DRYADLINQ CTP EVALUATION

Performance of Key Features and Interfaces in DryadLINQ CTP

Hui Li, Ruan Yang, Yuduo Zhou, Judy Qiu
July 31, 2011

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

Table of Contents

1 Introduction

Applying high-level parallel runtimes to data intensive applications is becoming increasingly common [1]. Systems such as, MapReduce and Hadoop allow developers to write applications that distribute tasks to remote environment containing the data, which follows the paradigm "moving the computation to data". The MapReduce programming model has been applied to a wide range of applications and attracts a lot of enthusiasm among distributed computing communities due to its easiness and efficiency to process large scale distributed data.

However, its rigid and flat data processing paradigm does not directly support relational operations having multiple related inhomogeneous input data streams; this limitation causes difficulties and inefficiency when using Map-Reduce to simulate relational operations, such as join, which is substantially common in database systems. For instance, the classic implementation of PageRank is notably inefficient due to the join simulations caused by a lot of network traffic in the computation. Further optimization of PageRank requires developers to have sophisticated knowledge on web graph structure.

Dryad [2], a general-purpose runtime to support data intensive applications on a Windows platform, is positioned between MapReduce and database systems, which addresses some of the limitations of MapReduce systems. DryadLINQ [3] is the programming interface for Dryad and enables developers to write a wide range of data-parallel applications easily and efficiently. It automatically translates .NET written LINQ programs into distributed computations executing on top of the Dryad system. For some applications, writing DryadLINQ programs are as simple as writing sequential programs. The DryadLINQ and Dryad runtime optimize the job execution and dynamically make changes during the computation. This optimization is handled by the runtime but transparent to the users. As an example, when implementing PageRank with DryadLINQ GroupAndAggregate() operator, it dynamically constructs a partial aggregation tree based on data locality to reduce the number of intermediate records being transferred across the compute nodes during computation.

This report investigates key features and interfaces of the DryadLINQ CTP with a focus on their efficiency and scalability. By implementing three different scientific applications, which include Pairwise Alu sequence alignment, Matrix Multiplication, and PageRank with large and real-world data, the following were evident: 1) Task scheduling in the Dryad CTP performs better than the Dryad (2009.11), but its default scheduling plan does not fit well for some applications and hardware settings, 2) Porting multi-core technologies, such as PLINQ and TPL to DryadLINQ tasks can increase the system, and 3) The choice of distributed grouped aggregation approaches with DryadLINQ CTP has a substantial impact on the performance of data aggregation/reduction applications.

This report explores programming models of DryadLINQ CTP and can be applied to three diffeent types of classic scientific applications including pleasingly parallel, hybrid distributed and shared memory, and distributed grouped aggregation. In Smith Waterman – Gotoh algorithm (SWG) [13], a pleasingly parallel application consists of Map and Reduce steps. In Matrix Multiplication, a hybrid parallel programming model combines inter-node distributed memory with intra node shared memory parallelization. In PageRank, the usability and performance of three distributed grouped aggregation approaches are investigated. A further study of DryadLINQ CTP includes performance comparisons of distributed grouped aggregation with MPI [4], Hadoop [5], and Twister [6]. Also, an education component illustrates how Professor Judy Qiu integrates Dryad/DryadLINQ into class projects for computer science graduate students at Indiana University.

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 explains the applicability of DryadLINQ to education and training. Note: in the report, "Dryad/DryadLINQ CTP" refers to Dryad/DryadLINQ community technical preview released in 2010.12; "Dryad/DryadLINQ (2009.11)" refers to the version released in 2009.11.11;

"Dryad/DryadLINQ" refers to all Dryad/DryadLINQ versions. Experiments are conducted on three Windows HPC clusters – STORM, TEMPEST, and MADRID [Appendix A, B, and C], where STORM consists of heterogeneous multicore nodes and TEMPEST and MADRID are homogeneous production systems of 768 and 128 cores each.

2 Overview

Dryad, DryadLINQ, and the Distributed Storage Catalog (DSC) [7] are sets of technologies to 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 graph vertex is an independent instance of the data processing for a particular stage. Graph edges are channels transferred data between vertices. A DSC component works with the NTFS to provide the data management functionality, 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 representation and processing of data. DryadLINQ data objects are a collection of.NET type objects, which can be split into partitions and distributed across 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.

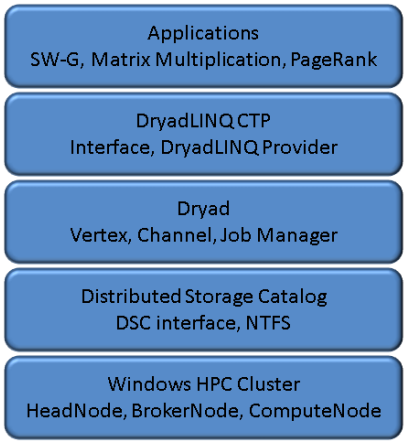


Figure1 Software Stack for DryadLINQ CTP

2.1 Task Scheduling

Task scheduling is a key feature of DryadLINQ CTP we investigate in this report. DryadLINQ provider translates LINQ queries into distributed computation and dispatch tasks to a cluster automatically. This process is handled by the runtime transparently from users. Dryad can handle fault tolerance and workload balance automatically to some extends. In addition, it enables developers to address workload issues based on the feature of their applications by tuning partition granularity with a unified data model and interface of DryadLINQ CTP. In later paragraphs, an argument is determined on why DryadLINQ CTP outperforms Dryad (2009.11) for scheduling inhomogeneous tasks on homogeneous resources and vice versa.

The above mechanism in DryadLINQ provides flexibility to developers for solving the workload balance issue. In batch job scheduling systems like PBS, programmers have to manually group/un-group (or split/combine) input/output data to control task granularity. Hadoop provides an interface that allows developers to define task granularity by indicating the size of input records in HDFS. This is an improvement, but it still requires developers to define the logic format of input data in HDFS. In contrast, DryadLINQ has a much simplified approach for data partition and access.

2.2 Parallel Programming Model

Dryad is designed to process coarse granularity tasks for large-scale distributed data. It schedules tasks to computing resources in the unit of compute nodes rather than cores. To make high utilization of multi-core resources of a HPC cluster for DryadLINQ jobs, one approach is to perform parallel computation on each node using PLINQ. The DryadLINQ provider can automatically transfer PLINQ query to parallel computation. The second approach is to apply the 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 the hybrid model, Dryad handles inter-node parallelism while PLINQ, TPL, and thread pool technologies deal with inner-node parallelism for multi-cores. This hybrid parallel programing model for Dryad/DryadLINQ has been proved successful and applied to data clustering applications, such as General Topography Mapping (GTM) interpolation and Multi-Dimensional Scaling (MDS) interpolation [8]. Most of the pleasingly parallel applications can be implemented with this model to increase the overall utilization of cluster resources.

2.3 Distributed Grouped Aggregation

The GROUP BY operator in parallel databases is often followed by aggregate functions. It 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. Sample applications of this pattern include sales data summarizations, log data analysis, and social network influence analysis.

MapReduce and SQL in databases are two programming models that can perform distributed grouped aggregation. MapReduce has been applied to process a wide range of flat distributed data. However, MapReduce is not efficient in processing relational operations, which have multiple inhomogeneous input data stream, such as JOIN. SQL queries provide relational operations of multiple inhomogeneous input data streams. However, a full-featured SQL database has extra overhead and constraints that prevent it from processing large scale input data.

DryadLINQ is between SQL and MapReduce, and it addresses some limitations of SQL and MapReduce. DryadLINQ provides developers with SQL-like queries to process efficient aggregation for single input data streams and multiple inhomogeneous input streams, but it does not have much overhead as SQL by eliminating some functionality of databases (transactions, data lockers, etc.). Further DryadLINQ can build an aggregation tree (some database also provides this kind of optimization) to decrease data transformation in the hash partitioning stage. This report investigates the usability and performance of distributed grouped aggregation in DryadLINQ CTP and compares it with other runtimes including: MPI, Hadoop, Haloop [9], and Twister for PageRank application.

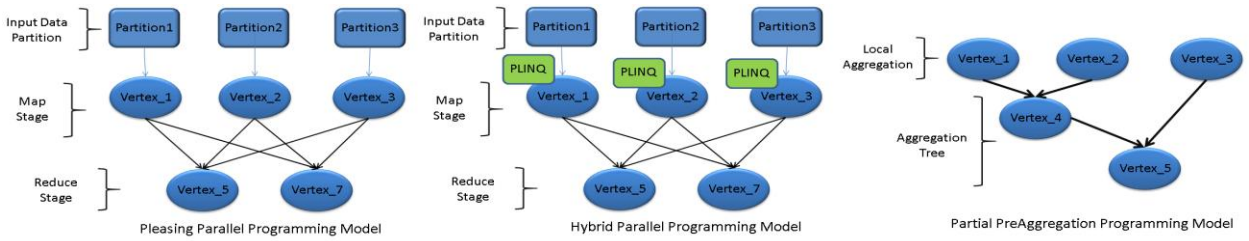


Figure 2: Three Programming Models for Scientific Applications in DryadLINQ CTP

3 Pleasingly Parallel Application in DryadLINQ CTP

3.1 Introduction

A pleasingly parallel application is the one for which no particular effort is needed to segment the problem into a very large number of parallel tasks, and there is neither essential dependency nor communication between those parallel tasks. The task scheduling mechanism and granularity play important roles on performance and are evaluated in Dryad CTP using 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 other applications.

Pairwise Alu Sequence Alignment Using Smith Waterman GOTOH

The Alu clustering problem [10][11] 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 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 problem in bioinformatics for the next few years, e.g. the automated protein family classification for a few millions of proteins predicted from large meta-genomics projects.

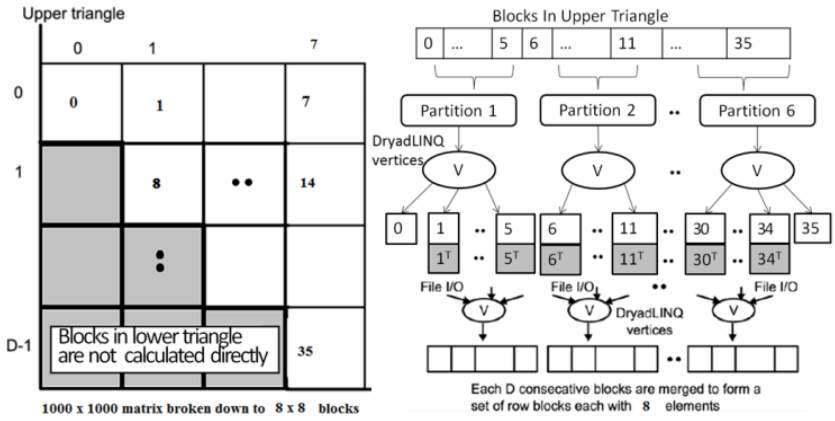


Figure 3: Task Decomposition (left) and the Dryad Vertex Hierarchy (right) of the DryadLINQ Implementation of SWG Pairwise Distance Calculation Application

An open source version, NAligner [12], of the Smith Waterman – Gotoh algorithm (SWG) [13] 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.

