performance. In particular, a parabolic chart suggests an initial overhead that drops as partitions and CPU utilization increase. Fine-grained partitions enable load balancing as SWG jobs start with sending small tasks to idle resources. Note that 124 partitions gives best performance in this experiment. With increasing partitions, the scheduling cost outweighs the gains of workload balancing. Figures 5 and 6 imply that the optimal number of partitions also depends on heterogeneity of input data.

DryadLINQ CTP divides input data into partitions by default with twice the number of compute nodes. It does not achieve good load balance for some applications, such as inhomogeneous SWG data. We have shown how to address the load imbalance issue. Firstly, the input data can be randomized and partitioned to increase load balance. However, it depends on the nature of randomness and good performance is not guaranteed. Secondly, a fine-grained partition can help tuning load balance among compute nodes. There's a trade off in drastically increasing partitions, as the scheduling cost becomes a dominant factor of performance.

We found a bug in *AsDistributed()* interface, namely a mismatch between partitions and compute nodes in the default setting of DryadLINQ CTP. Dryad provides two APIs to handle data partition, *AsDistributed()* and *AsDistributedFromPartitions()*. In our test on 8 nodes (1 head node and 7 compute nodes), Dryad chose one dedicated compute node for the graph manager which left only 6 nodes for computation. Since Dryad assigns each compute node 2 partitions, *AsDistributed()* divides data into 14 partitions disregarding the fact that the node for the graph manager does no computation. This causes 2 dangling partitions. In the following experiment, input data of 2000 sequences were partitioned into sub blocks of size 128×128 and 8 computing nodes were used from the TEMPEST cluster.

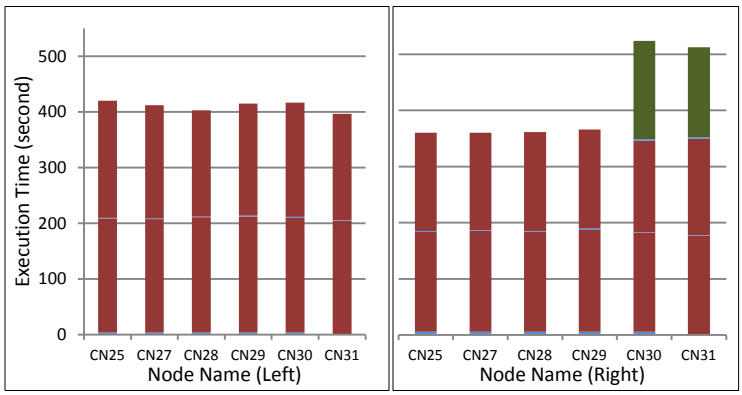


Figure 7: Mismatch between Partitions and Compute Nodes in Default Settings of DryadLINQ CTP

Figure 7 shows the execution time for 12 customized partitions on the left and the default partitions by *AsDistributed()* on the right. It is observed that input data are divided into 14 partitions over 6 compute nodes. The 2 dangling partitions colored in green slow down the whole calculation by almost 30%.

In summary, Dryad and Hadoop control task granularity by partitioning input data. DryadLINQ CTP has a default partition number twice that of the compute nodes. Hadoop partitions input data into chunks, each of which has a default size of 64MB. Hadoop implements a high-throughput model for dynamic scheduling and is insensitive to load imbalance issues. Dryad and Hadoop provide an interface allowing developers to tune partition and chunk granularity, with Dryad providing a simplified data model and interface on the .NET platform.

3.2.2 Scalability Study

Scalability is another key feature for parallel runtimes. The DryadLINQ CTP scalability test includes two sets of experiments conducted on the TEMPEST Cluster of 768 cores. A comparison of parallel efficiency for DryadLINQ CTP and DryadLINQ 2009 are discussed below.

The first experiment has an input size between 5,000 and 15,000 sequences with an average length of 2,500. The sub-block matrix size is 128 × 128 and there are 31 partitions, which is the optimal value found in previous experiments. Figure 8 shows performance results, where the red line represents execution time on 31 compute nodes, the green line represents execution time on a single compute node, and the blue line is parallel efficiency defined as the following:

$$\text{Parallel Efficiency} = \frac{\text{Execution Time on One Node}}{\text{Execution Time on Multinodes} \times \text{Number of Nodes}} \quad \text{(Eq. 1)}$$

Parallel efficiency is above 90% for most cases. An input size of 5000 sequences over a 32-node cluster shows a sign of underutilization for a slightly low start. When input data increases from 5000 to 15000, parallel efficiency jumps from 81.23% to 96.65%, as scheduling cost becomes less critical to the overall execution time as the input size increases.

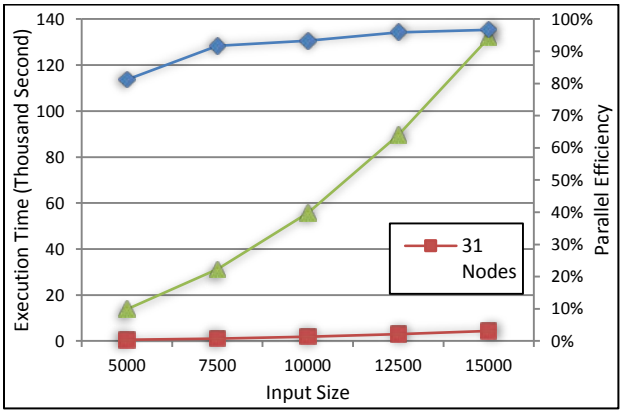


Figure 8: Performances and Parallel Efficiency on TEMPEST

The SWG jobs were also performed on 8 nodes of the MADRID cluster [Appendix C] using DryadLINQ 2009 and 8 nodes on the TEMPEST cluster [Appendix B] using DryadLINQ CTP. The input data is identical for both tests, which are 5,000 to 15,000 gene sequences partitioned into 128×128 sub blocks. Parallel efficiency (Eq. 1) is used as a metric for comparison. By computing 225 million pairwise distances both DryadLINQ CTP and DryadLINQ 2009 showed high utilization of CPUs with parallel efficiency of over 95% as displayed in Figure 9.

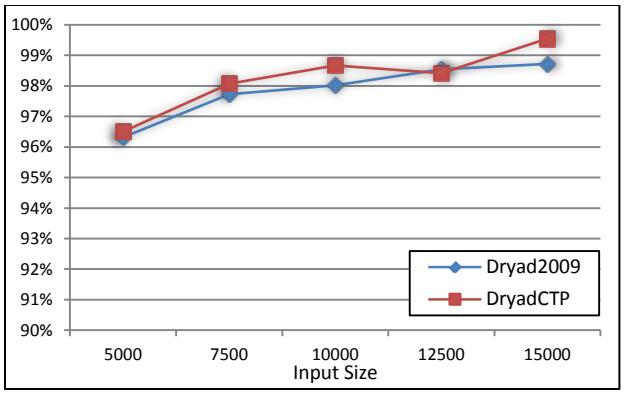


Figure 9: Parallel Efficiency on DryadLINQ CTP and DryadLINQ 2009

In the second set of experiments we calculated speed up to 10,000 input sequences (31 partitions with 128×128 sub block size) but varied the number of compute nodes in input sequence numbers 2, 4, 8, 16, and 31 (due to the cluster limitation of 31 compute nodes). The SWG application scaled up well on a 768-core HPC cluster. These results are presented in Table 4 of Appendix G. The execution time ranges between 40 minutes to 2 days. The speedup, as defined in equation 2, is almost linear with respect to the number of compute nodes as shown in Figure 10, which suggests that pleasingly parallel applications perform well on DryadLINQ CTP.

$$\text{Speedup} = \frac{\text{Execution time on one node}}{\text{Execution Time on multiple nodes}} \qquad \text{(Eq. 2)}$$

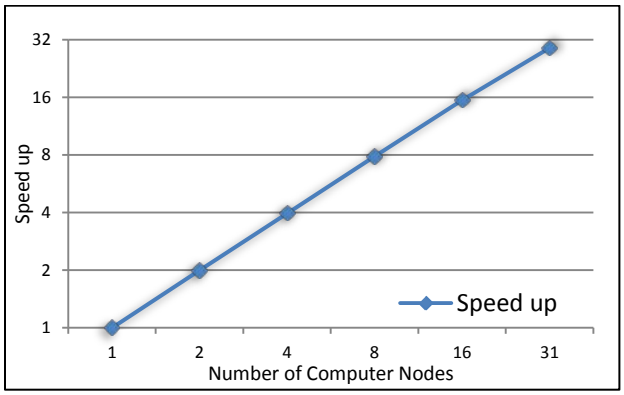


Figure 10: Speedup for SWG on Tempest with Varied Number of Compute Nodes

3.3 Scheduling on an Inhomogeneous Cluster

Adding a new hardware or integrating distributed hardware resources is common but may cause inhomogeneous issues for scheduling. In DryadLINQ 2009, the default execution plan is based on an assumption of a homogeneous computing environment. This motived us to investigate performance issues on an inhomogeneous cluster for DryadLINQ CTP. Task scheduling with attention to load balance is studied in this section.

3.3.1 Workload Balance with Different Partition Granularities

An optimal job-scheduling plan needs awareness of resource requirements and CPU time for each task, which is not practical in many applications. One approach is to split the input data set into small pieces and keep dispatching them to available resources.

This experiment was performed on STORM [Appendix A], an inhomogeneous HPC cluster. A set of SWG jobs is scheduled with different partition sizes, where input data contain 2048 sequences being divided into 64×64 sub blocks. These sub blocks are divided by *AsDistributedFromPartitions()* to form a set of partitions : {6, 12, 24, 48, 96, 192}. A smaller number of partitions implies a large number of sub blocks in each partition. As Dryad job manager keeps dispatching data to available nodes, the node with higher computation capability can process more SWG blocks. The distribution of partitions over compute nodes is shown in Table 5 of Appendix G; when the partition granularity is large, the distribution of SWG blocks among the nodes is proportional to the computational capacity of the nodes.

DryadLINQ CTP assigns a vertex to a compute node and each vertex contains one or more partitions. To study the relationship between partition granularity and load balance, the computation and scheduling time on 6 compute nodes for 3 sample SWG jobs were recorded separately. Results are presented in Figure 11 with compute nodes along the X-axis (e.g. cn01 ~ cn06) and elapsed time from the start of computation along the Y-axis. A red bar marks the time frame of a particular compute node doing computation, and a blue bar refers to the time frame for scheduling a new partition. Here are a few observations:

- When the number of partitions is small, workload is not well balanced, leading to significant variation in computation time on each node. Note that faster nodes stay idle and wait for slower ones to finish, as shown on the left graph in Figure 11.
- When the number of partitions is large, workload is distributed in proportion to the capacity of compute nodes. Too many small partitions cause high scheduling costs, thus slowing down overall computation, as illustrated on the right graph in Figure 11.
- Load balance favors a small number of partitions while scheduling costs favor a large number of jobs. An optimal performance is observed in the center graph in Figure 11.

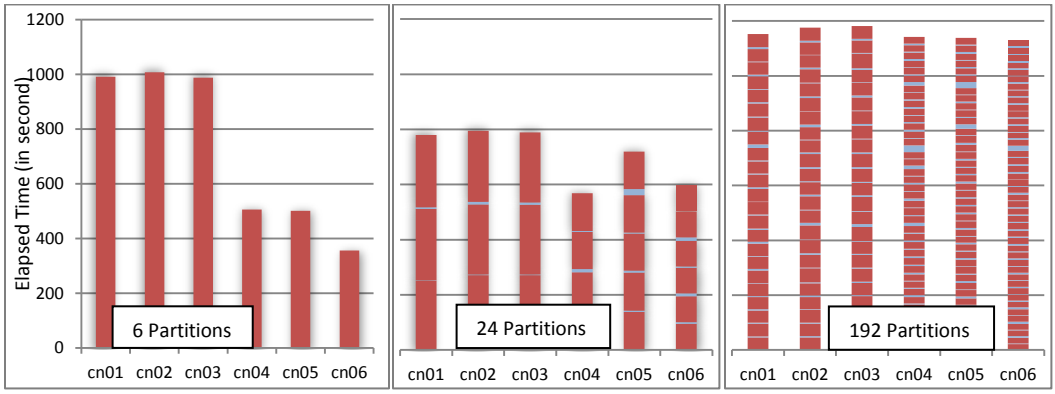


Figure 11: Scheduling Time vs. Computation Time of the SWG Application on DryadLINQ CTP

The optimal partition is a moderate number with respect to both load balance and scheduling cost. As shown in Figure 12 (middle), the optimal number of partitions is 24. Note that 24 partitions performed better than the default partition number, 14.

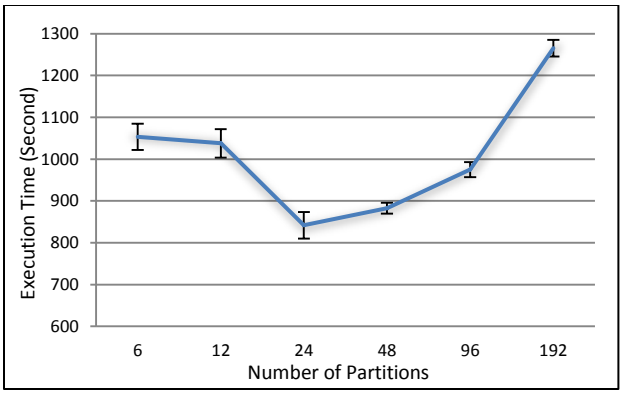


Figure 12: SWG Job Turnaround Time for Different Partition Granularities

Figure 13 shows that overall CPU usage drops as the number of partitions increases, due to increasing scheduling and data movement, which do not demand high CPU usage.

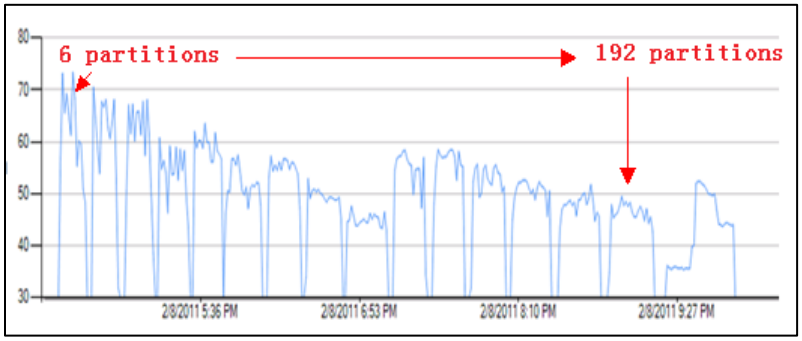


Figure 13: Cluster CPU Usage for Different SWG Partition Numbers

3.4 Evaluation and Findings

SWG is a pleasingly parallel application used to evaluate the performance of DryadLINQ CTP.

a) **The scalability test shows that if input data is homogeneous and the workload is balanced, then the optimal setting with low scheduling costs has the same number of partitions as compute nodes**.

b) In the partition granularity test where data is inhomogeneous and causes an imbalanced workload, the default DryadLINQ CTP setting of 62 partitions gave better results due to a finer balance between workload distribution and scheduling.

c) A comparison between DryadLINQ CTP and DryadLINQ 2009 shows that DryadLINQ CTP has achieved over 95% parallel efficiency in scale-up tests. Compared to the 2009 version DryadLINQ CTP also presents an improved load balance with a dynamic scheduling function.

d) **Our evaluation demonstrates that load balance, task granularity, and data homogeneity are major factors that impact the performance of pleasingly parallel applications using Dryad**.

e) Further, we found a bug involving mismatched partitions vs. compute nodes in the default setting of DryadLINQ CTP.

4 Hybrid Parallel Programming Model

4.1 Introduction

Matrix-matrix multiplication is a fundamental kernel [12], which can achieve high efficiency in both theory and practice. The computation can be partitioned into subtasks, which makes it an ideal candidate application in hybrid parallel programming studies using Dryad/DryadLINQ. However, there is not one optimal solution that fits all scenarios. *Different trade-offs of partition granularity largely correspond to computation and communication costs and are affected by memory/cache usage and network bandwidth/latency*. We investigated the performance of three matrix multiplication algorithms and three multi-core technologies in .NET, which run on both single and multiple cores of HPC clusters. The three matrix multiplication decomposition approaches are: 1) row decomposition, 2) row/column decomposition, and 3) block/block decomposition (Fox-Hey algorithm [13][14]). The multi-core technologies include: PLINQ, TPL [15], and Thread Pool, which correspond to three multithreaded programming models. In a hybrid parallel programming model, Dryad invokes inter-node parallelism while TPL, Threading, and PLINQ support inner-node parallelism. It is imperative to utilize new parallel programming paradigms that may potentially scale up to thousands or millions of multicore processors.