DryadLINQ Implementation

The SWG program runs in two steps. In the map stage, input data is divided into partitions being assigned to vertexes. A vertex calls external pair-wise distance calculation on each block and runs independently. In the reduce stage, this vertex starts a few merge threads to collect output from map stage and merge them into one file and then sends meta data of the file back to 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).

Furthermore, Dryad CTP divides the total workload of 36 blocks into 6 partitions, where each partition contains 6 blocks. After partitions being 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 files into one large file 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 that simply divides the input data into partitions. The task granularity was tuned by saving all SWG blocks into two-dimensional arrays and converting to distributed partitions with the *AsDistributedFromPartitions* operator.

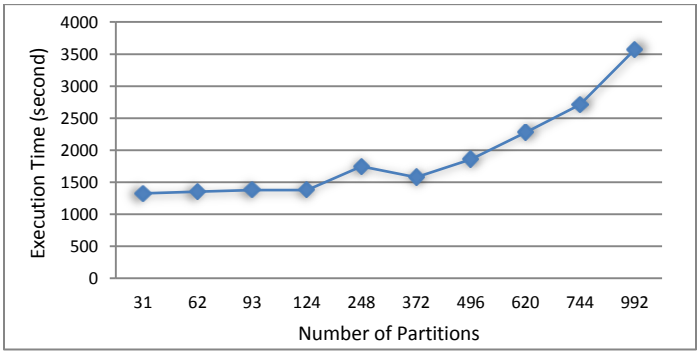


Figure 4: Execution Time for Various SWG Partitions
Executed on Tempest Cluster, with input of 10,000 sequences, and a 128×128 block matrix

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 10,000, which requires 100,000,000 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 400 with a standard deviation as 10, which gives essentially homogeneous distribution ensuring a good load balance. For a cluster of 32 compute nodes, the Dryad job manager takes one for its dedicated usage and leaves 31 nodes for the actual computation. As shown in Figure 4 and Table 1 (in Appendix F), 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.

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 [1]. We studied the scheduling issue in Dryad CTP using the same application.

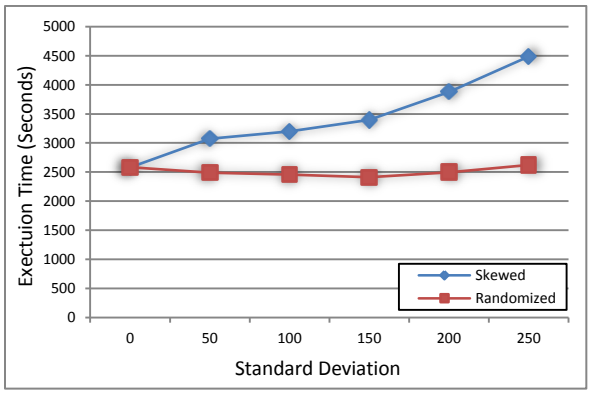


Figure 5: SWG Execution Time for Skewed and Randomized Distributed Input Data

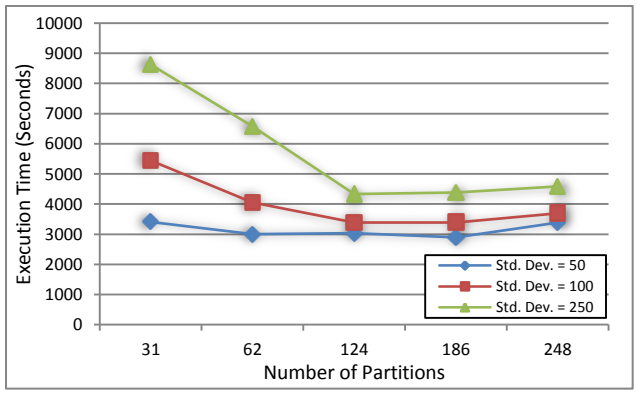


Figure 6: SWG Execution Time for Skewed Data with Different Partition Amount

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. As shown in Figure 5, randomized data showed better performance than skewed one. Similar results were presented in the Dryad (2009.11) report as well. Since sequences were sorted by length for a skewed sample, computation 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.

Dryad 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 overweighs gains from 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 if 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 Dryad 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 chooses one dedicated compute node for the graph manager which leaves only 6 nodes for computation. Since Dryad assigns each compute node with 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 TEMPEST cluster.

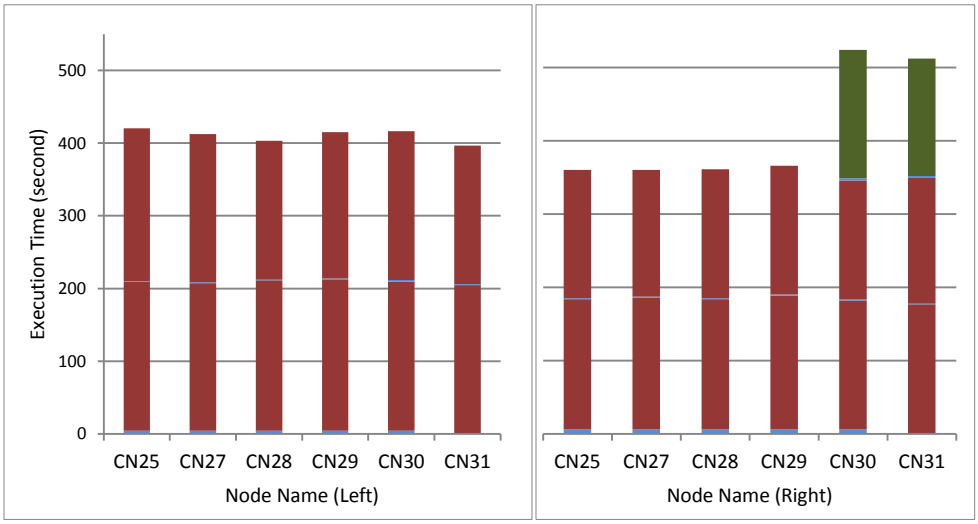


Figure 7: Mismatch between Partitions and Compute Nodes in Default Settings of Dryad CTP

Figure 7 (Left) is the execution timeline for customized 12 partitions and the right is default partitions by *AsDistributed()*. It is observed that data is divided into 14 partitions over 6 compute nodes, where 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 as that of compute nodes. Hadoop partitions input data into chunks, each of which has a default size of 64MB. Hadoop implements a high through put model for dynamic scheduling and is insensitive to load imbalance issues. Both Dryad and Hadoop provide interface that allows developers to tune partition and chunk granularity, where Dryad provides a simplified data model and interface on .NET platform.

Scalability Study

Scalability is another key feature for parallel runtimes. The scalability test of DryadLINQ CTP includes two set of experiments conducted on Tempest Cluster. A comparison of parallel efficiency for DryadLINQ CTP and DryadLINQ 2009 is discussed.

The first experiment has input size between 5,000 and 15,000 sequences, with an average length of 2500. The sub-block matrix size is 128 × 128 and the number of partitions is chosen as 31, which is the optimal value found in previous experiments on 31 compute nodes. 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 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 slight 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.

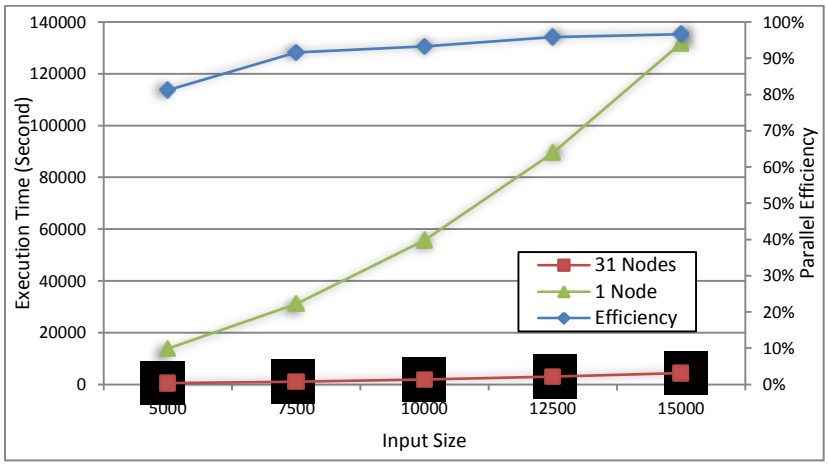


Figure 8: Performances and Parallel Efficiency on TEMPEST

Same SWG jobs are performed on 8 nodes of MADRID cluster [Appendix D] using Dryad 2009 and 8 nodes on TEMPEST cluster [Appendix C] using Dryad 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 Dryad CTP and Dryad 2009 showed high utilization of CPUs, where parallel efficiency is over 95% in Figure 9.

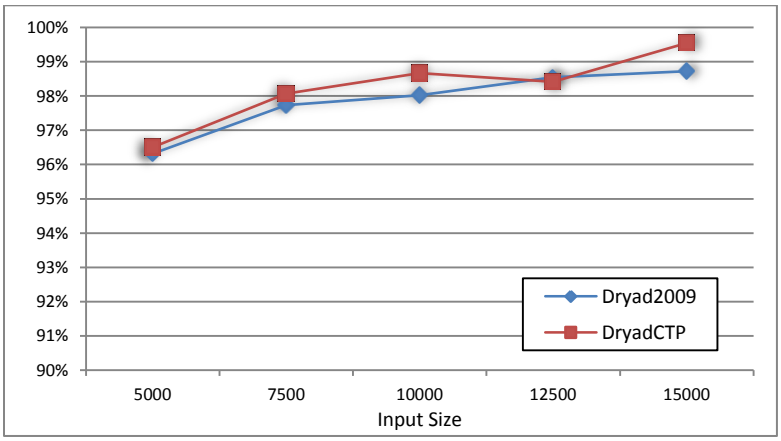


Figure 9: Parallel Efficiency on Dryad CTP and Dryad 2009

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

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

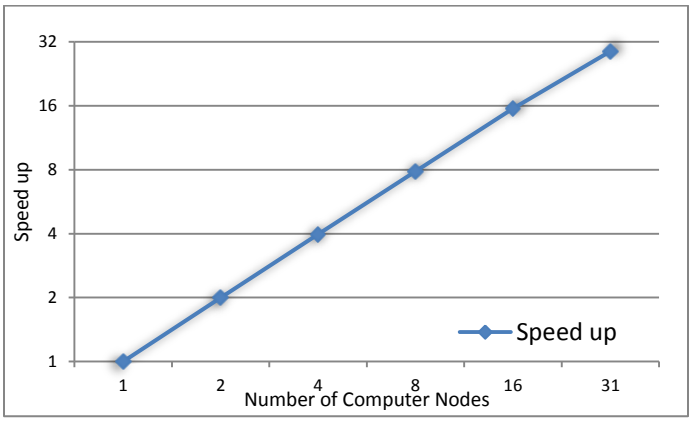


Figure 10: Speed up for SWG on Tempest with Varied Size of Compute Nodes

3.3 Scheduling on Inhomogeneous Cluster

Adding a new hardware or integrating distributed hardware resources is common but may cause inhomogeneous issues for scheduling. In Dryad 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 Dryad CT, where how to schedule tasks with attention to load balance is studied in this section.

Workload Balance with Different Partition Granularities

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

This experiment is performed on STORM [Appendix A], an inhomogeneous HPC cluster. A set of SWG jobs are scheduled with different partition size, where input data contains 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 imply 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 F, when the partitions granularity is large, the distribution of SWG blocks among the nodes is proportional to computation capability of nodes.

Dryad CTP assigns a vertex to a compute node and each vertex contains one to multiple 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, where X-axis labels compute node (e.g. cn01 ~ cn06) and Y-axis is the elapsed time from the start of computation. 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 and leads to a big variation of CPU time on each node. Note that faster nodes stay idle and wait for the slower ones to finish as shown in Figure 11 (left).

- When the number of partition is large, workload is distributed proportional to the capability of compute nodes. Too many partitions causes high scheduling cost thus slows down overall computation as in Figure 11 (right).
- There's a tradeoff − load balance favors small partitions while scheduling cost favors large jobs. An optimal performance is achieved in Figure 11 (center).

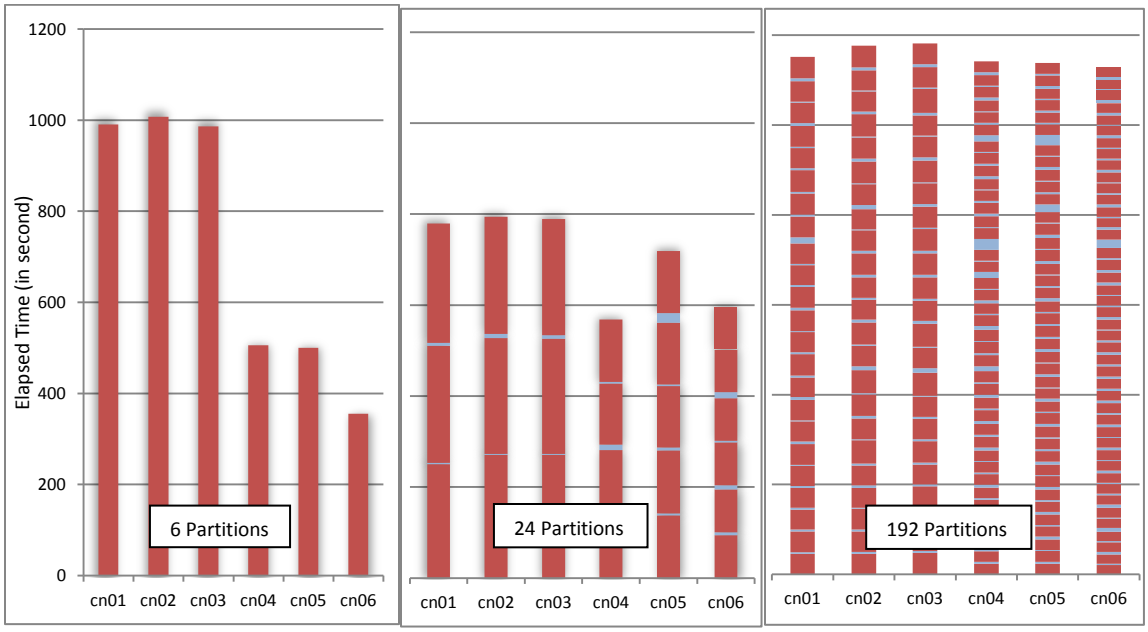


Figure 11: Scheduling Time vs. Computation Time of SWG application on Dryad CTP

The optimal number of partitions is a moderate number with respect to both load balance and scheduling cost. As shown in Figure 12, the optimal number of partitions is 24, in between the minimal number of 6 and the maximum number of 192. Note that 24 partitions performed better than the default partitions number, 14.

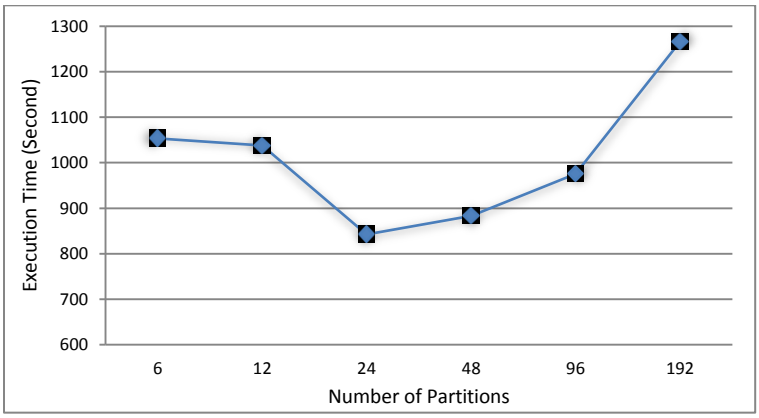


Figure 12: SWG job Turnaround Time for Different Partition Granularities

Figure 13 shows that the overall CPU usage drops as the number of partitions increases, due to more scheduling and data transferring, which don't demand high CPU usage.

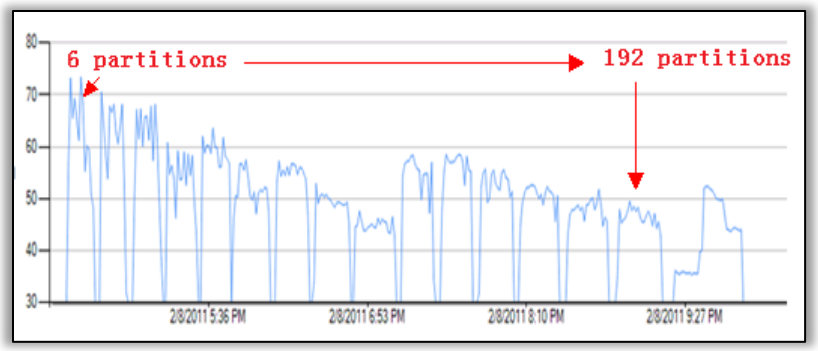


Figure 13: Cluster CPU Usage for Different SWG Partition Numbers

3.4 Conclusion

SWG is a pleasingly parallel application used to evaluate the performance of Dryad CTP. In scalability test, when input data is homogeneous and workload is balanced, same number of partitions as compute nodes (31 partitions) is an optimal value resulting in low scheduling cost. In partition granularity test, data is inhomogeneous and causes unbalanced workload. The default Dryad CTP setting (62 partitions) gave an optimal value due to a better balance between workload distribution and scheduling cost. A comparison between Dryad CTP and Dryad 2009 shows that Dryad CTP has achieved over 95% parallel efficiency in scale up tests. Compared to the 2009 version, it also presents an improved load balance with the dynamic scheduling function. Our research demonstrates that load balance, task granularity and data homogeneity are major factors that impact the performance of pleasingly parallel applications using Dryad. There's a bug of mismatched partitions vs. compute nodes in the default setting of Dryad CTP.

4 Hybrid Parallel Programming Model

4.1 Introduction

To explore the hybrid parallel programming model, we implemented DryadLINQ Matrix-Matrix Multiplication with three different algorithms and four multi-core technologies in .NET. The three matrix multiplication algorithms are: 1) row split algorithm, 2) row/column split algorithm, 3) two dimension block decomposition split in Fox algorithm [14]. The multi-core technologies are: PLINQ, TPL [15] and thread pool.

Matrix multiplication is defined as A * B = C (Eq. 3) where Matrix A and Matrix B are input matrixes and Matrix C is the result matrix. The **p** in Equation 3 is the number of columns in Matrix A and number of rows in Matrix B. Matrices in the experiments is square matrix by default in this report. Each element in the matrices is a double number.

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

Matrix-matrix multiplication is a fundamental kernel [16], which can achieve high efficiency in both theory and practice. The computation can be split into many homogeneous sub tasks, which make itself an ideal candidate application that explores hybrid parallel programming model in Dryad/DryadLINQ. We investigated the performance of three matrix multiplication algorithms with both single and multiple cores on HPC clusters.

4.2 Parallel Matrix-Matrix Multiplication Algorithms

We implemented DryadLINQ Matrix-Matrix Multiplication with three different algorithms: 1) row split algorithm, 2) row/column split algorithm, and 3) a 2 dimension block decomposition split in Fox's algorithm [14]. The performance comparisons for different input sizes are analyzed.

Row Split Algorithm

The row-split algorithm splits the Matrix A by rows. It scatters the split rows blocks of Matrix A onto compute nodes. The whole Matrix B is broadcasted or copied to every compute node. Each Dryad task multiplies the row blocks of Matrix A by whole matrix B and retrieves the output results to user main program to aggregate the complete output matrix C. The program flow of the Row Split Algorithm is shown in Figure 14.

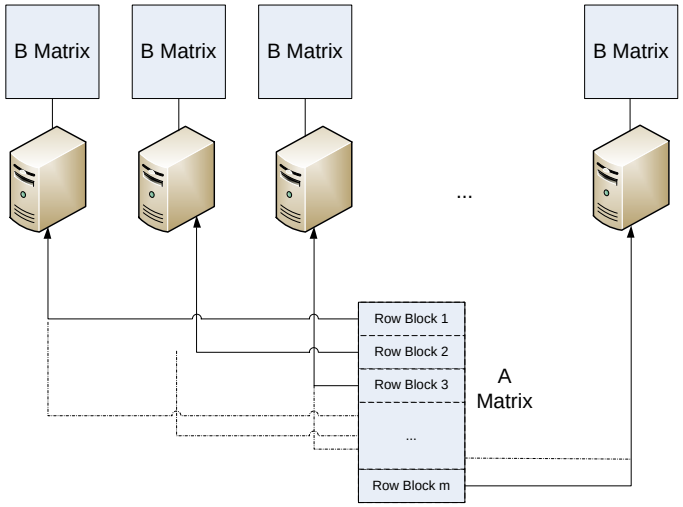


Figure 14: Row Split Algorithm

The blocks of Matrices A and B can be stored as a DistributedQuery and DistributedData objects defined in DryadLINQ. Afterwards, an ApplyPerPartition operator invokes user-defined function rowsXcolumnsMultiCoreTech to perform the sub task calculation. The compute node will get the file names for each input block and read the matrix remotely. Each task of row split algorithm of Matrix Multiplication are homogeneous in CPU time, thus the ideal partition number equals the of compute nodes. The pseudo code is shown as follows:

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

Row/Column Split Algorithm

The Row/Column split algorithm was brought up in [17]. The Row/Column split algorithm splits Matrix A by rows and split Matrix B by columns. The column blocks of B are scattered across the cluster in advance. The whole computation is divided into several iterations, each of which multiplies the one row block of A to all the column blocks of B on compute nodes. The output of tasks within the same iteration will be retrieved to the main program to aggregate one row block of Matrix C. The main program collects results in multiple iterations to generate the final output of Matrix C.