Matrix multiplication is defined as A * B = C (Eq. 3) where Matrix A and Matrix B are input matrices and Matrix C is the result matrix. The **p** in Equation 3 represents the number of columns in Matrix A and number of rows in Matrix B. Matrices in the experiments are square matrices with double precision elements.

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj} \qquad \text{(Eq. 3)}$$

4.2 Parallel Matrix-Matrix Multiplication Algorithms

4.2.1 Row-Partition Algorithm

The row-partition algorithm divides Matrix A into row blocks and distributes them onto compute nodes. Matrix B is copied to every compute node. Each Dryad task multiplies row blocks of Matrix A by all of matrix B and the main program aggregates a complete matrix C. The data flow of the Row Partition Algorithm is shown in Figure 14.

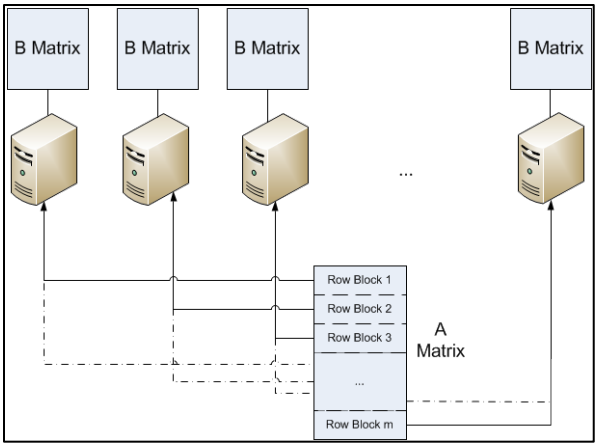


Figure 14: Row-Partition Algorithm

The blocks of Matrices A and B are first stored using *DistributedQuery* and *DistributedData* objects defined in DryadLINQ. Then an *ApplyPerPartition* operator invokes a user-defined function *rowsXcolumnsMultiCoreTech* to perform subtask computations. Compute nodes read file names for each input block and get the matrix remotely. As the row partition algorithm has a balanced distribution of workload over compute nodes, an ideal partition number equals the number of compute nodes. The pseudo code is in Appendix H, 1.

4.2.2 Row-Column Partition Algorithm

The Row-Column partition algorithm [16] divides Matrix A by rows and Matrix B by columns. The column blocks of Matrix B are distributed across the cluster in advance. The whole computation is divided into several iterations. In each iteration one row block of Matrix A is broadcast to all compute nodes and multiplies by the one-column blocks of Matrix B. The output of each compute node is sent to the main program to form a row block of Matrix C. The main program then collects the results of multiple iterations to generate the complete output of Matrix C.

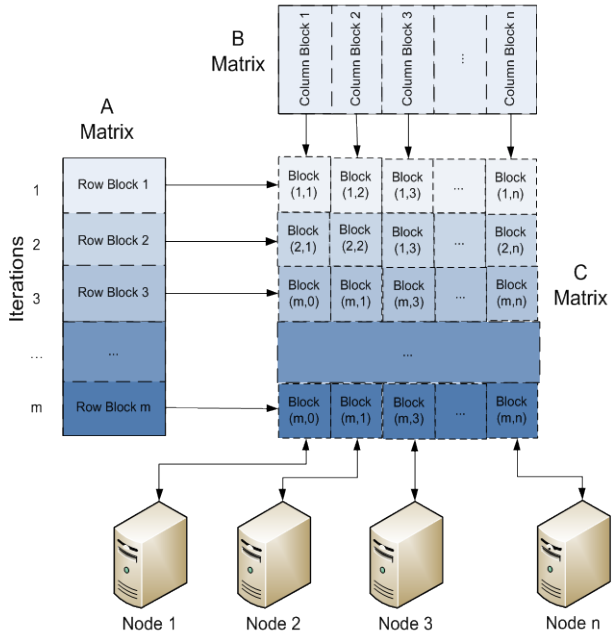


Figure 15: Row-Column Partition Algorithm

The column blocks of Matrix B are distributed by the *AsDistributed()* operator across the compute nodes. In each iteration an *ApplyPerPartition* operator invokes a user-defined function *aPartitionMultiplybPartition* to multiply one column block of Matrix B by one row block of Matrix A. The pseudo code is provided in Appendix H.2):

4.2.3 Block-Block Decomposition in the Fox-Hey Algorithm

The block-block decomposition in the Fox-Hey algorithm divides Matrix A and Matrix B into squared sub-blocks. These sub-blocks are dispatched to a virtual topology on a grid with the same dimensions for the simplest case. For example, to run the algorithm on a 2X2 processes mesh, Matrices A and B are split along both rows and columns to construct a matching 2X2 block data mesh. In each step of computation, every process holds a current block of Matrix A by broadcasting and a current block of Matrix B by shifting upwards and then computing a block of Matrix C. The algorithm is as follows:

For k = 0: s-1

1) The process in row I with A(I, (i+k)mod s) broadcasts it to all other processes I the same row i.
2) Processes in row I receive A(I, (i+k) mod s) in local array T.
3) For I =0;s-1 and j=0:s-1 in parallel
   C(I,j) = c(I,j)+T*B(I,j)
   End
4) Upward circular shift each column of B by 1:
   B(I,j) ← B((i+1)mod s, j)

End

Figure 16 shows the case where Matrices A and B are both divided into a block mesh of 2x2. Correspondingly, 4 compute nodes are divided into a grid labeled C(0,0), C(0,1), C(1,0), C(1,1). In step 0, Matrix A broadcasts the active blocks in column 0 to compute nodes in the same row of the virtual grid of compute nodes (or processes), i.e. A(0,0) to C(0,0), C(0,1) and A(1,0) to C(1,0), C(1,1). The blocks in Matrix B will be scattered onto each compute node. The algorithm computes C_{ij} = AB on each compute node. In Step 1, Matrix A will broadcast the blocks in column 1 to the compute nodes, i.e. A(0,1) to C(0,0), C(0,1) and A(1,1) to C(1,0), C(1,1). Matrix B distributes each block to its target compute node and performs an upward circular shift along each column, i.e. B(0,0) to C(1,0), B(0,1) to C(1,1), B(1,0) to C(0,0), B(1,1) to C(0,1). Then a summation operation on the results of each iteration forms the final result in C_{ij} += AB.

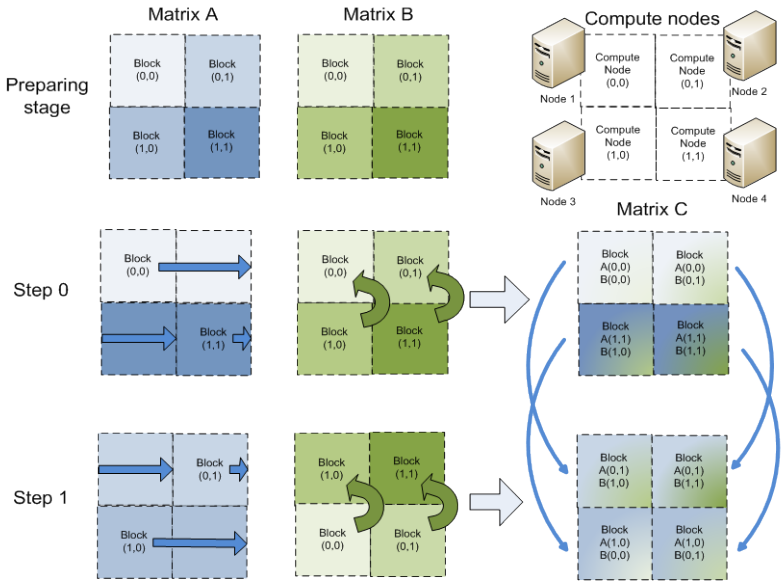


Figure 16: Different Stages of the Fox-Hey Algorithm in 2x2 Block Decompositions

Figure 17 illustrates a one-to-one mapping scenario for Dryad implementation where each compute node has one sub-block of Matrix A and one sub-block of Matrix B. In each step, the sub-blocks of Matrix A and Matrix B will be distributed onto compute nodes. Namely, the Fox-Hey algorithm achieves much better memory usage compared to the Row Partition algorithm which requires one row block of Matrix A and all of Matrix B for multiplication. In our future work, the Fox-Hey algorithm will be implemented in a general and powerful mapping schema to maximize cache usage, where the relationship between sub-blocks and the virtual grid of compute nodes will be many to one.

The pseudo code of basic Fox-Hey algorithm is given in Appendix H.3.1).

The Fox-Hey algorithm was originally implemented with MPI [17], which maintained intermediate status and data in processes during parallel computation. However, Dryad uses a data-flow runtime that does not support intermediate status of tasks during computation. To work around this, new values are assigned to *DistributedQuery<T>* objects by an updating operation as shown in the following pseudo code in Appendix H.3.2).

The sequential code of Matrix Multiplication is given in Appendix H. In addition, we provide three implementations to illustrate three multithreaded programming models.

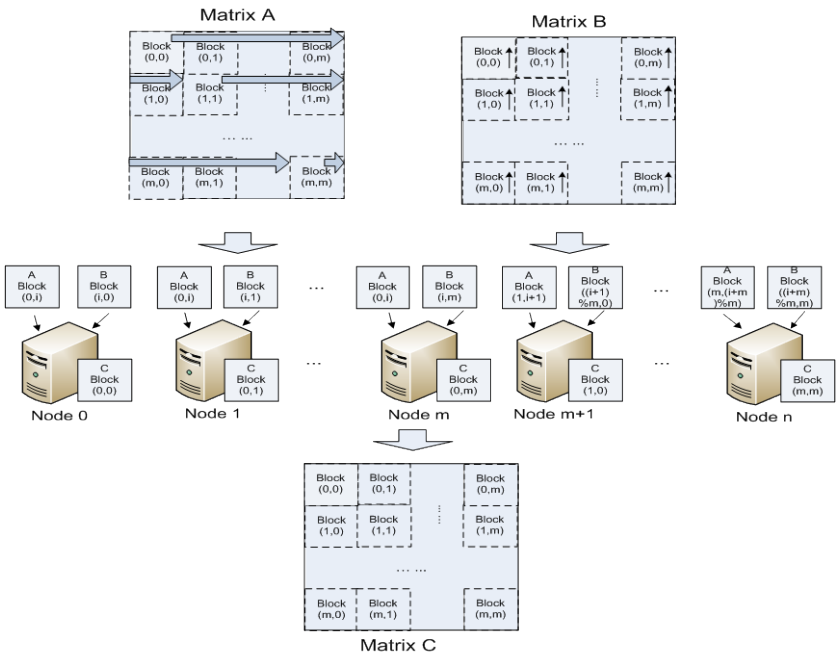


Figure 17: Structure of the 2D Decomposition Algorithm on Iteration I with n+1 Nodes

4.3 Performance Analysis in Hybrid Parallel Model

4.3.1 Performance on Multi Core

The baseline test of Matrix Multiplication was executed on the TEMPEST cluster [Appendix B] with the three multithreaded programming models: Task Parallel Library (TPL), Thread Pool and PLINQ. The comparison of these 3 technologies is shown in Table 2 below:

Table 2. Comparison of three Multicore Technologies

Technology Name	API in use	Optimization	Description
Thread Pool	ThreadPool.QueueUserWorkItem	User define	A common setting to use a group of threads. It is already been optimized in .NET 4, however, to further optimize the usage requires more user programming.
TPL	Parallel.For	Moderate Optimized	Parallel.For is a light weighted API to use maximum number of cores on the target machine using in a given application. It is more optimized than thread pool inside .NET 4. However, the experienced user can still optimize the usage of Parallel.For.
PLINQ	AsParallel()	Highly	Parallel LINQ (PLINQ) is a parallel implementation

		Optimized	of LINQ to Objects. It is highly optimized and can be used on any LINQ queries which make it suitable for DryadLINQ program. It doesn't require any optimization from user and it is most light weighted to user compare with thread and TPL.

Figure 18 shows performance results on a 24-core compute node with matrix size between 2,400 and 19,200 dimensions. The speed-up charts were calculated using Equation 3. T(P) standards for job turnaround time for Matrix Multiplication using multi-core technologies, where P is the number of cores across the cluster. T(S) refers to the job turnaround time of sequential Matrix Multiplication on one core.

$$\text{Speed-Up} = T(S)/T(P) \qquad \text{(Eq. 4)}$$

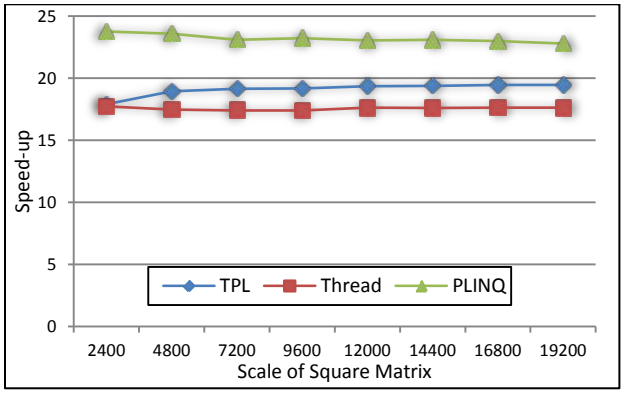


Figure 18: Speedup Charts for TPL, Thread Pool, and PLINQ Implementations of Matrix Multiplication on One Node

The parallel efficiency remains around 17 to 18 for TPL and Thread Pool. However, TPL outperforms Thread Pool as data size increases. PLINQ consistently achieves the best speed-up with values larger than 22, making parallel efficiency over 90% on 24 cores. We conclude that the main reason is due to PLINQ's memory/cache usage being optimized for large data size on multicore systems by observing metrics of context switches and system calls of the heat map from the HPC cluster manager.

4.3.2 Performance on a Cluster

We evaluated 3 different matrix multiplication algorithms implemented with DryadLINQ CTP on 16 compute nodes of the TEMPEST cluster using one core per node. The data size of input matrix ranges from 2400 x 2400 to 19200 x 19200. A variance of the speed-up definition is given in Equation 5 where T(P) stands for job turnaround time on P compute nodes. T(S') is an approximation of job turnaround time for the sequential matrix multiplication program where a fitting function [Appendix F] is used to calculate CPU time for large input data.

$$\text{Speed-up} = T(S')/T(P) \qquad \text{(Eq. 5)}$$

As shown in Figure 19, the performance of the Fox-Hey algorithm is similar to that of the Row Partition algorithm, increasing quickly as the input data size increases. The Row-Column Partition Algorithm performs the worst since it is an iterative algorithm that needs explicit evaluation of DryadLINQ queries in each iteration to collect an intermediate result. In particular, it invokes resubmission of a Dryad task to the HPC job manager during each iteration.

4.3.3 Performance of a Hybrid Model with Dryad and PLINQ

Porting multi-core technologies into Dryad tasks can potentially increase the overall performance due to extra processor cores. The hybrid model invokes Dryad for inter-node tasks and spawns concurrent threads through PLINQ. Three matrix multiplication algorithms were executed on 16 compute nodes of the TEMPEST cluster [Appendix B]. Compared to Figure 19, the speed-up charts of Figure 20 show significant performance gains by utilizing multicore technologies like PLINQ, where a factor of 9 is ultimately achieved as data size increases.