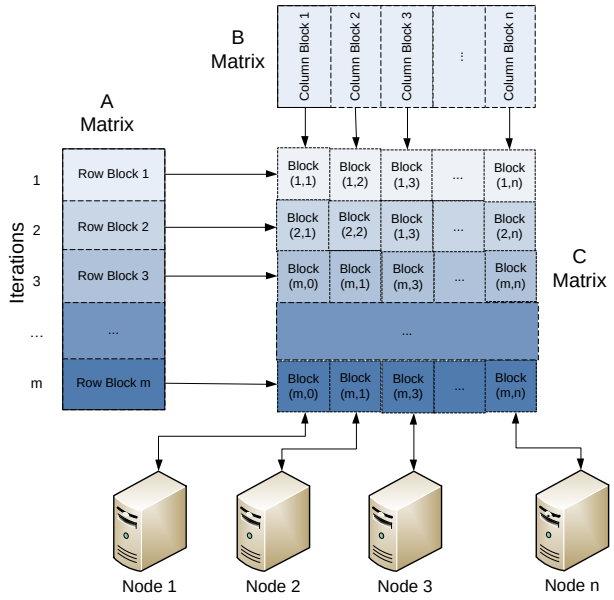


Figure 15: Row Column Split Algorithm

In the implementation, the *DistributedQuery* objects of columns blocks of Matrix B are distributed across the compute nodes initially For each iteration, an ApplyPerPartition operator is used to invoke a user-defined function *aPartitionMultiplybPartition* to multiply the one column block of Matrix B and one row block of Matrix A. The pseudo code is as follows:

```
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)));
}
```

2D Block Decomposition in Fox Algorithm

The two dimensional block decomposition in Fox algorithm splits Matrix A and Matrix B into squared sub-blocks. These sub-blocks are dispatched to a squared process mesh with same scale. For example, let's assume to run the algorithm on a 2X2 processes mesh. Accordingly, Matrices A and B are split by both rows and columns and construct a 2X2 block mesh respectively. In each computation step, every process holds a block of Matrix A and a block of Matrix B and computes 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

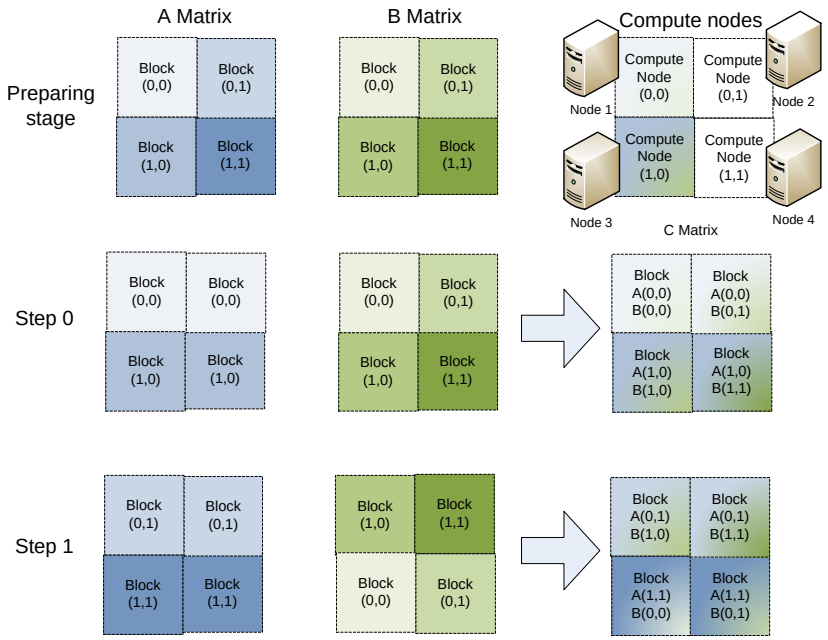


Figure 16: Different stages of an order 2 2D block decomposition illustrated

Figure 16 is an example of Matrices A and B both divided into blocks mesh of 2x2. Assume there are 4 compute nodes, which are labeled C(0,0), C(0,1), C(1,0), C(1,1). In step 0, the Matrix A will broadcast the blocks in column 0 to the compute nodes in the same row of logic grid of compute nodes. i.e. Block(0,0) to C(0,0), C(0,1) and Block(1,0) to C(1,0), C(1,1). The blocks in Matrix B will be scattered onto each compute node, and perform a upward circular shift in each column.,The algorithm compute the $C_{ij} = AB$ on each compute node. In the Step 1, the Matrix A will broadcast the blocks in the column 1 to the compute nodes, i.e. Block(0,1) to C(0,0), C(0,1) and Block(1,1) to C(1,0), C(1,1). The Matrix B scattered each block to the compute node within rotation in columns, 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 we will compute C_{ij} += AB.

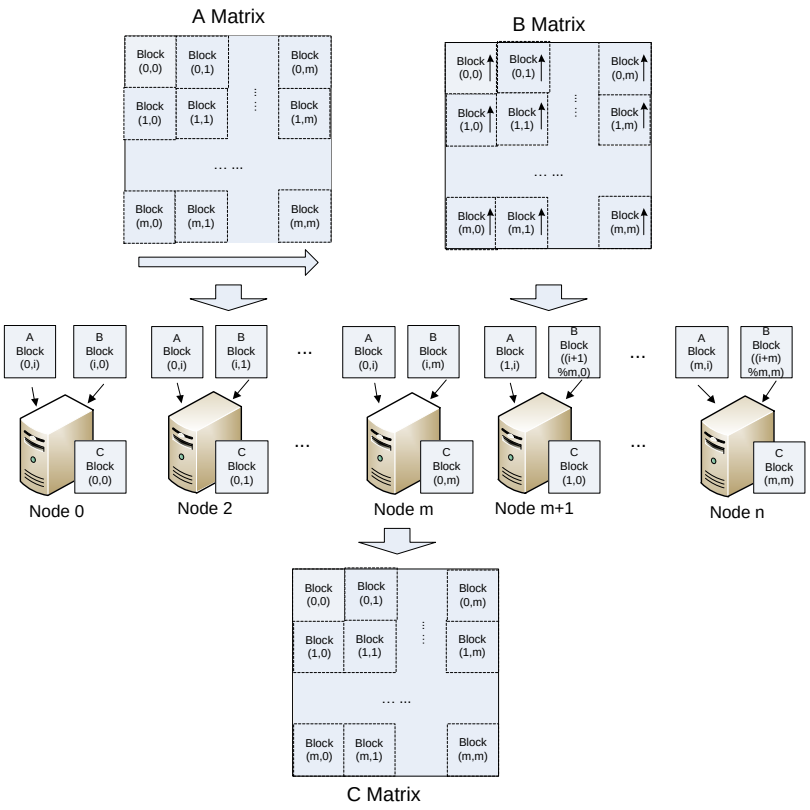


Figure 17: Structure of 2D decomposition algorithm on iteration i with n+1 nodes

Figure 17 is the workflow of the Fox algorithm for (M+1) order square matrix on (N+1) nodes. In each step, the blocks of Matrix A and Matrix B will be scattered onto each compute node; therefore, each compute node will have one block of Matrix A and one block of Matrix B, where the fraction of memory usage of Fox algorithm over that of row split algorithm is 2/(N + sqrt(N)). The pseudo code of the algorithm is as follows:

```
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));
}
```

The Fox algorithm is originally implemented with MPI, which requires maintaining intermediate status and data within processes during the computation. The Dryad implementation uses a data flow runtime, which does not support status of tasks during computation. In order to keep the intermediate status and data, the update operations to DistributedQuery<T> objects are assigned with new status and data to themselves, where the pseudo code is included as follows:

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

4.3 Multi-Core Technologies

Inside the function rowXcolumns, the sequential code without using any multicore technology is as follows:

```
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 to perform Matrix Multiplication is as follows:

```
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 to perform Matrix Multiplication is as follows:

```
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 to perform Matrix Multiplication is as follows:

```
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;
}
```

4.4 Performance Analysis in Hybrid Parallel Model

Performance in Multi Core Parallelism

We executed the Matrix Multiplication algorithms on TEMPEST with various multicore technologies in .NET. As the platform is based on .NET 4, parallel for inside Task Parallel Library (TPL), thread pool and PLINQ to do the parallelism has been used for the test.

The test has been done on one of tempest compute node, which has 24 cores; the size of matrix A and matrix B are from 2400 * 2400 to 19200 * 19200. In this way, the computation for each thread will increase as the data size increase, and the result is shown in Figure 18.

The speed up of the Matrix Multiplication application with various matrix sizes on TEMPEST nodes were calculated using one core in Equation 3. The T(P) standard for job turnaround time for Matrix Multiplication with multi-core technologies, where P represents the number of cores across the cluster. T(S) means the job turnaround time of sequential Matrix Multiplication with only one core.

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

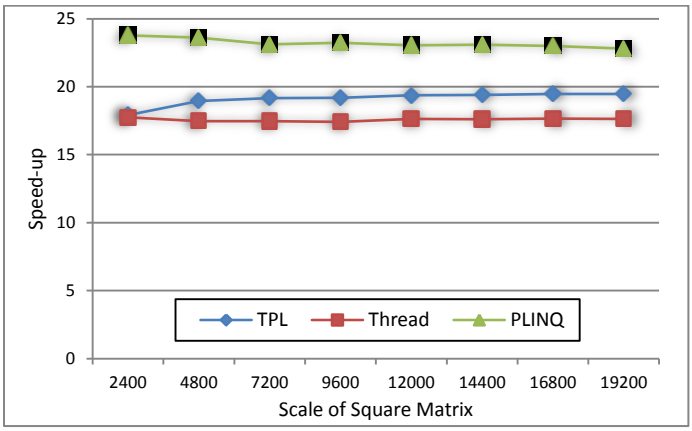


Figure 18: the speed up chart for different method of multi-core parallelism on 1 node

From Figure 18, the efficiency of parallelism remains around 17 to 18 for TPL and Thread. However, while the data size becomes larger, the TPL do performs better than the thread pool. PLINQ has the best speed up all the time; the speed up of PLINQ is always larger than 22, make the parallel efficiency over 90%. The efficiency of TPL and thread seems low should due to the memory cache issue as the data is large. Obviously, PLINQ has some optimization for large data size to make it very fast on multicore system.

Performance in Multi Node Parallelism

The Matrix Multiplication jobs on TEMPSEST with various sizes of square Matrices show 1) the applicability and performance of DryadLINQ to advanced parallel programming model that require complex messaging control and 2) the porting multi-core technologies in DryadLINQ task can increase the overall performance greatly.

We evaluate 3 different Matrix Multiplication algorithms without using any multicore technology on 16 nodes from TEMPEST. The data size of input matrix is from 2400 x 2400 to 19200 x 19200.

The approximate speed-up of Matrix Multiplication jobs is shown in Equation 4. The T(P) standards for job turnaround time for Matrix Multiplication on P compute nodes And we only use one core on each compute node. T(S') means the approximation of job turnaround time of sequential Matrix Multiplication program. As job turnaround time for sequential version is very large, we construct a fitting function in Appendix E to calculate CPU time for large input data.

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

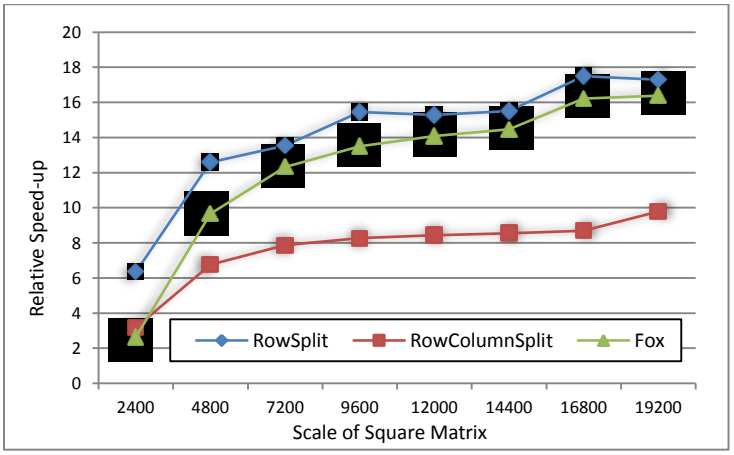


Figure 19: the speed up for different Matrix Multiplication models using one core per node

As shown in Figure 19, the Fox algorithm's performance is almost the same as the Row Split algorithm, and it increases faster as the input data size goes up. And the Row/Column Split Algorithm performs worst because this is an iterative algorithm, and it need explicitly evaluate the DryadLINQ query within iteration to collect intermediate result, which will trigger resubmit Dryad task to HPC job manager during each iteration.

Performance in Hybrid model with different multi-core technologies

Porting multi-core technologies into Dryad tasks can increase the overall performance greatly. Here, our conclusion by applying single-core and PLINQ into row split Matrix Multiplication algorithm. The experiments with various matrix sizes (from 2400x2400x2400 to 19200x19200x19200) on 16 compute nodes HPC cluster (Appendix B) with coarse parallel task granularity approach, where each parallel task deals with the calculation of one row of result matrix C. Within each compute node (24 cores), we make use of parallel for (TPL), thread pool, and PLINQ to perform the Matrix Multiplication calculation in parallel. As shown in Figure 20, 1) the speed up of implementation that utilizes multi core technology is much bigger than sequential version. 2) The speed-up increases as the input data size increases. We should also point out that the maximum speed-up ratio between sequential version and multi-core version in this chart is about 7 which are much smaller than the number of cores. The reason may be caused by the memory/cache mechanism of hardware and the size of dataset are fixed when we port the multicore technology. This issue has been discussed in the following section.

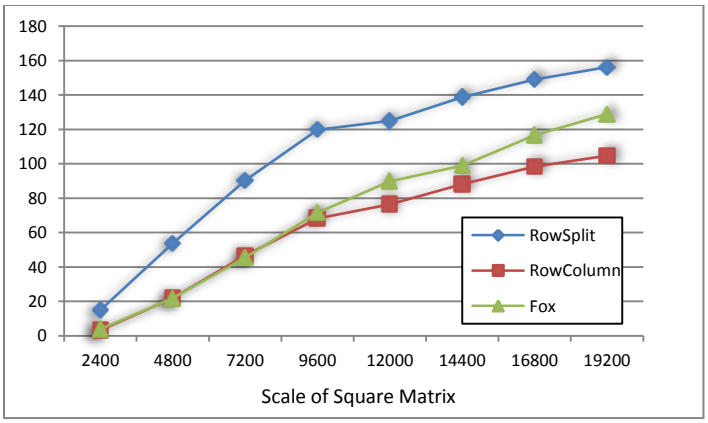


Figure 20: the time consumed for 3 Algorithms using PLINQ

Fig 19 shows that the Fox algorithm does not perform well as RowSplit alogrithm in Figure 19. In matrix-matrix multiplication, the computation cost O(n^3) increases faster than the communication cost O(n^2). Thus after porting multi-core into Dryad task, the task granularity for the Row Split and Row/Column Split algorithm becomes finer as well, which alleviates the low cache hit rate issue for coarse-granularity task.

Performance Analysis of Hybrid Parallel Models

Evaluations on the hybrid parallel programming models, which integrate different algorithms with different multicore technologies is studied. The scale of input matrix is 19200. Figure 21 is the performance results for these hybrid models. As it shown in the figure, hybrid model that applies Row/Column split algorithm delivers the best performance, where the reason is its simplified programming model and the lower overhead in scheduling. In the Row/Column split algorithm, the speed up using PLINQ is 104. In the Fox algorithm, using the PLINQ technology can bring the speed up at the 60, while the other technologies only bring the speed up at around 40. It is obviously that PLINQ in Fox is much faster than the other two approaches, where the reason may lie in its finer task granularity compared to other algorithms.

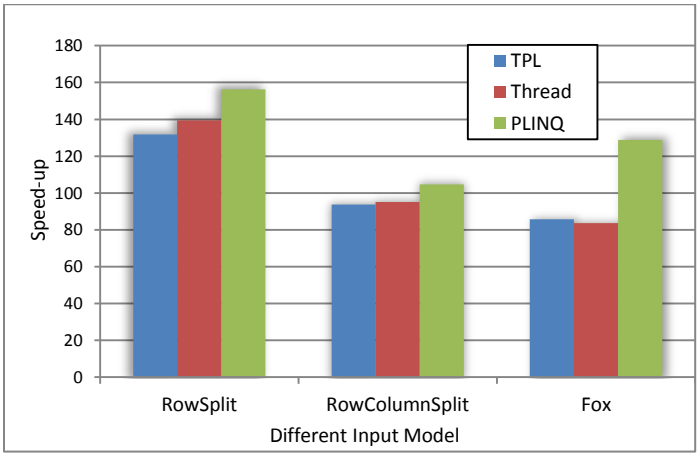


Figure 21: The speed up chart for different multicore technology apply on Matrix Multiplication

As it shown in Figure 21, applying the hybrid parallel programing models can dramatically decrease the parallel efficiency. This is caused by two kinds of overhead. The first one is the synchronization overhead. When the number of cores increases, so does the number of concurrent tasks. And, the overall job turnaround time is mainly determined by the slowest task. The second one is the communication overhead. On one side the communication

overhead among tasks/processes on one node increases, on the other side the communication overhead of jobs/tasks among cluster increases relatively. The reason is the CPU time for each Dryad task decreases due to the applying of multi-core technologies, while communication overhead among nodes almost remains the same.

Generally, keeping the problem size fixed, the parallel efficiency becomes lower after integrating the multicore technologies. This is because the task granularity for each core becomes finer, while the scheduling and communication overhead remain the same or increase. Assume the parallel efficiency $P_m = \frac{S_m}{N_m}$, where the S_m is the speed up and N_m is the number of cores after importing the multicore technologies. And the $N_m = N * m$ where the **N** is the number of nodes and **m** is number of cores per node. The speed up for single core per node is $S = T_1/T_n$ where T_1 is the sequential time for the program where the T_n is the actually running time using **N** nodes, so $P_n = \frac{S}{n} = \frac{1}{n} * \frac{T_1}{T_n}$. As $T_n = T_n^c + T_n^o$ where the T_n^c is the actual computation time and T_n^o is the parallel overhead (communication, scheduling) for **N** nodes. For pleasingly parallel applications, an approximation equation, where $T_1 = T_n^c * n$. And, for **m** cores per node, the $T_{nm}^c = T_n^c/m$ where the $T_{nm}^o \approx T_n^o$. (we ignore the parallel overhead among cores to simplify the deduction.) Then we have

$$P_{n*m} = \frac{S_m}{N_m} = \frac{1}{n*m} * \frac{T_1}{T_{nm}} = \frac{1}{n} * \frac{T_1}{T_n^c + m * T_n^o}$$

Obviously $T_n^c + m * T_n^o$ is larger than $T_n^c + T_n^o$, and the parallel efficiency P_{n*m} is smaller than P_n. In the Matrix Multiplication application in this report, **m** = 24 and $\frac{T_n^c}{T_n^o} = O(n)$.

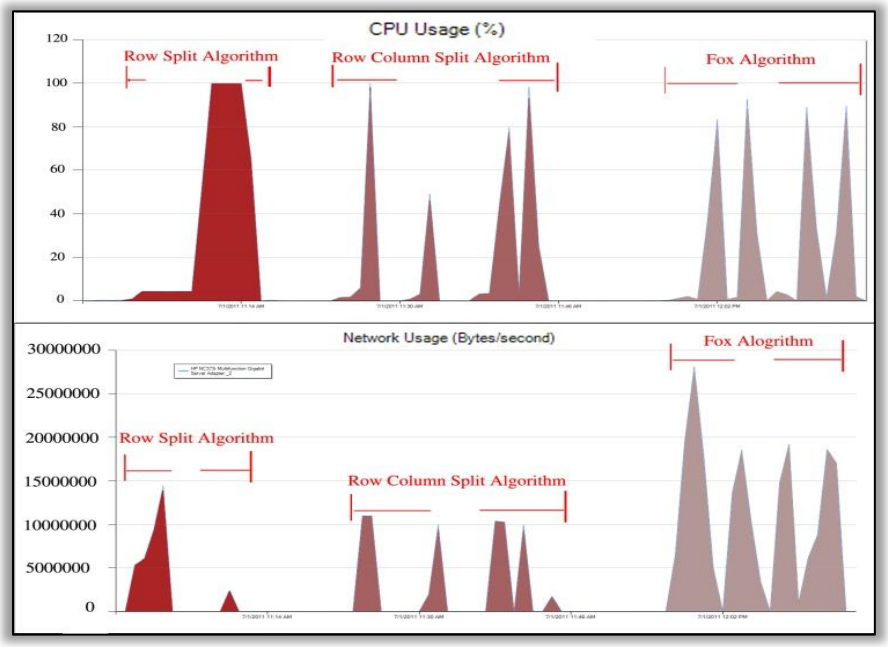


Figure 22: The CPU usage and network activity on one compute node for multiple algorithms

Figure 22 is chart of the CPU utilization and network activity on one node of the TEMPEST of three 19200 x 19200 matrix multiplication jobs that using PLINQ in Row Split Algorithm, Row/Column Split Algorithm and Fox Algorithm respectively. The lower graph is aggregated CPU usage rate and the higher graph is aggregated network activity. Figure 22 show that the Row Split Algorithm with PLINQ can reach a CPU utilization rate at 100% for longer time than the other two approaches. Besides, its aggregated network overhead is less than the other two

approaches as well. Thus Row Split algorithm with PLINQ has the shortest job turnaround time among the three approaches. Fox algorithm delivers a good performance in the sequential version due to cache and paging advantage of finer task granularity. But after integrating the multicore technology, the task granularity becomes too small, which leads to the low CPU utilization. Actually, the Fox algorithm is designed to execute with MPI on HPC, while the Dryad is a data flow processing runtime that deployed on Cloud. It is the original design goal leads to its not good performance in Cloud.

However, this effect will be decreases by increasing the size of matrix since the computation time is $O(N^3)$. Following chart is the test we have done with two different size of matrix computation using fox algorithm.

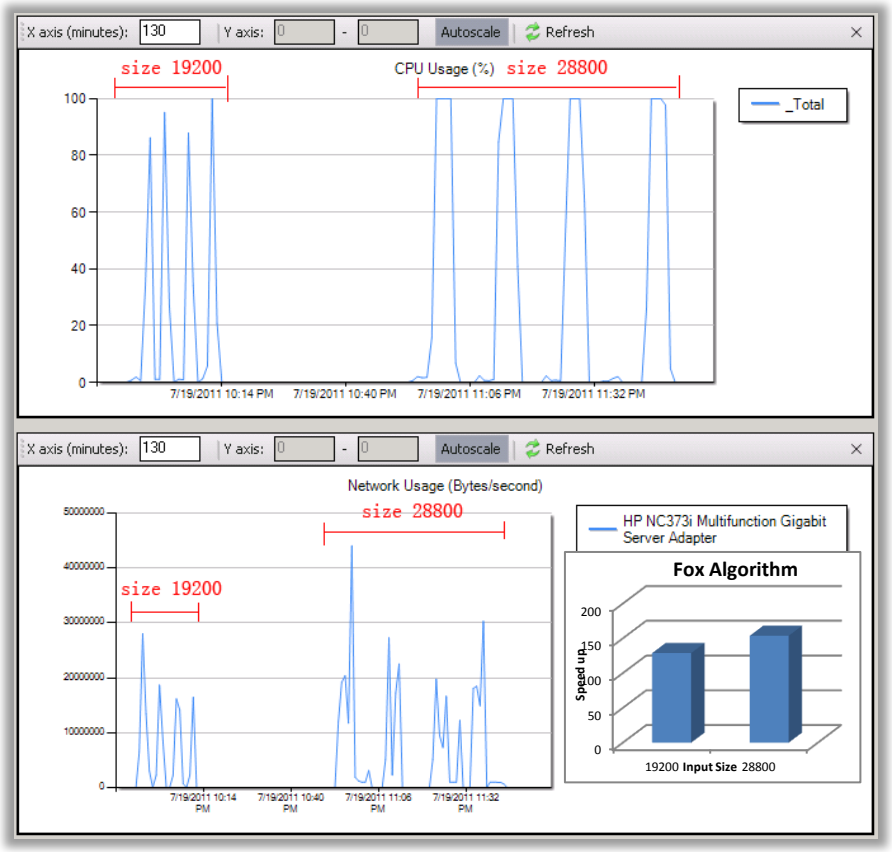


Figure 23: The CPU usage and network activity for Fox Algorithm-DSC by different sizes

Figures 22 and 23 show the Fox algorithm does not performing as well when integrating the PLINQ, where the reason may be the too small for task granularity. To verify this argument, we investigated the performance of Fox/PLINQ jobs with both small and large input data. Figure 23 is the CPU utilization and network utilization of two Fox/PLINQ jobs whose input size are 28800 x 28800 x 28800 and 19200 x 19200 x 19200 respectively.

Figure 23 shows that large input data increases both CPU and network utilization. More importantly, the increase rate of CPU utilization is faster than that of network utilization, which obeys the theory trend. In Matrix Multiplication, the CPU complexity of input size N is O(N^3), while the network complexity is O(N^2). Thus the overall speed up will keep increasing when we increase the data size.

4.5 Conclusion

We investigate the hybrid parallel programming models with the Matrix Multiplication application. We show that integrating the multicore technologies into Dryad tasks can increase the overall utilization of cluster. Besides, different combinations of multicore technologies and parallel algorithms have different performance where the reasons lie in task granularity, caching and paging issues. We also find that the parallel efficiency of jobs decreases dramatically after integrating the multicore technologies where the main reason is the task granularity becoming too small. Increasing the scale of input data can alleviate this issue.

5 Distributed Grouped Aggregation

5.1 Introduction

We investigated the usability and performance of programming interface of the distributed grouped aggregation in DryadLINQ CTP with PageRank with real data. Specifically, an evaluation for three distributed grouped aggregation approaches implemented with DryadLINQ CTP is: Hash Partition, Hierarchical Aggregation and Aggregation Tree. Further, the features of input data that affect the performance of distributed grouped aggregation implementations. In the end, we compare the performance of distributed grouped aggregation of DryadLINQ to that of other job execution engines: MPI, Hadoop, Haloop and Twister.

The PageRank is already a well-studied web graph ranking algorithm. It calculates the numerical value to each element of a hyperlinked set of web pages, which reflects the probability that the random surfer accesses those pages. The process of PageRank can be understood as a Markov Chain which needs recursive calculation to converge. An iteration of the algorithm calculates the new access probability for each web page based on values calculated in the previous computation. The iterations will not stop until the Euclidian distance between two subsequent rank value vectors becomes less than a predefined threshold. In this paper, we implemented the DryadLINQ PageRank with the ClueWeb09 data set [18] which contains 50 million web pages.

5.2 Distributed Grouped Aggregation Approaches

The Distributed Grouped Aggregation is a core primitive operator in many data mining applications like sales data summarizations, the log data analysis, and social network influence analysis. In this section, we study three distributed grouped aggregation approaches implemented with DryadLINQ, which include: Hash Partition, Hierarchical Aggregation and Aggregation Tree. Figure 24 is the work flow of the three approaches.

The Hash Partition approach uses hash partition operator to redistributes the records to compute nodes so that identical records store on the same node and form the group. Then it merges the records of each group into certain value. The hash partition approach is of simple implementation but will cause lots of network traffic when the number of input records is very large. A common way to optimize this approach is to apply partial pre-aggregation. It first aggregates the records on local compute node, and then redistributes aggregated partial results across cluster based on their key. This approach is better than directly hash partition because the number of records transferring across the cluster becomes much fewer after local aggregation operation. Further, there are two approaches to implement the partial aggregation: 1) hierarchical aggregation 2) aggregation tree. The hierarchical aggregation is usually of two or three aggregation layers each of which has the explicitly synchronization phase. The aggregation tree is the tree graph that guides job manager to perform the partial aggregation for many subsets of input records.

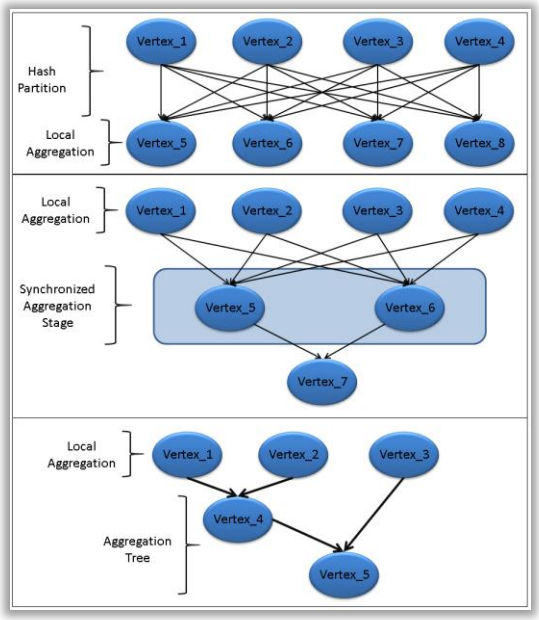


Figure 24: Three distributed grouped aggregation approaches

DryadLINQ can automatically translate the distributed grouped aggregation query and its combine functions to satisfy the associative and commutative rules into optimized aggregation tree. During computing, Dryad can adaptively change the structure of aggregation tree without additional code from developer side. This mechanism greatly simplifies the programming model and enhances the performance at the same time.

Hash Partition

A direct implementation of PageRank in DryadLINQ CTP is to use GroupBy and Join as follows:

PageRank implemented with GroupBy() and Join()