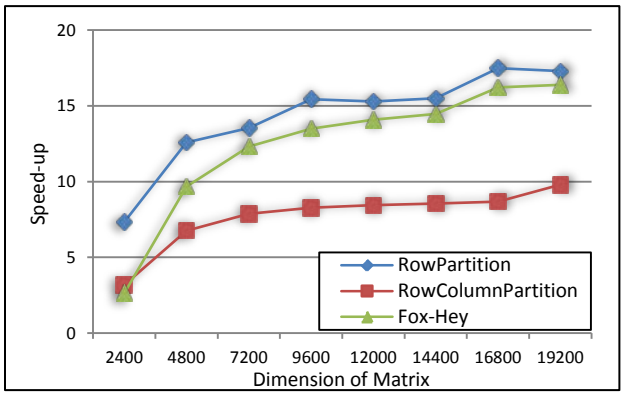


Figure 19: Speedup of Three Matrix Multiplication Algorithms Using Dryad on a Cluster

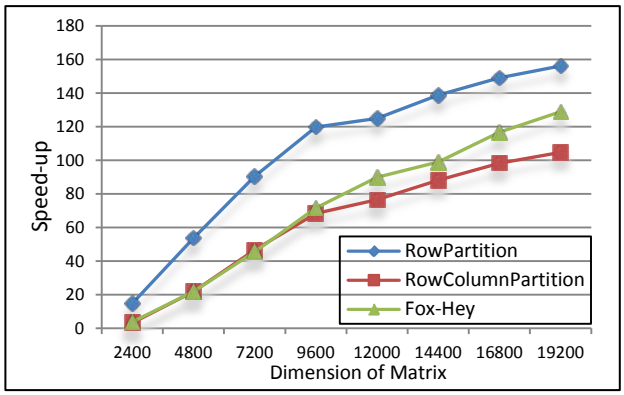


Figure 20: Speedup of Three Matrix Multiplication Algorithms Using a Hybrid Model with Dryad and PLINQ

Since the computational complexity in matrix multiplication is $O(n^3)$, which increases faster than that of the growth of communication cost $O(n^2)$, Figures 19 and 20 show that the speed-up increases with the size of input data. The Row Partition algorithm delivers the best performance for a hybrid model, as shown in Figure 20. Compared to the other 2 iterative algorithms, job submission occurs only once with the Row Partition algorithm. The Row-Column partition algorithm and the Fox-Hey algorithm both have 4 iterations and finer task granularity, leading to extra scheduling and communication overhead.

4.3.4 Performance Comparison of Three Hybrid Parallel Models

We studied three matrix multiplication algorithms in hybrid parallel programming models with Task Parallel Library (TPL), Thread, and PLINQ on multicore processors. Figure 21 shows the performance results of a 19200 by 19200 matrix on 16 nodes of the TEMPEST cluster with 24 cores on each node. In all 3 matrix multiplication algorithms PLINQ achieves better speed-up than TPL and Thread, which supports earlier performance results shown in Figure 18.

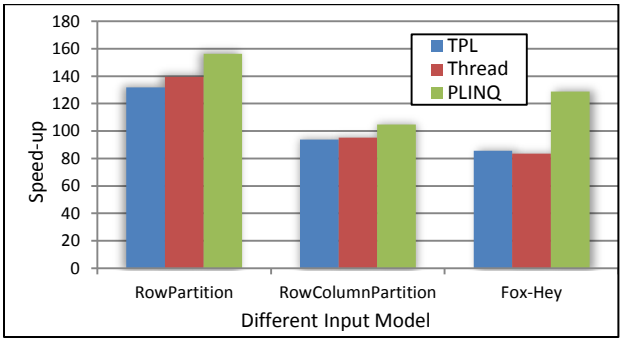


Figure 21: The Speedup Chart of TPL, Thread, and PLINQ for Three Matrix Multiplication Algorithms

When a problem size is fixed, parallel efficiency drops when multicore parallelism is used. This can be illustrated by the Row Partition algorithm running with or without PLINQ on 16 nodes (each has 24 cores), where the parallel efficiency is 17.3/16 = 108.1% when using one core per node (Figure 19), but becomes 156.2/384 = 40.68% over 384 cores (Figure 21). This is because the task granularity of Dryad on each core becomes finer and the node execution time decreases, while the overhead of scheduling, communication, and disk I/O remains the same or even increases.

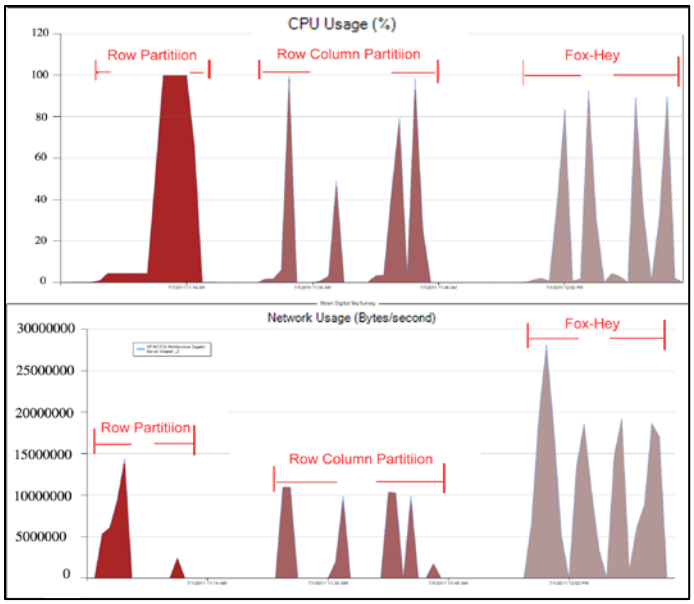


Figure 22: The CPU Usage and Network Activity on One Compute Node for Multiple Algorithms

Figure 22 shows charts of CPU utilization and network activity on one node of the TEMPEST cluster for three 19200 x 19200 matrix multiplication jobs using PLINQ. It is observed that the Row Partition Algorithm with PLINQ can reach a CPU utilization rate of 100% for a longer time than the other two approaches. Further, its aggregated network overhead is less than that of the other two approaches as well. Thus, the Row Partition algorithm with PLINQ has the shortest job turnaround time. The Fox-Hey algorithm delivers good performance in the sequential implementation due to its cache and paging advantage with finer task granularity.

Figure 23 shows the CPU and network utilization of the Fox-Hey algorithm on square matrices of 19,200 and 28,800 dimensions. Not only do they CPU and Network utilization increase with size of input data, but the rate of increase in CPU utilization is faster than that of network utilization, which follows the ratio of computation complexity O(n^3) vs. communication complexity O(n^2). The overall speed-up will continue to increase as we increase the data size. It suggests that when problem size grows large, the For-Hey algorithm will achieve high performance on low latency runtime environments.

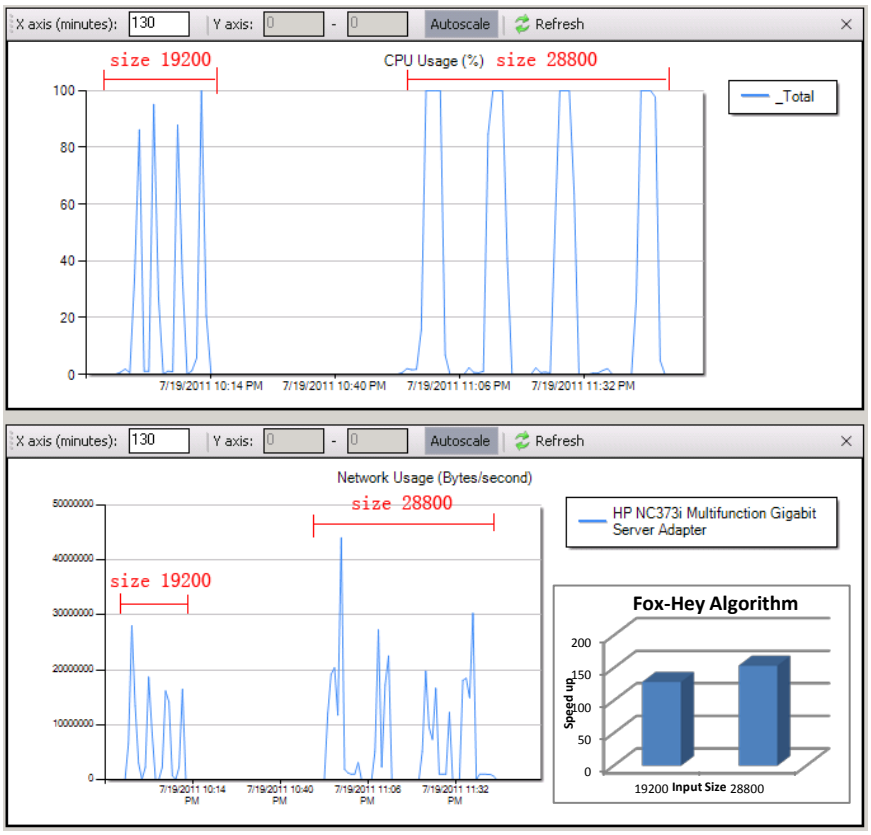


Figure 23: The CPU Usage and Network Activity for the Fox-Hey Algorithm-DSC with Different Data Sizes

4.4 Timing Analysis for Fox-Hey Algorithm on the Windows HPC cluster

We designed a timing model for the Fox-Hey algorithm to conduct detailed evaluation. Tcomm/Tflops represents the communication overhead per double point operation using Dryad on the TEMPEST cluster. Assume the M*M matrix multiplication jobs are partitioned to run on a mesh of $\sqrt{N} * \sqrt{N}$ compute nodes. The size of sub-blocks in each node is m*m, where $m = M/\sqrt{N}$. The "broadcast-multiply-roll" cycle of the algorithm, as shown in Figure 16, is repeated $\sqrt{N}$ times.

For each such cycle in our initial implementation it takes $\sqrt{N} - 1$ steps to broadcast subblocks of matrix A to the other $\sqrt{N} - 1$ nodes in the same row of mesh processors, as the network topology of TEMPEST is simply a star rather than a mesh. In each step the overhead of transferring data between two processors includes: 1) the startup time (latency T_{lat}), 2) the network time T_{comm} to transfer m*m data, and 3) the disk IO time T_{io} for writing data onto the local disk and reloading data from disk to memory. The extra disk IO overhead is common in cloud runtimes, such as Hadoop [18]. In Dryad, the data transfer usually goes through a file pipe over NTFS. Therefore, the time for broadcasting a sub-block is:

$$\left(\sqrt{\mathrm{N}} - 1\right) * (T_{lat} + m^2 * (T_{io} + T_{comm})).$$

Note that in an optimized implementation of pipelining it is possible to remove factor $\left(\sqrt{\mathrm{N}} - 1\right)$ of broadcast time.

Since the process to "roll" sub-blocks of Matrix B can be parallelized to complete within one step, the overhead is:

$$T_{lat} + m^2 * (T_{io} + T_{comm}).$$

The actual time required to compute the sub-matrix product (include the multiplication and addition) is:

$$2*m^3 * T_{flops}.$$

Therefore, the total computation time of the Fox-Hey matrix multiplication is defined as the following:

$$T = \sqrt{N} * \left(\sqrt{N} * \left(T_{lat} + m^2 * (T_{io} + T_{comm})\right) + 2 * m^3 * T_{flops}\right). \qquad (1)$$

By substituting m with $\frac{M}{\sqrt{N}}$, the equation becomes

$$T = N * T_{lat} + M^2 * (T_{io} + T_{comm}) + 2 * (M^3/N) * T_{flops} \qquad (2)$$

The last term in equation (2) is the expected "perfect linear speedup" while the other terms represent communication overheads. In the following paragraph we investigate T_{flops} and $T_{io}+T_{comm}$ by fitting measured performance as a function of matrix size.

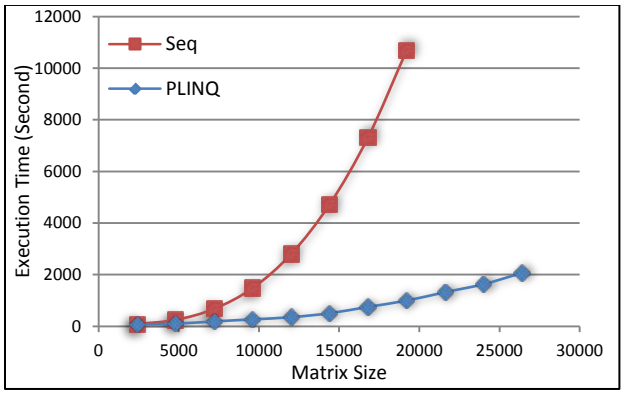


Figure 24: Execution Time of Sequential and PLINQ Execution of the Fox-Hey Algorithm

$$T_{1node\_1core} = 5.3s + 5.8us * M^2 + 35.78 * 10^{-3}us * M^3 \qquad (3)$$

$$T_{16nodes\_1core} = 21s + 3.24us * M^2 + 1.33 * 10^{-3}us * M^3 \qquad (4)$$

$$T_{16nodes\_24cores} = 61s + 1.55us * M^2 + 4.96 * 10^{-5}us * M^3 \qquad (5)$$

The timing equation for the sequential algorithm running on a one-core single node is shown in equation (3). Figure 24 and equation (4) represent the timing of the Fox-Hey algorithm running with one core per node on 16 nodes. Figure 24 and equation (5) represent the timing of the Fox-Hey/PLINQ algorithm that executes with 24 cores per node on 16 nodes. Equation (6) is the value of $\frac{T_{flops-single\ core}}{T_{flops-24\ cores}}$ for large matrices. As 26.8 is close to 24, i.e. the number of cores per node, it approximately verifies the correctness of the cubic term coefficient of equation (4) & (5). Equation (7) is the value of $\frac{(T_{comm}+T_{io})_{single\ core}}{(T_{comm}+T_{io})_{24\ cores}}$ for large matrices. The value is 2.08, while the ideal value is expected to be 1.0. We have investigated the differences between the ideal values and the measurements of equations (6) and (7), and find that a dominant issue is the effect of the cache, which improves performance in the parallel 24-core case and is not included in above formulae. The constant term in equation (3), (4), and (5) accounts for the cost of initialization of the computation, such as runtime startup and the allocation of memory for matrices.

$$\frac{T_{flops-single\ core}}{T_{flops-24\ cores}} = \frac{1.33*10\text{^}-3}{4.96*10\text{^}-5} \approx 26.8 \quad (6)$$

$$\frac{(T_{comm}+T_{io})_{single\ core}}{(T_{comm}+T_{io})_{24\ cores}} = \frac{3.24}{1.55} \approx 2.08 \quad (7)$$

$$\frac{T_{io}}{T_{comm}} \approx 5 \quad (8)$$

Equation (8) represents the value of $\frac{T_{io}}{T_{comm}}$ for large submatrix sizes. The value illustrates that though the disk IO cost has more effect on communication overhead than does network cost, they are of the same order for large sub-matrix sizes, thus we assign the sum of them as the coefficient of the quadratic term in equation (2). Besides, one must bear in mind that the so-called communication and IO overhead actually include other overheads, such as string parsing and index initialization, which are dependent upon how one writes the code.

4.5 Evaluation and Findings

We investigated hybrid parallel programming models with three kernel matrix multiplication applications.

f) We showed how integrating multicore technologies into Dryad tasks can increase the overall utilization of a cluster.

g) Further, different combinations of multicore technologies and parallel algorithms perform differently due to task granularity, caching, and paging issues.

h) We also find that the parallel efficiency of jobs decreases dramatically after integrating these multicore technologies given that the task granularity becomes too small per core. Increasing the scale of input data can alleviate this issue.

5 Distributed Grouped Aggregation

5.1 Introduction

Distributed Grouped Aggregation is a core primitive operator in many data mining applications, such as sales data summarizations, log data analysis, and social network influence analysis. We investigated the usability and performance of a programming interface for a distributed grouped aggregation in DryadLINQ CTP. Three distributed grouped aggregation approaches were implemented: Hash Partition, Hierarchical Aggregation, and Aggregation Tree.

PageRank is a well-known web graph ranking algorithm. It calculates the numerical value of a hyperlinked set of web pages, which reflects the probability of a random surfer accessing those pages. The process of PageRank can be understood as a Markov Chain that needs recursive calculation to converge. In each iteration the algorithm calculates a new access probability for each web page based on values calculated in the previous computation. The iterations will not stop until the values between two subsequent rank vectors are smaller than a predefined threshold. Our DryadLINQ PageRank implementation uses the ClueWeb09 data set [22], which contains 50 million web pages.

We used the PageRank application to study the features of input data that affect the performance of distributed grouped aggregation implementations. In the end, the performance of Dryad distributed grouped aggregation was compared with four other execution engines: MPI, Hadoop, Haloop [19], and Twister [20][21].

5.2 Distributed Grouped Aggregation Approaches

Figure 26 shows the workflow of the three distributed grouped aggregation approaches implemented with DryadLINQ.

The Hash Partition approach uses a hash partition operator to redistribute records to compute nodes so that identical records are stored on the same node and form the group. Then the operator usually aggregates certain values of the records in each group. The hash partition approach is simple in implementation, but causes a lot of network traffic when the number of input records becomes very large.

A common way to optimize this approach is to apply partial pre-aggregation, which first aggregates the records on local compute nodes and then redistributes aggregated partial results across a cluster based on their keys. The optimized approach is better than a direct hash partition because the number of records transferring across a cluster is drastically reduced after the local aggregation operation. Further, there are two ways to implement partial aggregation: hierarchical aggregation and tree aggregation. A hierarchical aggregation usually consists of two or three synchronized aggregation stages. An aggregation tree is a tree graph that guides a job manager to perform partial aggregation for many subsets of input records.

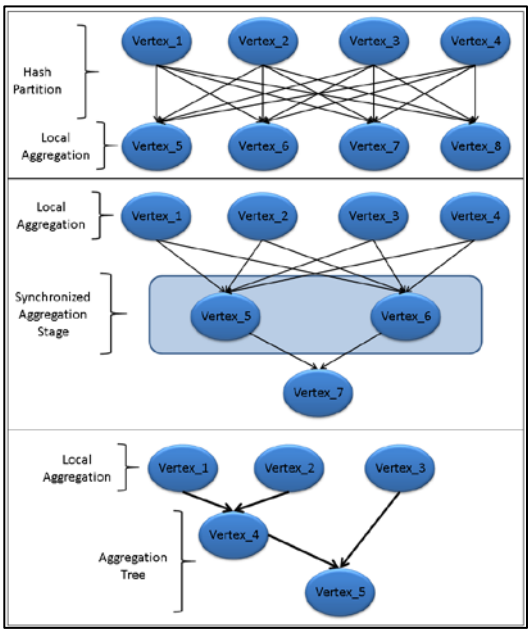


Figure 26: Three Distributed Grouped Aggregation Approaches

DryadLINQ can automatically translate a distributed grouped aggregation query into an optimized aggregation tree based on data locality information. Further, Dryad can adaptively change the structure of the aggregation tree, which greatly simplifies the programming model and enhances its performance.

5.2.1 Hash Partition

The implementation of PageRank in Appendix I used *GroupBy* and *Join* operators in DyradLINQ.

Page objects are used to store the structure of a web graph. Each element ***Page*** *<url_id, <destination_url_list>>* contains a unique identifier number page.key and a list of identifiers specifying all pages in the web graph that **page** links to. We construct the *DistributedQuery<Page>* **pages** objects from adjacency matrix files with function *BuildPagesFromAMFile()*. The **rank** *object <url_id, rank_value>* is a pair containing the identifier number of a page and its current estimated rank value. In each iteration the program joins the **pages** with **ranks** to calculate the partial rank values. Basically, it combines records from **pages** and **ranks** tables using a common keyword url_id. Then *GroupBy()* operator redistributes the calculated partial rank values across cluster nodes and returns the *IGrouping* objects, *DistributedQuery<IGrouping<url_id, rank_value>>*, where each *IGroup* represents a group of partial rank objects with all destination urls from one source URLs. The

grouped partial rank values are summed up as the final rank values and are used as input rank values for the next iteration [23]. In the above implementation, *GroupBy()* operator can be replaced by *HashPartition()* and *ApplyPerPartition()* as follows[24]:

5.2.2 Hierarchical Aggregation

The PageRank implementation using hash partition would not be efficient when the number of output tuples is much less than that of input tuples. In this scenario, we implemented PageRank with hierarchical aggregation, which consists of three synchronized aggregation stages: 1) user-defined Map tasks, 2) DryadLINQ partitions, and 3) final PageRank values. In stage one, each user-defined Map task calculates the partial results of some pages that belong to the sub-web graph represented by the adjacency matrix file. The output of a Map task is a partial rank value table, which is merged into the global rank value table in a later stage.

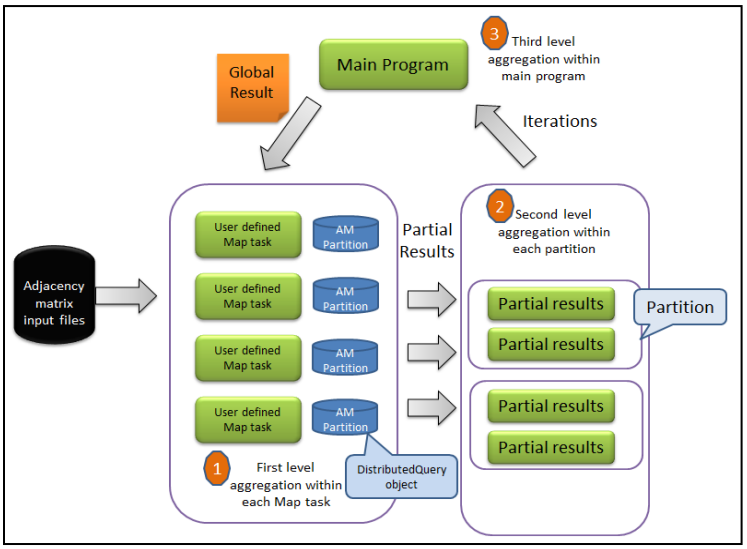


Figure 27: Hierarchical Aggregation in DryadLINQ PageRank

5.3.3 Aggregation Tree

The hierarchical aggregation implementation may not perform well in an inhomogeneous computation environment, which varies in network bandwidth, CPU, and memory capacities. As hierarchical aggregation has several global synchronization stages, the overall performance was determined by the slowest task. In this scenario, the aggregation tree approach is a better choice. It can construct a tree graph to guide the aggregation operations for many subsets of input tuples, which reduces intermediate data transfer. In the ClueWeb data set, URLs are stored in alphabetical order. Web pages that belonged to the same domain were likely being saved within one adjacency matrix file. By applying partial grouped aggregation to each adjacency matrix file in the hash partition stage, intermediate data transfer can be greatly reduced. The following implementation of PageRank uses the *GroupAndAggregate()* operator.

The *GroupAndAggregate* operator supports optimization of the aggregation tree. To analyze partial aggregation in detail, we simulated *GroupAndAggregate* with the HashPartition and *ApplyPerPartition* operators. There are two steps of *ApplyPerPartition*: one is to perform pre-partial aggregation on each sub-web graph; the other is to aggregate the partially aggregated results for global results.

5.3 Performance Analysis

5.3.1 Performance in Different Aggregation Strategies

We conducted performance measurements of PageRank with three aggregation approaches: hash partition, hierarchical aggregation, and tree aggregation. In the experiments, we split the entire ClueWeb09 graph into 1,280 partitions, each of which was processed and saved as an adjacency matrix (AM) file. The characteristics of input data are described below:

No of AM Files	File Size	No of Web Pages	No of Links	Ave Out-degree
1,280	9.7 GB	49.5 million	1.40 billion	29.3

The program ran on 17 compute nodes from the TEMPEST cluster. Figure 28 shows that tree aggregation is faster than hash partition due to the optimization of partial aggregation. Hierarchical aggregation outperforms the other two approaches because of coarse task granularity.

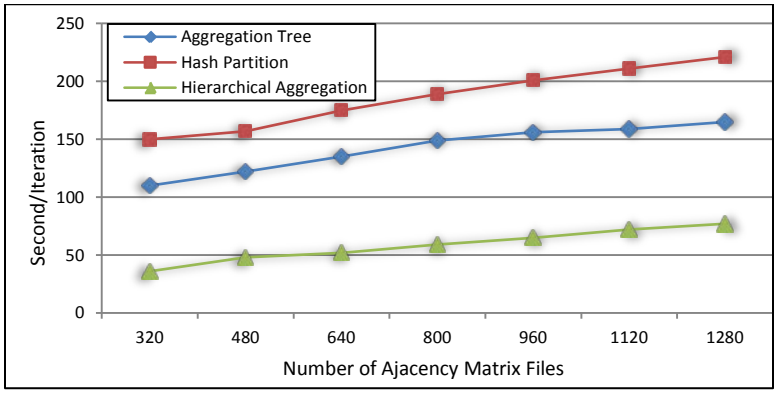


Figure 28: Time to Compute PageRank per Iteration by Three Aggregation Approaches Using Clue-web09 Data on 17 Compute Nodes of TEMPEST

Figure 29 provides CPU utilization and network utilization information of the three aggregation approaches obtained from the HPC cluster manager. It is apparent that hierarchical aggregation requires much less network bandwidth than the other two.

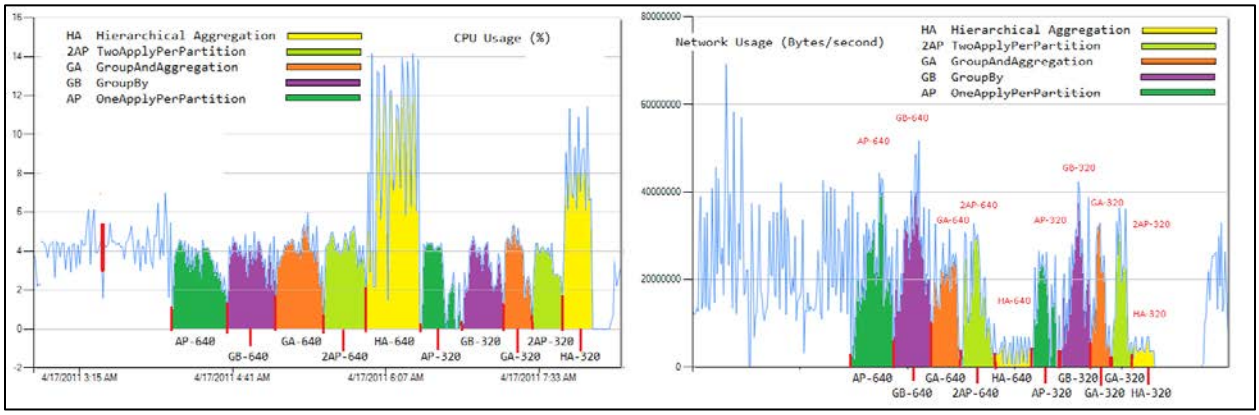


Figure 29 CPU (left) and Network Utilization (right) of Different Aggregation Strategies

The hierarchical aggregation and aggregation tree approaches work well when the number of output tuples was much smaller than input tuples. The hash partition worked well when the number of output tuples was

larger than input tuples. To describe how the ratio between input and output tuples affects the performance of different aggregation approaches, we define the data reduction proportion (DRP).

$$DRP = \frac{number\ of\ output\ tuples}{number\ of\ input\ tuples} \qquad \text{(Eq. 6)}$$

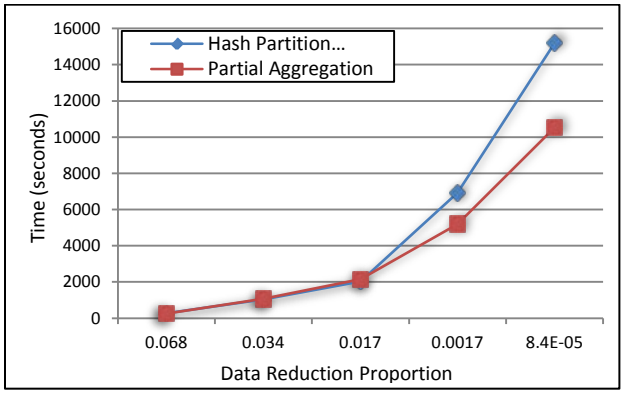


Figure 30: Time required for PageRank Jobs for Two Aggregation Approaches with Different DRP

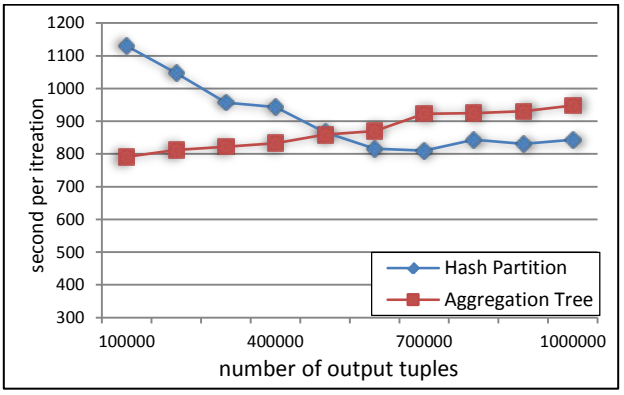


Figure 31: Time Per Iteration for Two Aggregation Approaches with Different Numbers of Output Tuples (from 100,000 to 1,000,000) When the Number of Input Tuples is 4.3 Billion

Assume M input tuples are evenly distributed among N compute nodes. In a hash partition approach, tuples with the same key are hashed into one group, which requires M aggregation operations. In a partial aggregation approach, the average number of input tuples with the same key is about M/N on each node, which requires M/N aggregation operations on each node and generates N partial aggregated tuples in total. Further, it needs N more aggregation operations to form the final aggregated tuple. Thus, the total number of aggregation operations for the M tuples is (M/N)*N+N. The average number of aggregation operations of each tuple from the two approaches is as follows:

$$\begin{cases} O\left(\frac{M}{M}\right) = O(1) \\ O\left(\frac{M+N}{M}\right) = O(1 + N * DRP) \end{cases} \qquad \text{(Eq. 7)}$$

In many applications, DRP is much smaller than the number of compute nodes, which suggests that the overhead of applying partial aggregation is small compared to the hash partition. Taking the word count application as an example, documents with millions of words may consist of only several thousand words that occur frequently. Word count is very suitable for applying partial aggregation. In PageRank, as the web graph structure obeys Zipf's law, DRP is higher. Thus, the partial aggregation approach may not deliver good performance when applied to PageRank [23].

To quantify the impact of DRP on different aggregation approaches, we ran PageRank with web graphs of different DRP values. As shown in Figure 30, when DRP is smaller than 0.017, the partial aggregation performs better than hash partition aggregation. When DRP is bigger than 0.017, there is not much difference between these two aggregation approaches. The results of Figures 30 and 31 indicate the changes in performance when the input tuples are fixed with varying output tuples.

5.3.2 Comparison with Other Implementations

We implemented a PageRank application with five runtimes: DryadLINQ, Twister, Hadoop, Haloop, and MPI using ClueWeb data, which is listed in Table 8 of Appendix G. Parallel efficiency T(S)/(P*T(P)) (refer to Eq. 1) is used to compare the performance of five implementations. T(P) stands for job turnaround time of parallel PageRank, where P represents the number of cores. T(S) is the time of sequential PageRank on one core.

Figure 32 shows that all parallel efficiency charts are noticeably smaller than 5%. PageRank is a communication-intensive application, where the computation complexity of PageRank is $O(N^2)$ while its communication complexity is $O(N^2)$. As the communication overhead per float point calculation of PageRank is high, the bottlenecks of PageRank applications are network, memory, and CPU. Therefore, a major challenge is to reduce synchronization cost among tasks.

MPI, Twister, and Haloop outperform Dryad and Hadoop implementations for the following reasons: 1) MPI, Twister, and Haloop cache static data in memory between iterations; and 2) Haloop uses chained tasks without the need to restart task daemons for each iteration. Dryad is faster than Hadoop, but is slower than MPI and Twister. Dryad can chain DryadLINQ queries together and thereby save in communication cost, but it has higher scheduling overhead for each Dryad vertex. Hadoop has the lowest performance in writing intermediate data to HDFS between interactions.

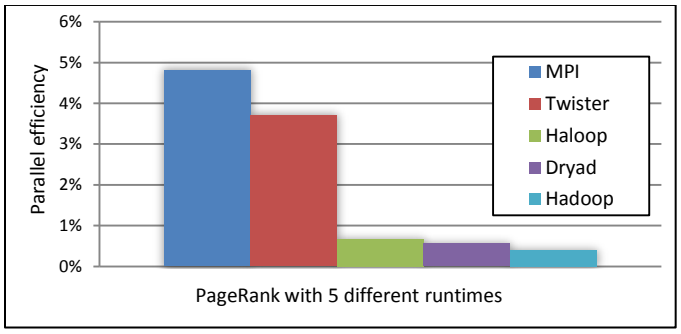


Figure 32: Parallel Efficiency of Five PageRank Implementations

5.3.3 Chaining Tasks Within BSP Jobs

Dryad can chain the execution of multiple DryadLINQ queries together using late evaluation technology. The chained DryadLINQ queries will not get evaluated until the program explicitly accesses queries. Figure 33 shows that after chaining DryadLINQ queries, performance increases by 30% for 1280 adjacency matrix files of PageRank over 10 iterations.