```
for (int i = 0; i < iterations; i++)
{
        newRanks = pages.Join(ranks, page => page.source, rank => rank.source,
        (page, rank) => page.links.Select(dest =>newRank(dest, rank.value /
(double)page.numLinks)))
        .SelectMany(list => list).GroupBy(rank => rank.source)
        .Select(group =>newRank(group.Key, group.Select(rank => rank.value).Sum() * 0.85 + 0.15 /
(double)_numUrls));
         ranks = newRanks;
}
```

The **Page** objects are used to store the structure of web graph. Each element **Page** in collection **pages** contains a unique identifier number page.key and a list of identifiers specifying all the pages in the web graph that **page** links to. We construct the DistributedQuery<Page> **pages** objects from the AM files with function BuildPagesFromAMFile(). The **rank** object is a pair specifying the identifier number of a page and its current estimated rank value. In each iteration the program JOIN the **pages** with **ranks** to calculate the partial rank values. Then GroupBy() operator redistribute the calculated partial rank values across cluster and return the IGrouping objects (groups of group), where each group represents a set of partial ranks with the same source page pointing to them. The grouped partial rank values are summed up to new final rank values and will be used as input rank values for the next iteration[19].

In above implementation, the GroupBy() operator can be replaced by the HashPartition() and ApplyPerPartition() operators as follows:

PageRank implemented with HashPartition() and ApplyPerPartition()