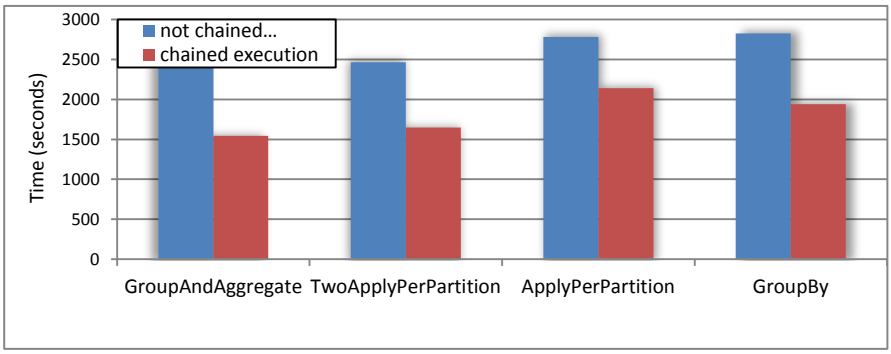


Figure 33: Performance Difference of Chained and Unchained DryadLINQ Queries

Although DryadLINQ chains the execution of queries, it does not support the execution of jobs that consist of Bulk Synchronous Parallel [26] (BSP) style tasks very well. For example, in DryadLINQ hierarchical aggregation PageRank, the program has to be resubmitted to a Dryad job on the HPC scheduler for every synchronization step that calculates the global PageRank value table.

5.4 Evaluation and Findings

We investigated three distributed grouped aggregation approaches with DryadLINQ CTP. Programmability and performance of these approaches were studied using the PageRank application. The results show correlations with different ratio of data reduction proportion (DRP).

i) **Partial pre-aggregation requires more memory than hash partition**.
j) **Hash partition has larger communication overhead than partial pre-aggregation.**
k) Detailed implementation of partial pre-aggregation such as accumulator fullhash and iterator fullhash/fullsort, has different requirements for memory and network bandwidth.

6 Programming Issues in DryadLINQ CTP

6.1 Class Path in Working Directory

We found the following issue when running DryadLINQ CTP SWG jobs: Dryad can automatically transfer files required by a user program to remote working directories on each compute node. In order to save storage space in compute nodes, Dryad does not copy all DLL and shared libraries to working directory for each task. Instead, it stores only one copy of shared libraries in the job working directory shared by all Dryad tasks. When running jobs, the Dryad vertex can add the job working directory into the class path of DryadLINQ program. So all Dryad tasks can refer to DLLs and shared libraries without a problem. **However, when Dryad tasks invoke a third party executable binary file as process, Dryad process is not aware of the class path that the Dryad vertex maintains, and it throws out an error : "required file cannot be found."**

6.2 Late Evaluation in Chained Queries within One Job

DryadLINQ can chain the execution of multiple queries by applying late evaluation technology. This mechanism allows further optimization of the execution plan of DryadLINQ queries. As shown in the following code, DryadLINQ queries within different iterations are chained together and will not get evaluated until the Execute() operator is invoked explicitly. The integer parameter "iterations" is supposed to increase by one at each iteration. However, **when applying late evaluation, DryadLINQ only evaluates the iterations parameter at the last iteration (which is nProcess -1 in this case) and uses that value for further execution of all the queries including previous iterations**. This imposes an ambiguous variable scope issue, which should be mentioned in the DryadLINQ programming guide.

```
for (int iterations = 0; iterations < nProcesses; iterations++)
{
  inputACquery = inputACquery.ApplyPerPartition(sublist => sublist.Select(acBlock =>
  acBlock.updateAMatrixBlockFromFile(aPartitionsFile[acBlock.ci],    iterations,nProcesses)));
  inputACquery = inputACquery.Join(inputBquery, x => x.key, y => y.key, (x, y) =>
  x.taskMultiplyBBlock(y.bMatrix));
  inputBquery = inputBquery.Select(x => x.updateIndex(nProcesses));
}
inputACquery.Execute();
```

6.3 Serialization for a Two Dimensional Array

DryadLINQ and Dryad Vertex can automatically serialize and unserialize the standard .NET objects. However, **when using a two dimensional array, objects in matrix multiplication, and PageRank applications, the program will throw out an error message when a Dryad task tries to access unserialized two**

dimensional array objects on remote compute nodes. We investigated the serialization code being automatically generated by DryadLINQ and found it may not be able to unserialize two dimensional array objects correctly. The reason for this needs further investigation.

```
private void SerializeArray_2(Transport transport, double[][]value)
{
   if ((value == null)){
      transport.Write(((byte)(0)));
       return;
   }
   transport.Write(((byte)(1)));
   int count = value.Length;
   transport.Write(count);
   for (int i=0; (i<count);i=(i+1)){
       SerializeArray_4(transport, value[i]);
   }
}
```

6.4 Fault tolerant in DryadLINQ CTP

6.4.1 Failures and Fault Tolerance

DryadLINQ supports fault tolerance as it is a major advantage of new parallel frameworks like MapReduce over the traditional parallel runtimes like MPI. We examine Dryad fault tolerance with respects to following types of failure: process level failure, operating system (OS) level failure, node level failure, and multiple node failure as shown in Figure 34. These tests were executed on 7 nodes of TEMPEST Cluster [Appendix B] using SWG application. The input data consists of 2,000 gene sequences that were partitioned into 12 blocks with 64×64 for each sub-block.

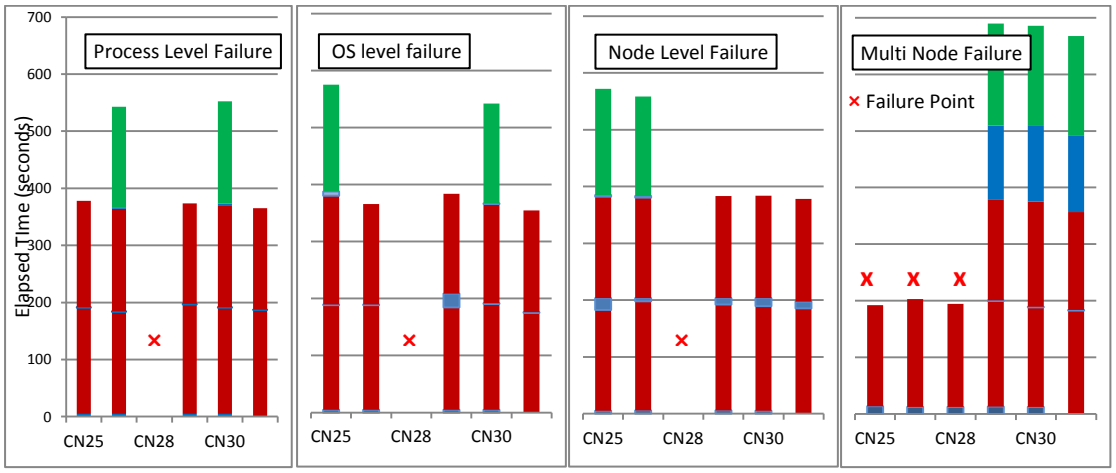


Figure 34: SWG Execution Timeline for Different Failure Types

In Figure 34, X-axis labels compute node (e.g. cn25 ~ cn31) and Y-axis is the elapsed time from the start of computation. A failed note is marked by "x". A red bar marks the time frame of a particular compute node doing computation, a blue bar refers to the time frame for scheduling a new partition, and a green bar means this is a partition shifted to this node due to the failure.

Dryad usually handles a failure by re-scheduling the failed vertex job to other available compute nodes. Once a failed node is excluded from the list of available nodes, it will not be used again during this job execution, even after the failure is fixed. There are two special node failures – HPC head node and Dryad Job Manager node. Although both are a single point failure, the former doesn't not have impact current job's completion

while the latter currently has no fault tolerance support from Dryad. A summary of the fault tolerance features for Dryad is listed in Table 3.

Table 3: Fault Tolerance Features of Dryad

	Failure types	Failure Description	Fault Tolerance Strategy
1	Process	Process HPCDryadVertex.exe failed Operating system crashed	The job manager will re-schedule the failed vertex job on this compute node to other available compute nodes. No new vertexes assignment to this compute node after it is brought back online or the operating system resumes.
2	Compute Node	Compute node offline or crashed	The job manager will have the same fault tolerance strategy as the Process failure.
3	HPC Head Node	HPC head node offline or crashed	Once the job is submitted, the status of head node makes no impact on current running Dryad job.
4	Dryad Job Manger Node	Dryad job manager node offline or HPCLinqToDryadJM.exe crashed	The Dryad job manager is a **single point of failure**. Currently there is no fault tolerance support for this scenario.
5	Multi-Node	Multiple compute node offline or crashed	Dryad can handle multi nodes failure by assigning all the failed vertices to other available compute nodes. Figure 1 (rightmost) shows the recovery of 3 out of 6 compute nodes failure

6.4.2 Partition Granularity and Fault Tolerance

Unlike static scheduling used in DryadLINQ 2009, DryadLINQ CTP divides data into a predefined number of partitions, which is twice the number of compute nodes by default. This number is customizable by users. DryadLINQ CTP dispatches the partitions across compute nodes dynamically. We've shown in our earlier work [2] that partition granularity has huge impact on load re-balancing in fault tolerance of Dryad. To evaluate the performance of recovery from a failure, we used the same SWG experiment but selected a set of partitions from {6, 12, 24, 48, 96} instead of the default partition. CN-28 was taken offline after 50% workload has been processed to simulate a node failure. The result in Figure 35 (left) suggests that when the partition granularity is large, the shifted workload (colored in green) was re-scheduled to a limited number of node leaving most nodes idle. When the partition granularity is small, the distribution of shifted workload is well balanced. However, fine partition granularity may cause extra scheduling cost that slows down overall computation as in Figure 35 (right).

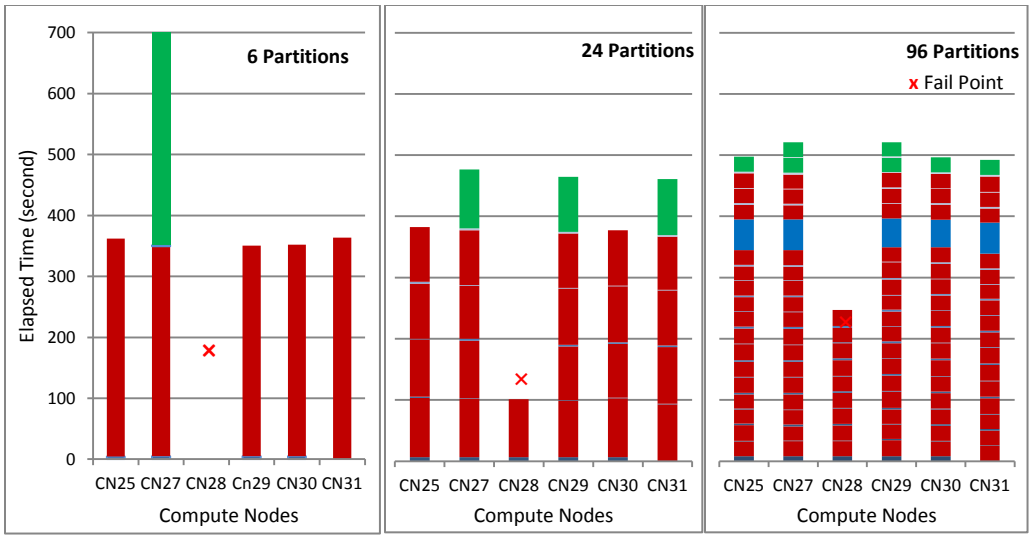


Figure 35: SWG Execution Timeline with Varied Partition Numbers with One Node Failure

The optimal number of partitions is a moderate number with respect to both load balance and scheduling cost. Figure 36 shows that the optimal number of partitions is 24, in between the minimal number of 6 and the maximum number of 96.

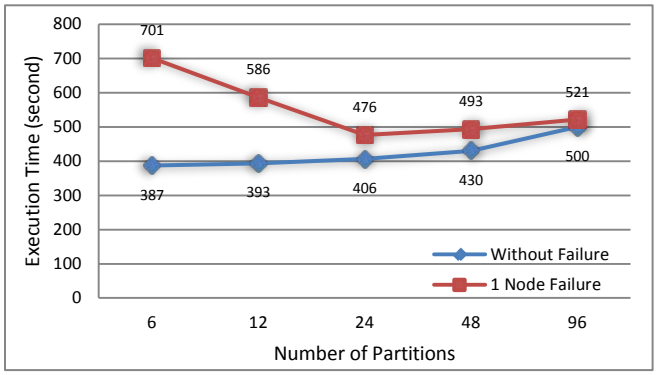


Figure 36: Average SWG Turnaround Time Different Partition Granularities

6.4.3 Evaluation and Findings

DryadLINQ CTP is able to tolerate up to 50% compute node failure. **The job manager node failure is a single point failure that has no fault tolerance support from Dryad**. The recovery speed of a failure is in favor of small granularity of partition.

7 Classroom Experience with Dryad

7.1 Dryad in Eduction

Dryad/DryadLINQ has applicability in a wide range of applications in both industry and academia, which include: image processing in WorldWideTeleScope, data mining in Xbox, High Energy Physics (HEP), SWG, CAP3, and PhyloD in bioinformatics applications. An increasing number of graduate students in the computer science department, especially Master's students, have shown interest and a willingness to learn Dryad/DryadLINQ in classes taught by Professor Judy Qiu at Indiana University.

In the CSCI B649 Cloud Computing for Data Intensive Science course, 8 Master's students selected topics related to Dryad/DryadLINQ as a term-long project. The following are three projects completed by the end of the Fall 2010 semester:

1) Efficiency and Programmability of matrix multiplication with DryadLINQ;
2) The Study of Implementing PhyloD application with DryadLINQ;
3) Large Scale PageRank with DryadLINQ

Projects 2 and 3 were accepted as posters at the CloudCom2010 conference hosted in Indianapolis, IN. In the 2011 Spring semester, two students in CSCI B534 Distributed Systems studied Dryad/DryadLINQ as a term-long project and contributed small, but solid results for this report.

7.2 Concurrent Dryad jobs

These two courses provide an excellent educational setting for us to study how students would utilize a cluster to understand the theories and applications of large-scale computing:

1. Ordinary classes contain 30-40 students, which can form 10 – 15 groups;
2. Student groups do experiments in a simulation environment where each group runs jobs on 1 to 8 compute nodes;
3. Students may not submit jobs until the due date is approaching. In another words, when a deadline is forthcoming, there are many jobs in the queue waiting to be executed while at other times the cluster may be left idle.

Based on the above observations, **it is critical to run multiple Dryad jobs simultaneously on a HPC cluster**, especially in an educational setting. In the DryadLINQ CTP, we managed to allow each job to reside in a different node group as shown in Figure 37. In this way, a middle-sized cluster with 32 compute nodes can sustain up to 16 concurrent jobs. **However, this feature is not mentioned in either Programming or Guides**.

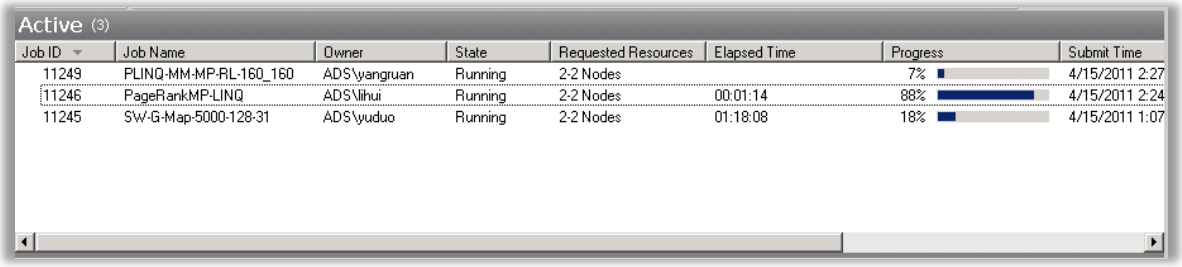


Figure 37: Concurrent Job Execution in the DryadLINQ CTP Version

Although concurrent jobs are enabled, the overall resource utilization of Dryad is not perfect. Figure 38 shows the CPU usage on each node while the jobs execution is displayed in Figure 37. Dryad jobs are assigned to compute node STORM-CN01 through STORM-CN06. Each compute node group contains 2 compute nodes, where only one of the nodes does actual computation. The reason is that **every Dryad job requires an extra node acting as a job manager**. **CPU usage of this particular node is low** and seldom exceeds 3%. In a cluster of 8 nodes, the overall usage of three concurrent jobs is only about 37%.

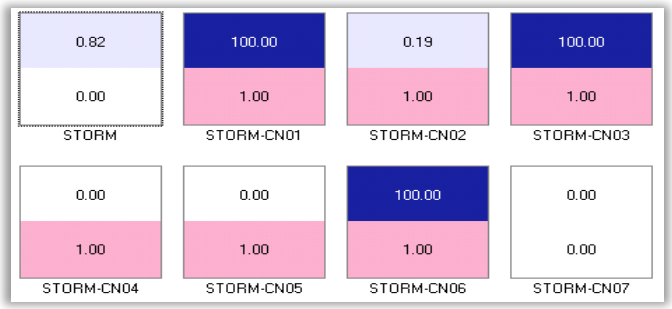


Figure 38: CPU Usage for Concurrent Job Execution

8 Comparison of DryadLINQ CTP and Hadoop

8.1 Key features in which DryadLINQ CTP outperforms Hadoop

Table 4 summarizes highlighted features and their comparison across DryadLINQ CTP and Hadoop. We now extend the discussion to the following five features in which Dryad outperforms Hadoop.

1) **Higher-level programming interface**
Hadoop's Pig [35] is a natural analogue to DryadLINQ. DryadLINQ provides a programming model based on LINQ and a rich set of data objects by integrating with .NET. **DryadLINQ performs better than Pig when processing relational queries and iterative MapReduce tasks (section 8.2.2)**. One main reason for this underlies the DAG execution model, which is more expressive and flexible than that of Hadoop. Another reason for the performance difference is that Pig translates relational queries into a set of MapReduce circles for execution. YSmart [36] is another SQL-to-MapReduce translator that outperforms Pig by exploring query correlations. Note that DAG model in Dryad can be implemented as Workflow and Workflow plus Hadoop is an interesting approach. Further although PIG is not good on relational queries, there are many applications where data is stored in files where PIG works well.

2) **Inter-Task communication**
By default, **Dryad uses files to transfer data between vertices**. However, **Dryad can also take advantage of TCP Pipe, and shared-memory FIFO to set up channels between vertices in some scenarios**. These approaches avoid the overhead of materializing intermediate data into disk. In addition, **Dryad can compress data based on the data type and compress schemes defined within DryadLINQ optimizer**. In Hadoop, reduce tasks use HTTP to fetch the output files of map tasks. This approach causes low data fetching throughput and does not utilize high bandwidth network in data centers. The papers by Wang and Gu et al. [27, 33] discussed optimizing network use and presented their solution of using RDMA and a modified TCP/IP protocol.

3) **Data-locality-aware scheduling**
Hadoop utilizes data-locality information when scheduling map tasks to TaskTrackers, but not for reduce tasks. **Dryad can maintain the data locality for processing vertices during different computation stages of Dryad jobs**. **The new vertex can reuse the static data stored in the memory of the previous vertex on the same compute node**. This feature can significantly accelerate iterative MapReduce applications, such as PageRank. The Paper by Yu et al. [4] studied DryadLINQ PageRank and claimed that when joining a web graph table with a rank table, Dryad vertices on the same compute node in different iterations were able to reuse the graph table stored in the memory of that machine. Note that it may be preferable to tackle iteration directly with a system like Daytona or Twister(4Azure) supporting directly Iterative MapReduce.

4) **Pipelining technology**
Pipelining technology can hide network overhead and increase the parallelism of overall computation. **Dryad runtime claims to ensure efficient, pipelined execution by running asynchronous vertices while still performing the simple abstraction of reading and writing a single record at a time**. To support fault tolerance, Hadoop materializes intermediate data into disk and does not support pipelining for execution between (Map, Shuffle, Merge, Reduce) stages. The Hadoop community is becoming aware of this issue, and the papers by Wang and Condie et al. [27,29,37] showed their effort to pipeline the execution of different Hadoop computation stages.

5) **Distributed grouped aggregation**
Distributed grouped aggregation is a core primitive of many distributed programming models. Dryad and Hadoop both support this processing pattern with different strategies and implementation details. **Dryad supports Hash Partition, Hierarchical Aggregation, and Aggregation Tree strategies (section 5.2)**. **Hadoop does not support the aggregation tree strategy that allows aggregation applications to perform multiple asynchronous aggregation computation instances concurrently**. Furthermore, the paper by Isard [3] claims to implement distributed grouped aggregation with six different approaches: FullSort, PartialSort, Accumulator-FullHash, Accumulator-PartialHash, Iterator-FullHash, and Iterator-PartialHash. Hadoop supports the hierarchical aggregation strategy that performs aggregation within the local combining and reducing stages. Hadoop uses the FullSort approach, which is similar to the PartialSort implementation of DryadLINQ due to the fixed size of each input partition. In Hadoop a very large number of small input partitions may degrade performance due to the scheduling overhead of short-lived processes. Figure 42 shows that DryadLINQ outperformed Hadoop while running a K-means application, which needs to aggregate the position of points within the same group. It is possible to enhance Hadoop with additional aggregation collectives. This idea underlies Indiana University's Map Collective model [40].

Table 4 Comparison of DryadLINQ CTP and Hadoop

	Key Features	**DryadLINQ CTP**	**Hadoop 0.20.0**	**Comments**
	Programming Interface			
1	Execution model	DAG of data flowing between operations [3,4,7]	Map, Shuffle, Merge, Reduce stages [34]	DAG is more flexible than Hadoop MapReduce for expressing data flow processing
2	Data Model	Unified data model defined within .NET and LINQ	Defined by Hadoop classes: InputFormat, InputSplit, RecordReader	DryadLINQ provides a friendly data model by leveraging .NET and LINQ
3	Programming Interface	1) Based on LINQ [4,7] model with an interface extension for Dryad 2) Able to use relational operators defined in LINQ	1) Map and Reduce class [34] 2) Does not natively support relational operations that have multi-heterogeneous input data sets	There is no public document about the Dryad lower level API
4	Higher-Level Programming Language	2) DryadLINQ supports standard query operations defined within LINQ such as *Select* and *Join*	3) Pig allows developers to utilize relational queries in Hadoop 4) YSmart is another SQL-to-MapReduce translator	DryadLINQ outperforms Pig when processing relational datasets

		3) Evaluations of queries are converted into DAG	that outperforms Pig	
	Job scheduling			
5	Data-locality-aware scheduling	DryadLINQ optimizes the execution plan according to data locality information, and it can maintain the data locality for different execution stages [25]	Hadoop schedules map tasks to TaskTrackers according to data locality information, but not for reduce tasks [18, 39]	DryadLINQ provides better data-locality-aware scheduling implementation than Hadoop
6	Load balance	1) Dynamic assigning available tasks to idle resources 2) Easy to tune the task granularity by simply changing the parameters within the related interface	1) Dynamic assigning available tasks to idle resources 2) Tune the task granularity according to NumofMapTasks, Blocksize, InputFormat	Both DryadLINQ and Hadoop support dynamic scheduling strategy. DryadLINQ provides better interface for developers to tune the task granularity.
	Performance Issues			
7	Data movement, communication	DryadLINQ provides three channel protocols: File (the default), TCP Pipe, and Shared-memory FIFO[3] Note: RDMA is available in Windows8	Hadoop uses HTTP to transfer data between Map tasks and Reduce tasks during shuffling [18]	DryadLINQ provides better data transferring approaches than Hadoop
8	Pipelining within one job	The evaluation of DryadLINQ query is split into computation stages whose executions are pipelined [4, 7, 25]	Hadoop does not support pipelined execution between (Map, Shuffle, Merge, Reduce) stages, as the intermediate data is materialized into local disks [27, 28]	DryadLINQ outperforms Hadoop in pipelining execution
9	Pipelining between Jobs (iterative MapReduce)	Chain the execution of multiple queries by using late evaluation technology, TCP pipe, and shared memory FIFO [4,7]	Hadoop cannot pipeline the execution of jobs, as it needs to materialize the output of MapReduce jobs into disk (HDFS) when jobs are done [27,28]	In DryadLINQ the pipelining can be broken when it explicitly evaluates the queries or materializes output results to disk
10	Aggregation	DryadLINQ can dynamically build Aggregation Trees [10] based on data locality information. DryadLINQ implements distributed grouped aggregation with six approaches	Hadoop supports a hierarchical aggregation strategy Implementation: FullSort (Iterator based)	In general, aggregation tree (DryadLINQ) outperforms hierarchical aggregation (Hadoop)

8.2 Programming Models Analysis of DryadLINQ CTP and Hadoop

The programming model is a critical factor in determining runtimes. One reason why Cloud is more popular than Grid is that Cloud has several practical programming models. In this section we analyze the programming models of DryadLINQ CTP and Hadoop. Table 11 in Appendix K shows eleven applications implemented with DryadLINQ by SALSA group. We classify them into three programing models in Table 12 in Appendix K, which includes pleasingly parallel programming models, relational data set processing, and iterative MapReduce.

8.2.1 Pleasingly parallel programming model

Developers can implement pleasingly parallel applications with DryadLINQ and MapReduce without difficulty. DryadLINQ provides a friendly interface (compared to Hadoop) by utilizing data model and programming interfaces defined within .NET and LINQ. For some applications, writing DryadLINQ distributed applications are similar as writing SQL queries.

There is no big performance difference between DryadLINQ and Hadoop for pleasingly parallel applications unless there is a large number of input data. This is because DAG can be simplified as MapReduce when processing pleasingly parallel applications. The advance features of DryadLINQ, such as pipelining, are rarely used in this case. The performance difference between them for processing big data was mainly caused by scheduling overhead for large number of tasks and work load distribution among compute nodes. We discussed how to tune the task granularity to optimize tradeoff between scheduling overhead and workload balance in DryadLINQ CTP and Hadoop in a recent publication [5]. We use two essentially pleasingly parallel applications for benchmark: SWG [6] for pairwise distance calculation of gene sequence alignment and Cap3 [41,42] for gene sequence assembly applications. Figures 39 and 40 show that DryadLINQ (2009) and Hadoop do not have much of a performance difference for these applications.

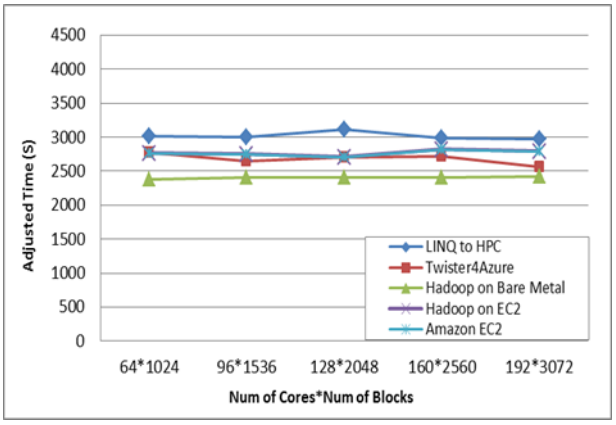


Figure 39: SWG with LINQ to HPC, Hadoop, Hadoop on EC2, Amazon EC2 and TwisterAzure

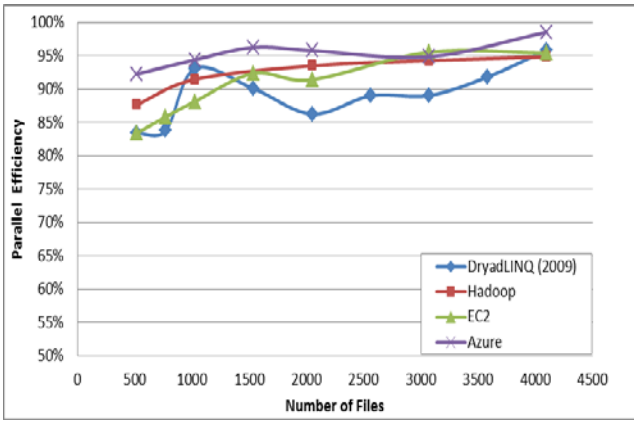


Figure 40: Cap3 with DryadLINQ (2009), Hadoop, EC2 and Azure

We benchmarked SWG on the latest Dryad release LINQ to HPC and compared it with Hadoop, Hadoop on Amazon EC2, Amazon ElasticMapReduce and Twister4Azure iterative MapReduce on Microsoft Azure Cloud. Figure 39 shows the execution time of SWG, adjusted for the sequential performance difference of SWG between the different environments, to perform up to 120 million sequence alignments. High-CPU-Extra-Large instances were used for the Amazon EC2 testing and 128 small Azure instances were used for the Azure testing. Hadoop bare metal testing was performed on a 16 node iDataPlex cluster [Appendix D], while LINQ to HPC tests were performed on Tempest cluster [Appendix B].

Figure 40 shows the parallel efficiency charts of Cap3 sequence assembly using a replicated set of FASTA formatted data files, each file containing 458 reads, with the largest test case assembling more than 1.8

million reads. A 16 node IBM iDataPlex cluster [Appendix D] was used for Cap3 DryadLINQ (2009) and Cap3 Hadoop with Microsoft Window HPC Server 2008, service Pack 1 - 64 bit and Red Hat Enterprise Linux Server release 5.3 -64 bit respectively as the operating system. 16 High-CPU-Extra-Large instances were used for the EC2 testing and 128 small Azure instances were used for the Azure Cap3 testing. The results suggest that pleasingly parallel applications work well on Cloud with variations only in detail of runtime environments.

8.2.2 Relational Dataset Approach to Iterative Algorithms

DryadLINQ and Pig are high-level programming interfaces that allow developers to express the data flow of their applications with relational queries on DryadLINQ and Hadoop respectively. The performance difference between DryadLINQ and Pig for relational data processing lies in the underlying DAG and MapReduce execution model. The rigid and flat data processing model of MapReduce requires all input be output of the previous stage, which prevents MapReduce from efficiently implementing *join* that processes multiple, heterogeneous datasets. For example, the classic implementation as the *join* step in MapReduce PageRank spawns a very large number of Map and Reduce tasks during processing. Further optimization of MapReduce PageRank, such as map-side *join*, requires developers to have sophisticated knowledge of the web graph structure. Figure 41 shows the performance of PageRank implemented with different *join* approaches with DryadLINQ, Hadoop, and Twister. DryadLINQ is faster than Hadoop by a factor of 2 to 3 for Pagerank. Twister Pagerank performs better than Hadoop Pagerank by a factor of 10. DryadLINQ has the lower performance than Twister because the execution pipeline of chained DryadLINQ queries break in each iteration. DryadLINQ PageRank implemented with reduce side join approach can maintain the execution pipeline, but it has a huge number of small tasks which cause significant network traffic.

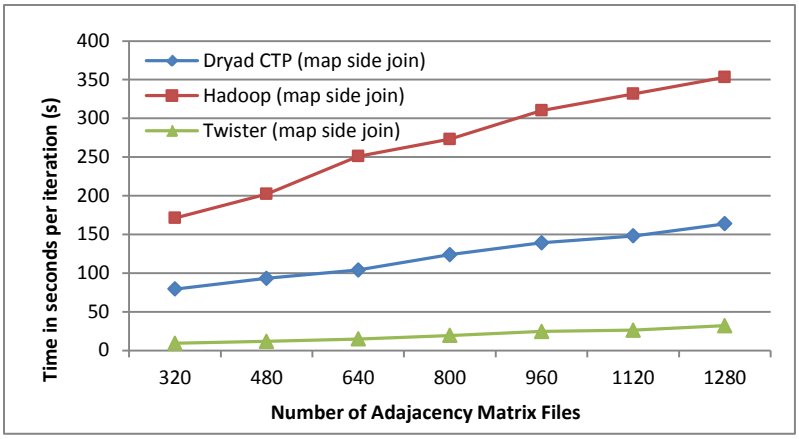


Figure 41: Time in seconds per iteration of PageRank with DryadLINQ CTP, Hadoop and Twister

8.2.3 Iterative MapReduce

New parallel programming models such as MapReduce and Hadoop were successful in processing massive datasets in a distributed environment. Data locality can be implicitly integrated into the runtime platform, as the same set of nodes is used for both computation and storage, which instantiates the paradigm of 'moving the computation to data'. MapReduce works well for pleasingly parallel applications. However, the MapReduce framework has Map only or Map and Reduce phases with disk access, which is not suitable for supporting more complicated data analysis or data mining that typically has many iterations.