```
for (int i = 0; i < _iteration; i++)
{
       newRanks = pages.Join(ranks, page => page.source, rank =>rank.source,
        (page, rank) => page.links.Select(dest =>new Vertex(dest, rank.value /
(double)page.numLinks)))
       .SelectMany(list => list).HashPartition(record => record.source)
       .ApplyPerPartition(list => list.GroupBy(record => record.source))
       .Select(group =>newRank(group.Key, group.Select(rank =>rank.value).Sum() * 0.85 + 0.15 /
(double)_numUrls));
       ranks = newRanks.Execute();
}
```

Hierarchical Aggregation

The PageRank implemented with hash partition approach is not efficiency when the number of output tuples is much less than that of input tuples. For this scenario, we implement PageRank with hierarchical aggregation approach which has three synchronized aggregation stages: 1) the initial aggregation stage for each user defined Map task. 2) the second stage for each DryadLINQ partition. 3) the third stage to calculate the final PageRank values. In stage one, each user-defined Map task calculates the partial results of some pages that belongs to sub web graph represented by the AM file. The output of Map task is a partial rank value table, which will be merged into global rank value table in later stage. Thus the basic processing unit of our hierarchical aggregation implementation is a sub web graph rather than one paper in hash partition implementation. The coarse granularity processing strategy has a lower cost in task scheduling, but it requires additional program effort and the sophisticated understanding of web graph from developer side.

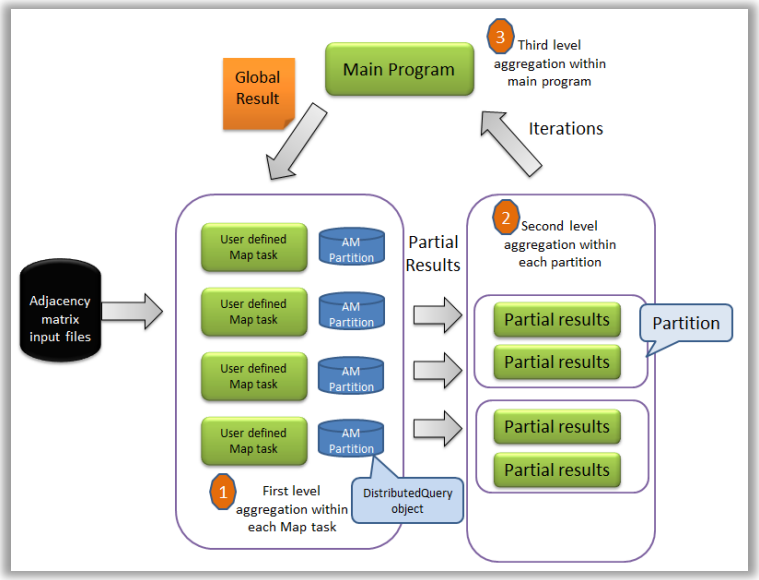


Figure 25: Hierarchical aggregation in DryadLINQ PageRank

Hierarchical Aggregation with User Defined Aggregation function

```
DistributedQuery<amPartition> webgraphPartitions =
Directory.GetFiles(inputDir).AsDistributed().Select(fileName =>
buildWebGraphPartition(fileName));
for (int i = 0; i < numIteration; i++)
{
        DistributedQuery<double[]> partialRVTs = null;
        partialRVTs = webgraphPartitions.ApplyPerPartition(subWebGraphPartitions =>
        calculateMultipleWebgraphPartitionsWithPLINQ(subWebGraphPartitions, rankValueTable,
numUrls));
        rankValueTable = mergeResults(partialRVTs);
        //synchronized step to merge all the partial rank value tables.
```

```
}
```

Aggregation Tree

The hierarchical aggregation approach may not perform well for computation environment which is inhomogeneous in network bandwidth, CPU and memory capability; because it has several synchronization stages and the performance of whole computation is mainly determined by slowest task. In this scenario, the aggregation tree approach is a better choice. It can construct a tree graph to guide the job manager to make aggregation operations for many subsets of input tuples so as to decrease intermediate data transformation. In ClueWeb data set, the URLs are stored in alphabet order, web pages belong to same domain are more likely saved within one AM file. Thus the intermediate data transfer in the hash partition stage can be greatly reduced by applying the partial grouped aggregation to each AM file. The GroupAndAggregate() operator of DryadLINQ CTP supports the aggregation tree mechanism. We implemented PageRank with GroupAndAggregate() operator as follows:

PageRank implemented with GroupAndAggregate

```
for (int i = 0; i < numIteration; i++)
{
        newRanks = pages.Join(ranks, page => page.source, rank =>rank.source,
                   (page, rank) => page.links.Select(targetPage =>newRank(targetPage, rank.value /
(double)page.numLinks)))
        .SelectMany(list => list)
        .GroupAndAggregate(partialRank  =>partialRank.source,  g  =>newRank(g.Key,  g.Sum(x  =>
x.value)*0.85+0.15 / (double)numUrls));
       ranks = newRanks;
}
```

The GroupAndAggregate operator makes aggregation optimization transparent to the programmers. To analyze the partial aggregation in detail, we simulate the GroupAndAggregate operator with the ApplyPerPartition and HashPartition APIs as follows. There are two ApplyPerPartition step in following code. The first ApplyPerPartition in following code segment perform the pre-partial aggregation to each partition which presents some part of web graph. The second ApplyPerPartition aggregate the aggregated partial results produced by first ApplyPerPartition.

PageRank implemented with two ApplyPerPartition steps