Intel's RMS (recognition, mining, and synthesis) taxonomy [43] offers a way to describe a class of emerging applications. The technology underlying these applications is likely to have broad applicability, ranging across computer vision, rendering, physical simulation, (financial) analysis, and data mining. There are

common computing kernels at the core of these applications, which require iterative solvers and basic matrix primitives.

These observations suggest that Iterative MapReduce will be an important runtime to a spectrum of scientific, industrial, and societal applications, and as the kernel framework for large-scale data processing. Open source Java Twister [44,45] and Twister4Azure [46,47] have been released as Iterative MapReduce frameworks. Twister interpolates between MPI and MapReduce and, suitably configured, can mimic their characteristics, and, more interestingly, can be positioned as a programming model that has the performance of MPI and the fault tolerance and dynamic flexibility of the original MapReduce.

DryadLINQ can build up pipelines to accelerate the execution of Iterative MapReduce jobs by using late evaluation, asynchronous vertices processing, TCP pipe, and shared memory FIFO technologies. In addition, DryadLINQ can maintain the data locality for processing vertices during different computation stages of DryadLINQ jobs [4]. Therefore, it is able to reuse static data for multiple iterations in DryadLINQ [4]. However, a constraint of DryadLINQ in support of iterative MapReduce applications is that there is no explicit evaluation stage within the chained DryadLINQ queries. This is because once a query is explicitly evaluated and the output is retrieved to user program, the DryadLINQ runtime disposes all the intermediate data stored in memory. Hadoop cannot pipeline the execution of multiple jobs, as it needs to materialize the output of MapReduce jobs into HDFS.

Figures 41 and 42 show overhead of the different runtimes for Pagerank and Kmeans algorithms. The use of file system based communication mechanisms and the loading of static input data at each iteration as in Hadoop and in each unrolled loop as in DryadLINQ resulted considerably higher overheads compared to Twister and MPI. Although DAG supports dataflow though vertices, iterations defined via graphs is more like workflow. **DAG is not a suitable parallel runtime model for scientific computation where there're fine grained synchronizations between iterations** that is directly supported by Iterative MapReduce. The performance comparison of Hadoop, DryadLINQ and Twister suggest that **neither Hadoop nor DryadLINQ support Iterative MapReduce properly**.

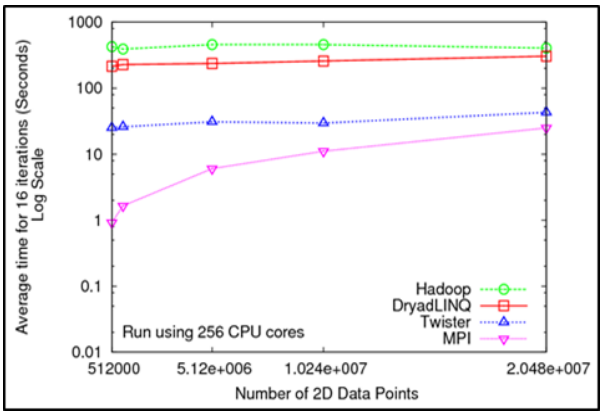


Figure 42: Performance of k-means implemented with Hadoop, DryadLINQ (2009), Twister and MPI

8.3 Evaluation and Findings

We analyze the key features of DryadLINQ and Hadoop, especially those in which DryadLINQ outperforms Hadoop in programming interface, task scheduling, performance, and applications. We also studied performance-related issues between DryadLINQ CTP and Hadoop. A major factor that leads to performance differences between DryadLINQ and Hadoop is the underling execution model of DAG that is more flexible and expressive than that of MapReduce. Other important features that contribute to performance differences

between DryadLINQ and Hadoop include fast communication, data-locality-aware scheduling, and pipelining between jobs. In summary, we make the following observations when comparing DryadLINQ CTP with Hadoop:

l) **DryadLINQ provides a data model and better language support by interfacing with .NET and LINQ**;
m) **DryadLINQ performs better than Pig when processing relational queries and supports a relational dataset approach to Iterative algorithms;**
n) **DryadLINQ supports advanced inter-task communication technologies such as files, TCP Pipe, and shared-memory FIFO. Hadoop transfers intermediate data via files and http;**
o) **DryadLINQ can maintain data locality at both Map and Reduce phases while Hadoop only supports Map-side data locality;**
p) **DryadLINQ supports pipelining execution stages for high performance but Hadoop doesn't;**
q) **DryadLINQ provides a rich set of distributed group aggregation strategies to reduce data movement, but Hadoop has limited support;**
r) **Neither Hadoop nor DryadLINQ support Iterative MapReduce properly.**

Acknowledgements

This work is partially funded by Microsoft. We want to thank John Naab and Ryan Hartman for system support of the STORM, TEMPEST, and MADRID HPC clusters, which were critical for our experiments. Second, we thank Thilina Gunarathne and Stephen Wu for their generosity in sharing the SWG application and data for our task scheduling analysis. We would also like to thank Ratul Bhawal and Pradnya Kakodkar, two Master's students who enrolled in Professor Qiu's B534 course in Spring 2011 for their contributions to this report.

References

[1] G. Bell, T. Hey, and A. Szalay, "Beyond the data deluge," Science, vol. 323, no. 5919, pp. 1297-1298, 2009
[2] Jaliya Ekanayake, Thilina Gunarathne, Judy Qiu, Geoffrey Fox, Scott Beason, Jong Youl Choi, Yang Ruan, Seung-Hee Bae, Hui Li. *Applicability of DryadLINQ to Scientific Applications*, Technical Report. SALSA Group, Indiana University. October 16, 2009.
[3] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly. (2007). *Dryad: distributed data-parallel programs from sequential building blocks*. SIGOPS Oper. Syst. Rev. 41(3): 59-72.
[4] Yu, Y., M. Isard, Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P.K., J., Currey. (2008). *DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language*. Symposium on Operating System Design and Implementation (OSDI). San Diego, CA.
[5] Hui Li, Yang Ruan, Yuduo Zhou, Judy Qiu and Geoffrey Fox, Design Patterns for Scientific Applications in DryadLINQ CTP, to appear in Proceedings of The Second International Workshop on Data Intensive Computing in the Clouds (DataCloud-2) 2011, The International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), Seattle, WA, November 12-18, 2011
[6] Gotoh, O. (1982). *An improved algorithm for matching biological sequences*. Journal of Molecular Biology 162: 705-708.
[7] *DryadLINQ and DSC Programmers Guilde*. Microsoft Research. 2011
[8] Seung-Hee Bae, Jong Youl Choi, Judy Qiu, Geoffrey C. Fox,. (2010). *Dimension reduction and visualization of large high-dimensional data via interpolation*. Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. Chicago, Illinois, ACM: 203-214.
[9] Batzer MA and Deininger PL (2002). *Alu repeats and human genomic diversity*. Nature Reviews Genetics 3(5): 370-379.
[10] *JAligner*. Retrieved December, 2009, from http://jaligner.sourceforge.net.
[11] Smith, T. F. and M. S. Waterman (1981). *Identification of common molecular subsequences*. Journal of Molecular Biology 147(1): 195-197.
[12] Johnsson, S. L., T. Harris, K. K. Mathur. 1989, *Matrix Multiplication on the connection machine*. Proceedings of the 1989 ACM/IEEE conference on Supercomputing. Reno, Nevada, United States, ACM.

[13] Geoffrey Fox, Tony Hey and Steve Otto, *Matrix algorithms on a hypercube I: Matrix multiplication*, parallel computing, pp. 17-31, 1987.
[14] Fox, G. C., *What Have We Learnt from Using Real Parallel Machines to Solve Real Problems*. Third Conference on Hypercube Concurrent Computers and Applications. G. C. Fox, ACM Press. 2: 897-955. 1988.
[15] Daan Leijen and Judd Hall (2007, October). *Parallel Performance: Optimize Managed Code For Multi-Core Machinesl*. Retrieved November 26, 2010, from http://msdn.microsoft.com/en-us/magazine/cc163340.aspx.
[16] Jaliya Ekanayake (2010). *Architecture and Performance of Runtime Environments for Data Intensive Scalable Computing*. School of Informatics and Computing. Bloomington, Indiana University.
[17] Argonne National Laboratory. *MPI Message Passing Interface*. Retrieved November 27, 2010, from http://www-unix.mcs.anl.gov/mpi
[18] Apache Hadoop. Retrieved November 27, 2010, from http://hadoop.apache.org/.
[19] Yingyi Bu, Bill Howe, Magdalena Balazinska, Michael D. Ernst,. (2010). *HaLoop: Efficient Iterative Data Processing on Large Clusters*. The 36th International Conference on Very Large Data Bases. Singapore, VLDB Endowment. 3.
[20] J.Ekanayake, H.Li, B.Zhang, T.Gunarathne, S.Bae, J.Qiu, G.Fox. *Twister: A Runtime for iterative MapReduce*. Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010. Chicago, Illinois, ACM.
[21] Ekanayake, J., S. Pallickara, Shrideep, Fox, Geoffrey. *MapReduce for Data Intensive Scientific Analyses*. Fourth IEEE International Conference on eScience, IEEE Press: 277-284. 2008.
[22] ClueWeb Data: http://boston.lti.cs.cmu.edu/Data/clueweb09/
[23] PageRank wiki: http://en.wikipedia.org/wiki/PageRank
[24] Y. Yu, M. Isard, D.Fetterly, M. Budiu, U.Erlingsson, P.K. Gunda, J.Currey, F.McSherry, and K. Achan. Technical Report MSR-TR-2008-74, Microsoft.
[25] Yu, Y., P. K. Gunda, M. Isard. (2009). *Distributed aggregation for data-parallel computing: interfaces and implementations*. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. Big Sky, Montana, USA, ACM: 247-260.
[26] BSP, Bulk Synchronous Parallel http://en.wikipedia.org/wiki/Bulk_Synchronous_Parallel
[27] Yangdong Wang, Xianyu Que. (2011). Hadoop Acceleration Through Network Leviated Merge SC11
[28] Dawei jiang, Beng Chin Ooi, Lei Shi (2010). The Performance of MapReduce: An In-depth Study VLDB 2010.
[29] Tyson Condie, Neil Conway, Peter Alvaro. (2010). Online Aggregation and Continuous Query Support in MapReduce SIGMOD'10
[30] Padmashree Ravindra, Seokyong Hong, Hyeong Sik Kim. (2011). Efficient Processing of RDF Graph Pattern Matching on MapReduce Platforms DataCloud-SC-11
[31] Jens Dittrich, Jorge Arnulfo Quiane Ruiz. (2010). Hadoop++: Making a Yellow Elephant Run Like Cheetah VLDB 2010/ PVLDB
[32] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac. (2010) A Comparison of Join Algorithms for Log Processing in MapReduce. SIGMOD'10
[33] Yunhong Gu, Robert Grossman, Sector and Sphere: The Design and Implementation of a High Performance Data Cloud, Theme Issue of the Philosophical Transactions of the Royal Society A: Crossing Boundaries: Computational Science, E-Science and Global E-Infrastructure, 28 June 2009 vol. 367 no. 1897 2429-2445
[34] Jeffrey Dean, Sanjay Ghemawat. (2008). MapReduce: simplified data processing on large clusters
[35] Apache Pig Latin. Available from http://pig.apache.org/docs/r0.7.0/piglatin_ref2.html
[36] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. 2011. YSmart: Yet Another SQL-to-MapReduce Translator. In Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS '11). IEEE Computer Society, Washington, DC, USA, 25-36. DOI=10.1109/ICDCS.2011.26 http://dx.doi.org/10.1109/ICDCS.2011.26
[37] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online. In Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI'10). USENIX Association, Berkeley, CA, USA, 21-21
[38] JIRA ticket https://issues.apache.org/jira/browse/MAPREDUCE-225
[39] JIRA ticket https://issues.apache.org/jira/browse/MAPREDUCE-2038
[40] Judy Qiu, Generalizing MapReduce as a Unified Cloud and HPC Runtime, Invited Talk Abstract, at Petascale Data Analytics: Challenges, and Opportunities (PDAC-11) Workshop at SC11 Seattle November 14, 2011. Available at http://grids.ucs.indiana.edu/ptliupages/publications/pdac22j-qiu.pdf
[41] X. Huang, and A. Madan, "CAP3: A DNA sequence assembly program," Genome Res, vol. 9, no. 9, pp. 868-77, 1999.
[42] Thilina Gunarathne, Tak-Lon Wu, Judy Qiu et al., "Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications," in Proceedings of the Emerging Computational Methods for the Life Sciences Workshop of ACM HPDC 2010 conference, Chicago, Illinois, 2010.
[43] Dubey, Pradeep. A Platform 2015 Model: Recognition, Mining and Synthesis Moves Computers to the Era of Tera. Compute-Intensive, Highly Parallel Applications and Uses. Volume 09 Issue 02. ISSN 1535-864X. February 2005.
[44] SALSA Group. Iterative MapReduce. Twister Home Page. Available from http://www.iterativemapreduce.org/

[45] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, Geoffrey Fox Twister: A Runtime for Iterative MapReduce, in Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference, Chicago, Illinois, June 20-25, 2010
[46] SALSA Group. Twister4Azure. Home Page. Available from http://salsahpc.indiana.edu/twister4azure/
[47] Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, Judy Qiu, Portable Parallel Programming on Cloud and HPC: Scientific Applications of Twister4Azure, in Proceedings of Fourth IEEE International Conference on Utility and Cloud Computing (UCC 2011), Melbourne, Australia, December 5-8, 2011

Appendix

Appendix A

STORM Cluster

8-node inhomogeneous HPC R2 cluster

	STORM	STORM-CN01	STORM-CN02	STORM-CN03	STORM-CN04	STORM-CN05	STORM-CN06	STORM-CN07
CPU	AMD 2356	AMD 2356	AMD 2356	AMD 2356	AMD 8356	AMD 8356	Intel E7450	AMD 8435
Cores	8	8	8	8	16	16	24	24
Memory	16G	16G	16G	16G	16G	16G	48G	32G
Mem/Core	2G	2G	2G	2G	1G	1G	2G	1.33G
NIC Enterprise)	N/a	N/a	N/a	N/a	N/a	N/a	N/a	N/a
NIC (Private)	BCM5708C	BCM5708C	BCM5708C	BCM5708C	BCM5708C	BCM5708C	BCM5708C	BCM5708C

Appendix B

TEMPEST Cluster

33-node homogeneous HPC R2 cluster

	TEMPEST	TEMPEST-CNXX
CPU	Intel E7450	Intel E7450
Cores	24	24
Memory	24.0GB	50.0 GB
Mem/Core	1 GB	2 GB
NIC (Enterprise)	HP NC 360T	n/a
NIC (Private)	HP NC373i	HP NC373i
NIC (Application)	Mellanox IPolB	Mellanox IPoIB

Appendix C

MADRID Cluster

9-node homogeneous HPC cluster

	MADRID-HEADNODE	MADRID-10X
CPU	AMD Opteron 2356 2.29GHz	AMD Opteron 8356 2.30GHz
Cores	8	16

Memory	8GB	16GB
Memory/Core	1GB	1GB
NIC	BCM5708C	BCM5708C

Appendix D

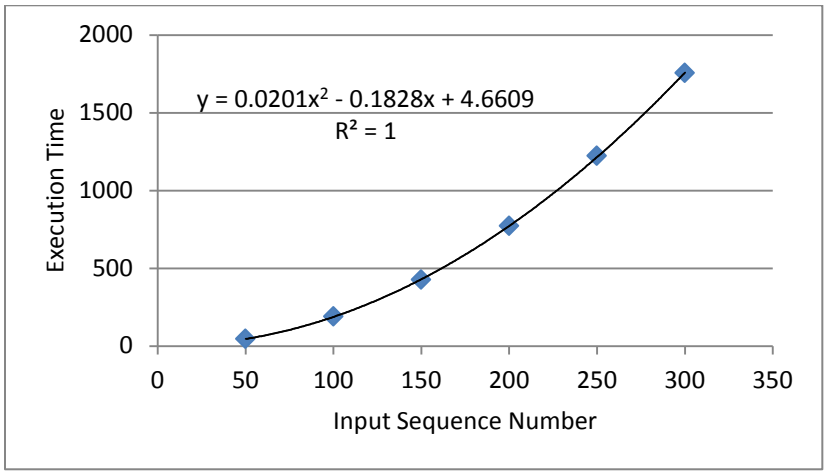
iDataPlex Cluster

32-node homogeneous HPC cluster

	iDP node
CPU	Intel(R) Xeon(R) CPU L5420 2.50GHz
Cores	8
Memory	32GB
Memory/Core	4GB
NIC	Giga bit Ethernet

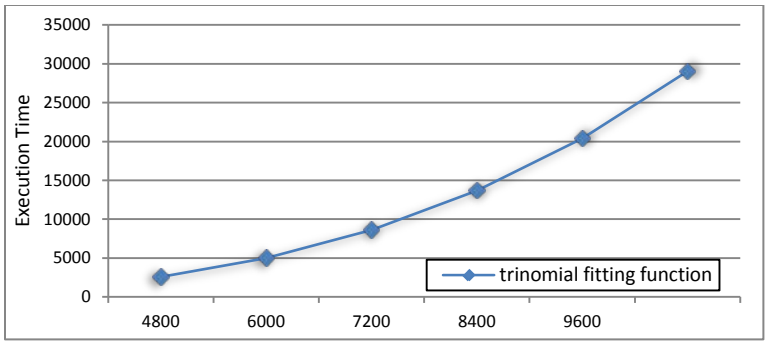
Appendix E

Binomial fitting function for sequential SWG jobs



Appendix F

Trinomial fitting chart for sequential matrix multiplication jobs



Appendix G

Tables mentioned in the report.

Table 1: Execution Time for Various SWG Partitions on Tempest Cluster

Partition Number	31	62	93	124	248	372	496	620	744	992
Test 1	1324.54	1345.41	1369.01	1379.01	1761.09	1564.79	1866.14	2280.37	2677.57	3578.50
Test 2	1317.45	1356.68	1386.09	1364.43	1735.46	1588.92	1843.70	2286.76	2736.07	3552.58
Test 3	1322.01	1348.89	1368.74	1384.87	1730.47	1568.59	1857.00	2258.25	2709.61	3568.21
Average	**1321.33**	**1350.33**	**1374.61**	**1376.10**	**1742.34**	**1574.10**	**1855.61**	**2275.13**	**2707.75**	**3566.43**

Table 2: Execution Time for Skewed and Randomized Data

Std. Dev.	1	50	100	150	200	250
Skewed	2582	3076	3198	3396	3878	4488
Randomized	2582	2489	2458	2413	2498	2622

Table 3: Average Execution Time of Tempest

No. of Nodes	Input length				
	5000	7500	10000	12500	15000
1	13854.71	31169.03	55734.36	89500.57	131857.4
32	550.255	1096.925	1927.436	3010.681	4400.221
Parallel Efficiency	**81.22%**	**91.66%**	**93.28%**	**95.90%**	**96.66%**

Table 4: Execution Time and Speed-up for SWG on Tempest with Varied Size of Compute Nodes

Num. of Nodes	**1**	**2**	**4**	**8**	**16**	**31**
Average Execution Time	55734.36	27979.78	14068.49	7099.70	3598.99	1927.44
Relative Speed-up	1	1.99	3.96	7.85	15.49	28.92

Table 5: Blocks Assigned to Each Compute Node

Node Name	**Partition Number**					
	6	**12**	**24**	**48**	**96**	**192**
STORM-CN01	687	345	502	502	549	563
STORM-CN02	681	683	510	423	547	575
STORM-CN03	685	684	508	511	548	571
STORM-CN04	688	685	689	775	599	669
STORM-CN05	667	681	685	679	592	635
STORM-CN06	688	1018	1202	1206	1261	1083

Table 6 Characteristic of PageRank input data

No of am files	File size	No of web pages	No of links	Ave out-degree
1280	9.7GB	49.5million	1.40 billion	29.3

Table 7 DRP of different number of AM files of three aggregation approaches

Input size	hash aggregation	partial aggregation	hierarchical aggregation
320 files 2.3G	1: 306	1:6.6:306	1:6.6:2.1:306
640 files 5.1G	1: 389	1:7.9:389	1:7.9:2.3:389
1280 files 9.7G	1: 587	1:11.8:587	1:11.8:3.7:587

Table 8: Job turnaround time for different PageRank implementations

Parallel Implementations	**Average job turnaround time for 3 runs**
MPI PageRank on 32 node Linux Cluster (8 cores/node)	101 sec
Twister PageRank on 32 node Linux Cluster (8 cores/node)	271 sec

Haloop PageRank on 32 node Linux Cluster (8 cores/node)	1954 sec
Dryad PageRank on 32 node HPC Cluster (24 cores/node)	1905 sec
Hadoop PageRank on 32 node Linux Cluster (8 cores/node)	3260 sec
Sequential Implementations	
C PageRank on Linux OS (use 1 core)	831 sec
Java PageRank on Linux OS (use 1 core)	3360 sec
C# PageRank on Windows Server (use 1 core)	8316 sec

Table 9: Parallel Efficiency of DryadLINQ CTP and DryadLINQ 2009 on same input data

DryadLINQ CTP					
# of Nodes	**Input size**				
	5000	**7500**	**10000**	**12500**	**15000**
7 Nodes	2051	4540	8070	12992	18923
1 Node	13855	31169	55734	89501	131857
Parallel Efficiency	**96.50%**	**98.07%**	**98.66%**	**98.41%**	**99.54%**
DryadLINQ 2009					
# of Nodes	**Input size**				
	5000	**7500**	**10000**	**12500**	**15000**
7	2523	5365	9348	14310	20615
1	17010	36702	64141	98709	142455
Parallel Efficiency	**96.31%**	**97.73%**	**98.02%**	**98.54%**	**98.72%**

Table 10: Execution Time for SWG with Data Partitions

Number of Partitions	**6**	**12**	**24**	**36**	**48**	**60**	**72**	**84**	**96**
Execution Time 1	1105	1135	928	981	952	1026	979	1178	1103

Execution Time 2	1026	1063	868	973	933	1047	968	1171	1146
Execution Time 3	1030	1049	861	896	918	1046	996	1185	1134
Execution Time 4	1047	1060	844	970	923	1041	985	1160	1106
Average Time	**1052**	**1076**	**875**	**955**	**931**	**1040**	**982**	**1173**	**1122**
Speed-Up	79.8	78.0	95.9	87.9	90.1	80.7	85.5	71.5	74.8

Appendix H

Implementation of different Matrix Multiplication algorithms

1)The Row Partition Algorithm

```
results = aMatrixFiles.Select(aFilePath => rowsXcolumns(aFilePath, bMatrixFilePath));
```

2) The Row Column Partition Algorithm

```
string[] aMatrixPartitionsFiles = splitInputFile(aMatrixPath, numIterations);
string[] bMatrixPartitionsFiles = splitInputFile(bMatrixPath, numComputeNodes);
DistributedQuery<matrixPartition> bMatrixPartitions =
bMatrixPartitionsFiles.AsDistributed().HashPartition(x => x, numComputeNodes).
Select(file => buildMatrixPartitionFromFile(file));

for (int iterations = 0; iterations<numIterations;iterations++)
{
  DistributedQuery<matrixBlock> outputs = bMatrixPartitions.ApplyPerPartition(bSubPartitions =>
bSubPartitions.Select(bPartition =>
aPartitionMultiplybPartition(aMatrixPartitionsFiles[iterations], bPartition)));
}
```

3.1)The Fox-Hey Algorithm

```
string[] aPartitionsFile = splitInputFile(aMatrixPath, nProcesses);
string[] bPartitionsFile = splitInputFile(bMatrixPath, nProcesses);
IEnumerable<aMatrixCMatrixInput> inputAC = buildBlocksInputOfAMatrixCMatrix(rowNum, colNum, 0,
nProcesses);
DistributedQuery<aMatrixCMatrixInput> inputACquery = inputAC.AsDistributed().HashPartition(x =>
x, nProcesses * nProcesses);
DistributedQuery<bMatrixInput> inputBquery = bPartitionsFile.AsDistributed().Select(x =>
buildBMatrixInput(x, 0, nProcesses)).SelectMany(x => x);

for (int iterations = 0; iterations < nProcesses; iterations++){
  inputACquery = inputACquery.ApplyPerPartition(sublist => sublist.Select(acBlock =>
  acBlock.updateAMatrixBlockFromFile(aPartitionsFile[acBlock.ci], iterations,nProcesses)));
  inputACquery = inputACquery.Join(inputBquery, x => x.key, y => y.key, (x, y) =>
  x.taskMultiplyBBlock(y.bMatrix));
  inputBquery = inputBquery.Select(x => x.updateIndex(nProcesses));
}
```

3.2) Maintain intermediate data and status in iterative MapReduce using DryadLINQ

```
DistributedQuery<Type> inputData = inputObjects.AsDistributed();
inputData = inputData.Select(data=>update(data));
```

Appendix I

Implementation of the Matrix Multiplication utilizing multi-core technology.

```
while (localRows.MoveNext())
{
        double[] row_result = newdouble[colNum];
        for (int i = 0; i < colNum; i++)
        {
            double tmp = 0.0;
```

```
            for (int j = 0; j < rowNum; j++)
                 tmp += localRows.Current.row[j] * columns[i][j];
             row_result[i] = tmp;
        }
         yieldreturn row_result;
}
```

1) The Parallel.For version of Matrix Multiplication

```
while (localRows.MoveNext())
{
        blockWrapper rows_result = new blockWrapper(size,colNum,rowNum);
        Parallel.For(0, size, (int k) =>
        {
            for (int i = 0; i < colNum; i++)
            {
                 double tmp = 0.0;
                 for (int j = 0; j < rowNum; j++)
                      tmp += localRows.Current.rows[k * rowNum + j] * columns[i][j];
                rows_result.block[k * colNum + i] = tmp;
            }
        });
        yieldreturn rows_result;
}
```

2) The ThreadPool version of Matrix Multiplication

```
while (localRows.MoveNext())
{
   blockWrapper rows_result = new blockWrapper(size, rowNum, colNum);
   ManualResetEvent signal = new ManualResetEvent(false);
   for (int n = 0; n < size; n++)
   {
        int k = n;
        ThreadPool.QueueUserWorkItem(_ =>
        {
            for (int i = 0; i < colNum; i++)
            {
                 double tmp = 0;
                 for (int j = 0; j < rowNum; j++)
                          tmp += localRows.Current.rows[k * rowNum + j] * columns[i][j];
                rows_result.block[k * colNum + i] = tmp;
            }
            if (Interlocked.Decrement(ref iters) == 0)
                signal.Set();
        });
   }
   signal.WaitOne();
   yieldreturn rows_result;
}
```

3) The PLINQ version of Matrix Multiplication

```
while (localRows.MoveNext())
{
   double[][] rowsInput = initRows(localRows.Current.block);
   IEnumerable<double[]> results = rowsInput.AsEnumerable().AsParallel().AsOrdered()
              .Select(x => oneRowMultiplyColumns(x, columns));
   blockWrapper outputResult = new blockWrapper(size,rowNum,colNum, results);
   yieldreturn outputResult;
}
```

Appendix J

PageRank implementation code sample.

GroupBy() and Join()

```
for (int i = 0; i < iterations; i++)
{
```

```
 newRanks = pages.Join(ranks, page => page.source, rank => rank.source,
  //join page objects with rank objects where they have the same source url
 (page, rank) => page.links.Select(dest =>newRank(dest, rank.value / (double)page.numLinks)))
  //calculate the partial rank value for each destination url to which the source url points
.SelectMany(list => list).GroupBy(rank => rank.source)
  //group partial rank objects by their url id across a cluster
.Select(group =>newRank(group.Key, group.Select(rank => rank.value).Sum() * 0.85 + 0.15 /
(double)_numUrls));
 //aggregate partial rank values for each url for final rank values
 ranks = newRanks;
}
```

HashPartition() and ApplyPerPartition()

```
for (int i = 0; i < _iteration; i++)
{
 newRanks = pages.Join(ranks, page => page.source, rank =>rank.source,
 // join page objects with rank objects where they have the same source url
 (page, rank) => page.links.Select(dest =>new Vertex(dest, rank.value / (double)page.numLinks)))
 //calculate the partial rank value for each destination url which the source url points to
 .SelectMany(list => list).HashPartition(record => record.source)
 //hash partition partial rank objects so that the objects with same url are sent to same node
 .ApplyPerPartition(list => list.GroupBy(record => record.source))
 //group partial rank objects by their url id on local machine
 .Select(group =>newRank(group.Key, group.Select(rank =>rank.value).Sum() * 0.85 + 0.15 /
(double)_numUrls));
 //aggregate grouped partial rank values for each url for final rank values
 ranks = newRanks.Execute();
}
```

Hierarchical Aggregation with User Defined Aggregation function

```
DistributedQuery<amPartition> webgraphPartitions =
Directory.GetFiles(inputDir).AsDistributed().Select(fileName =>
buildWebGraphPartition(fileName));
//construct partial rank values using adjacency matrixfiles stored in inputDir
for (int i = 0; i < numIteration; i++)
{
        DistributedQuery<double[]> partialRVTs = null;
        partialRVTs = webgraphPartitions.ApplyPerPartition(subWebGraphPartitions =>
        calculateMultipleWebgraphPartitionsWithPLINQ(subWebGraphPartitions, rankValueTable,
numUrls));
       //calculate partial rank values with user-defined function
        rankValueTable = mergeResults(partialRVTs);
       //merge calculated parital rank values with user-defined aggregation function
        //synchronized step to merge all partial rank value tables
}
```

GroupAndAggregate

```
for (int i = 0; i < numIteration; i++)
{
newRanks = pages.Join(ranks, page => page.source, rank =>rank.source,(page, rank) =>
// join page objects with rank objects where they have the same source url
page.links.Select(targetPage =>newRank(targetPage, rank.value / (double)page.numLinks)))
// calculate the partial rank value for each destination url which the source url points to
.SelectMany(list => list).GroupAndAggregate(partialRank =>partialRank.source, g =>
newRank(g.Key, g.Sum(x => x.value)*0.85+0.15 / (double)numUrls));
group calculated partial rank values by the url id and aggregate the grouped partial ranks values
ranks = newRanks;
}
```

Two steps of ApplyPerPartition

```
for (int i = 0; i < numIteration; i++)
{
newRanks = pages.Join(ranks, page => page.source, rank =>rank.source,
// join page objects with rank objects where they have the same source url
(page, rank) => page.links.Select(dest =>newVertex(dest, rank.value / (double)page.numLinks)))
.SelectMany(list => list)
// calculate the partial rank value for each destination url which the source url points to
```

```
.ApplyPerPartition(subGroup => subGroup.GroupBy(e => e.source))
.Select(subGroup =>new Tuple<int, double>(subGroup.Key,subGroup.Select(rank =>rank.value).Sum()))
// group the partial rank objects by their url id on local machine
.HashPartition(e => e.Item1)
.ApplyPerPartition(subGroup => subGroup.GroupBy(e => e.Item1))
// group the partial rank objects by their url id over the cluster
.Select(subGroup =>newRank(subGroup.Key, subGroup.Select(e => e.Item2).Sum() * 0.85 + 0.15 /
(double)_numUrls));
// aggregate the grouped partial rank value for each url for final rank values
ranks = newRanks.Execute();
}
```

Appendix K

Table 11. Applications implemented with different DryadLINQ versions by SALSA Group

DryadLINQ versions	**Applications**	**Comments**
DryadLINQ (2009. 11)	Kmeans, SWG, Cap3, Blast, MDS, GTM, High Energy Physics, PhyloD	
DryadLINQ CTP	SWG, Matrix Multiplication, PageRank	Matrix Multiplication and PageRank were implemented with 3 approaches respectively.
LINQtoHPC	Matrix Multiplication Fox algorithm	

Table 12. Different programming models supported by Dryad and Hadoop

Programming model		**DryadLINQ applications**	**Hadoop applications**
Pleasingly Parallel Programming		SWG, Cap3, Blast, High Energy Physics, PhyloD	SWG, Cap3, Blast, High Energy Physics,
Relational Dataset Approach to Iterative Algorithms	Parallel Join	Matrix Multiply Fox algorithm, PageRank	PageRank (map side join)
	Distributed Aggregation	Kmeans, PageRank (aggregation tree)	Kmeans, PageRank (within map aggregation, local combiner)
Iterative MapReduce		Kmeans, Matrix Multiply Fox, PageRank	Kmeans, PageRank