```
for (int i = 0; i < numIteration; i++)
{
    newRanks = pages.Join(ranks, page => page.source, rank =>rank.source,
    (page, rank) => page.links.Select(dest =>newVertex(dest, rank.value /
(double)page.numLinks)))
    .SelectMany(list => list)
    .ApplyPerPartition(subGroup => subGroup.GroupBy(e => e.source))
    .Select(subGroup =>new Tuple<int, double>(subGroup.Key,subGroup.Select(rank
=>rank.value).Sum()))
    .HashPartition(e => e.Item1)
    .ApplyPerPartition(subGroup => subGroup.GroupBy(e => e.Item1))
    .Select(subGroup =>newRank(subGroup.Key, subGroup.Select(e => e.Item2).Sum() * 0.85 + 0.15 /
(double)_numUrls));
    ranks = newRanks.Execute();
}
```

5.3 Performance Analysis

Performance in Different Aggregation Strategies

The performance of the three approaches by running PageRank jobs with various sizes of input data on 17 compute nodes of TEMPEST are evaluated. Figure 26 is the time in second of per iteration of the three aggregation approaches of PageRank with different number of AM files. The results show that aggregation tree approach is faster than hash partition approach due to the partial pre-aggregation optimization. And, the hierarchical aggregation outperforms the other two approaches because of its much coarser task granularity. Figure 27 is the CPU utilization

and network utilization statistic data obtained from HPC cluster manager for the three aggregation approaches. It is very obvious that the hierarchical aggregation requires much less network traffic than the other two approaches in the cost of CPU overhead. Besides, our experiment environment -- TEMPEST is homogeneous in network and CPU capability which is suitable to perform the hierarchical aggregation computation.

Note: in the experiments we split the entire ClueWeb09 graph into 1280 partitions, each of which is processed and saved as an Adjacency Matrix (AM) file. The characteristics of the input data are described as below:

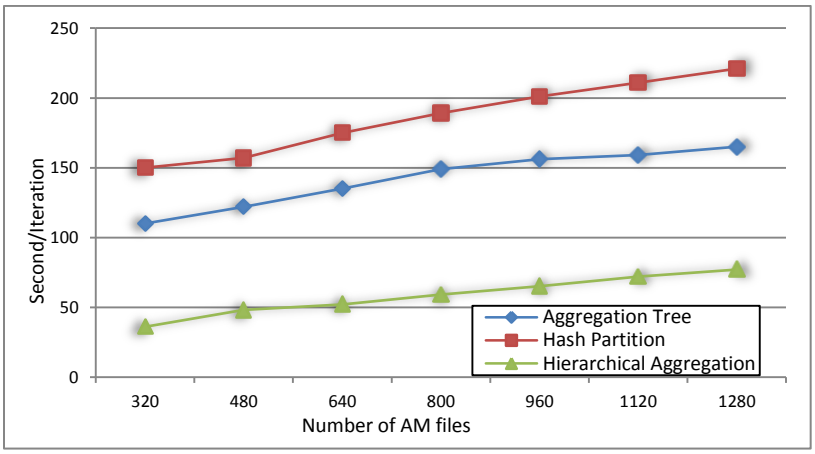


Figure 26: Time in sec to compute PageRank per iteration with three aggregation approaches with clue-web09 data set on 17 nodes of TEMPEST

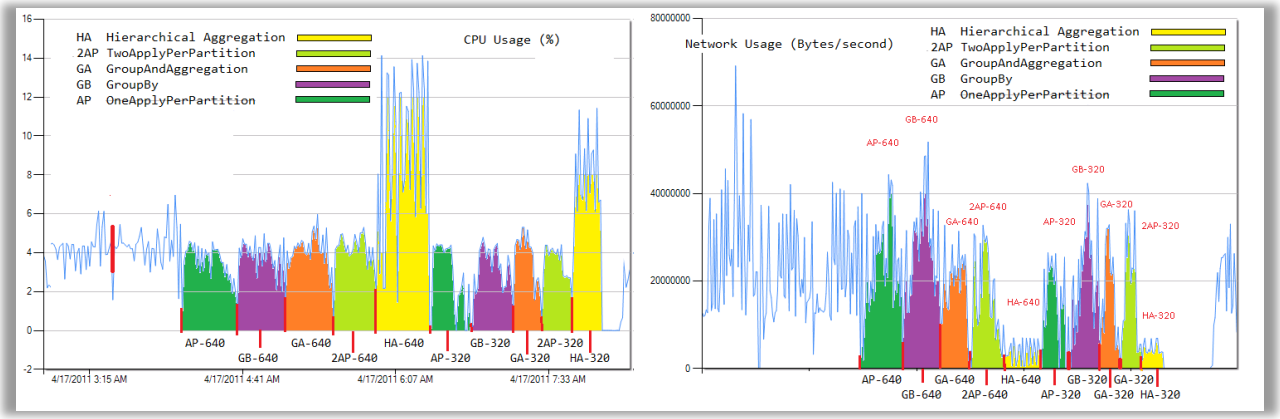


Figure 27 CPU and Network Utilization for Different Aggregation Strategies

The hierarchical aggregation and aggregation tree approaches mediate the tradeoffs between CPU overhead and network overhead. And, they work well only when the number of output tuples is much smaller than that of input tuples; while hash partition works well only when the number of output tuples is larger than that of input tuples. A mathematical model is used to describe how the ratio between input and output tuples affects the performance of aggregation approaches. First, the data reduction proportion (DRP) to describe the ratio is as follows:

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

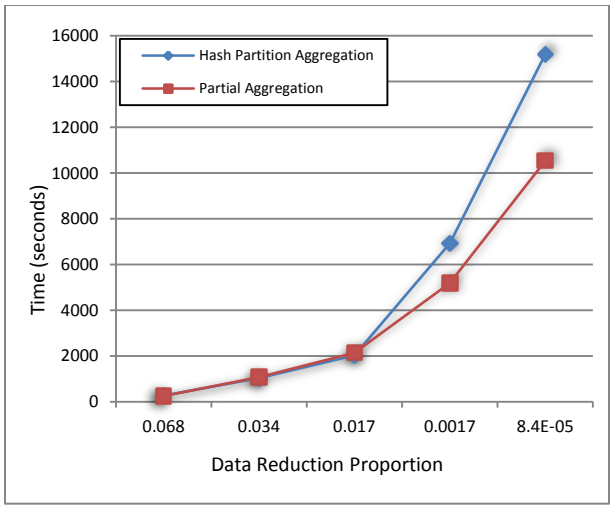


Figure 28: Time of PageRank jobs with two aggregation approaches with different DRP

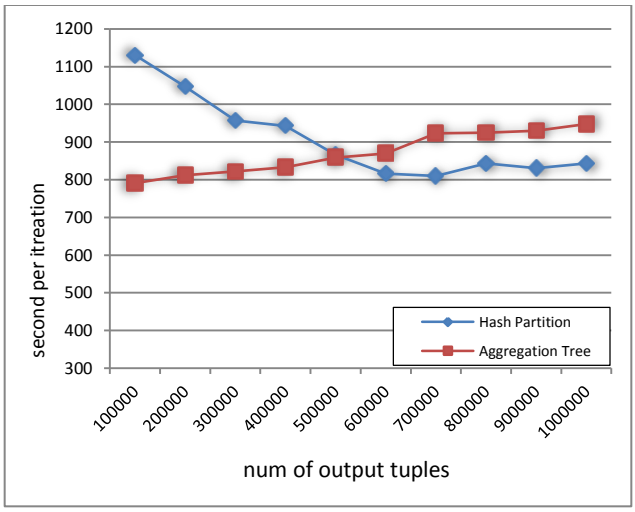


Figure 29: Time in seconds per iteration for two aggregation approaches with different number of output tuples (from 100000 to 1000000) when number of input tuples is 4.3 billion

Further, a mathematical model is derived to explain why DRP can affect the performance of aggregation approaches. Assume the average number of input tuples of each group is M (M=1/DRP); and there are N compute nodes; and assume the M tuples of each group are evenly distributed among the N nodes. In hash partition approach, the M tuples among N nodes with same key are hashed into one group, which requires M aggregation operations. In partial aggregation approach, the number of input tuples with 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 still 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. Then the average numbers of aggregation operations of each tuples of the two approaches are as follows:

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

Usually, DRP is much smaller than the number of compute nodes. Taking word count as an example, the documents with millions words may consist of only several thousand regular words. In PageRank, as the web graph structure obeys zipf's law, DRP is not as small as that in word count. Thus, the partial aggregation approach may not deliver performance as well as word count [21].

To quantitively analysis of how DRP affects the aggregation performance, we compare two aggregation approaches with a set of web graphs have different DRP by fixing the number of output tuples and changing that of input tuples. It is illustrated in Fig. 30 that when the DRP smaller than 0.017 the partial aggregation perform better than hash partition aggregation. When DRP bigger than 0.017, there is not much different between these two aggregation approaches. Fig. 31 shown the time in seconds per iteration of PageRank jobs of web graphs with different number of output tuples when that of input tuples fixed. Fig.30 and 31 indicate the DRP has material effect on the performance aggregation approaches.

Comparison with other implementations

We have implemented the PageRank with DryadLINQ, Twister, Hadoop, Haloop, and MPI on ClueWeb data set [18] with the Power method [19]. Five implementations with same 1280 AM input files but with different hardware and software environment are illustrated in Appendix F Table 8. To remove those inhomogeneous factors as much as

possible, parallel efficiency is calculated for each implementation. The parallel efficiencies of different implementations are calculated with Equation 8. The T(P) standard for job turnaround time of parallel version PageRank, where P represents the number of cores in the computation. T(S) means the job turnaround time of sequential PageRank with only one core.

$$\text{Parallel efficiency} = T(S)/(P*T(P)) \qquad \text{(Eq. 8)}$$

Figure 30 shows the parallel efficiency is noticeably lower than 1%. The first reason is that PageRank is communication intensive application, and the computation does not take large proportion of overall PageRank job turnaround time. Second, the bottlenecks of PageRank applications are network, memory, then the computation. Thus using multi-core technology does not help much to increase the parallel efficiency; instead it decreases overall parallel efficiency due to the synchronization cost among tasks. Usually, the purpose to make use of multi nodes in PageRank is to split large web graph to fit in local memory. It also shows how MPI, Twister and Haloop implementations outperform other implementations, where the reasons are: 1) they can cache loop-invariable data or static data in the memory during the overall computation, 2) make use of chained tasks during multiple iterations without the need to restart the tasks daemons. Based on our findings, Dryad is faster than Hadoop, but is still much slower than MPI and Twister for PageRank jobs. Dryad can chain the DryadLINQ queries together so as to save communication cost, but its scheduling overhead of each Dryad vertex is still large compare to that of MPI, Twister workers. Hadoop has the lowest performance, because it has to write the intermediate data to HDFS in each iteration.

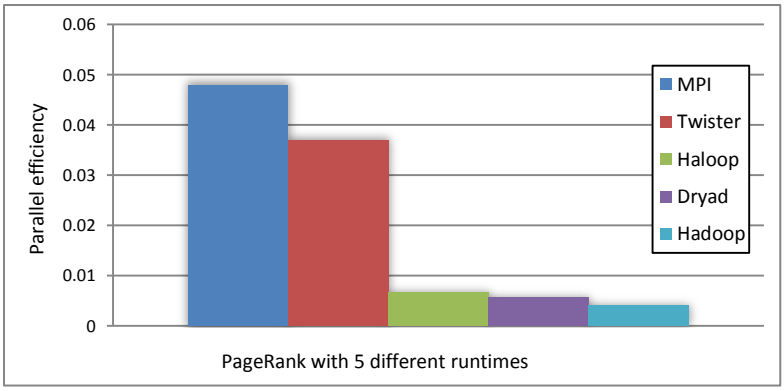


Figure 30: Parallel efficiency of five different PageRank implementations

Chaining Tasks within BSP Job

Dryad can chain the execution of multiple DryadLINQ queries together with the late evaluation technology. The chained DryadLINQ queries will not get evaluated until program explicitly evaluates chained queries or access the output results of queries. Figure 31 shows the performance improvement of chaining DryadLINQ queries when run 1280 AM files of PageRank jobs on TEMPEST for 10 iterations. However, DryadLINQ does not efficiently support chaining the execution of jobs consist of Bulk Synchronous Parallel [20] (BSP) style tasks due to the explicitly synchronization step in BSP jobs. For example, in DryadLINQ hierarchical aggregation PageRank, the program has to resubmit a Dryad job to HPC scheduler for each synchronization step that calculate global PageRank value table. This situation is not necessary in MPI, Twister and Haloop.

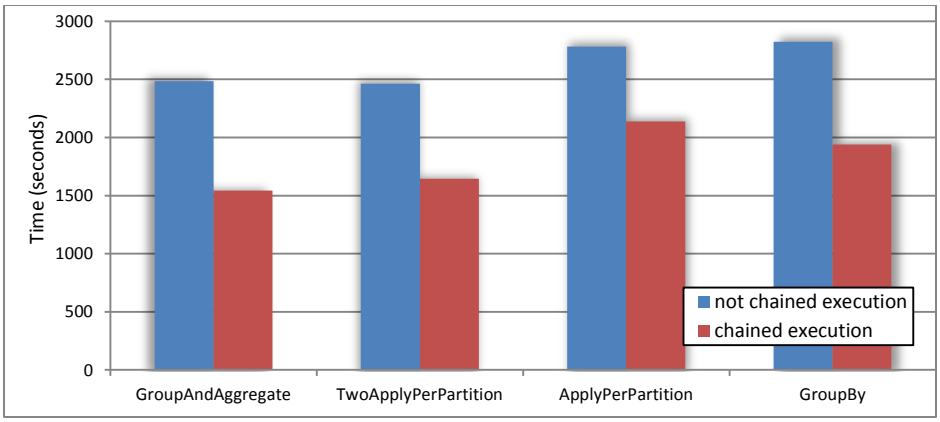


Figure 31: performance difference between chained and not chained DryadLINQ queries

5.4 Conclusion:

We investigated three distributed grouped aggregation approaches implemented with DryadLINQ CTP. Specifically, an investigation on the usability and performance of these approaches with PageRank using ClueWeb09 data were studied As a definitive argument, varying aggregation approaches work well with input data with a different range of data reduction proportion (DRP). To support this argument, a simple mathematical model was used to describe the number of aggregation operations for different approaches. For a complete analysis on the overhead of aggregation approaches, one has to take in consideration following factors: 1) The size of memory on each node. The partial pre-aggregation requires more memory than hash partition. 2) The bandwidth of network. Hash partition has larger communication overhead than partial pre-aggregation. 3) The choice of implementation of partial pre-aggregation in DryadLINQ, like the accumulator fullhash, iterator fullhash/fullsort. The different implementations require different size of memory and bandwidth. Our future job is to supplement the mathematics model with above factors to describe the timing of overhead of distributed grouped aggregation.

DryadLINQ support the iterative tasks by chaining the execution of LINQ queries in multiple iterations together. However, for BSP style applications that need explicitly evaluate LINQ query in each synchronization step, it has to resubmit Dryad job to HPC scheduler, which does harm to the overall performance. Systems, such as MPI, Twister, and Haloop can perform iterative computations by chaining the iterative task without restarting the task daemons.

6 Programming Issues in DryadLINQ CTP

Class Path in Working Directory

We found this issue when running DryadLINQ CTP SWG jobs. Dryad can automatically transfer files required by user program to remote working directory on each compute node. To save the communication and storage overhead, Dryad does not copy all DLL and shared files in each task working directory, rather it stores only one copy of shared files in the job working directory shared by all Dryad tasks. And the Dryad vertex can add the job working directory into the class path. However, when Dryad task invokes a third party executable binary file as process, the process is not aware of class path that Dryad vertex maintains, and it throws out the error about “required file cannot be found.”

Late Evaluation in Chained Queries within One Job

DryadLINQ can chain the execution of multiple queries by applying the late evaluation technology. This mechanism allows a DryadLINQ provider to optimize the execution plan of DryadLINQ queries with multiple iterations. However, a program issue was determined when executing chained DryadLINQ queries in iterative PageRank jobs. As shown in following code, the DryadLINQ query within different iteration are chained together and does not get

evaluated until the Execute() operator was invoked explicitly. The integer parameter "iterations" increase by one in each iteration. However, when applying the later evaluation, the DryadLINQ provider evaluates the "iterations" parameter at the last iteration (which is nProcess -1 in this case) and uses that value in all the queries of previous iteration. In short, the late evaluation may separate the evaluation of the parameters from that of the queries.

```
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();
```

Serialization for Two Dimension Array

We have this program issue when running the DryadLINQ CTP Matrix Multiplication and PageRank jobs. DryadLINQ provider and Dryad Vertex can automatically serialize and unserialize the standard .NET objects. However, when we defined the two dimension array object in Matrix Multiplication and PageRank application, the program will throw out error when Dryad task try to access unserialized two dimension array object on remote compute node. We look into serialization code that automatically generated by DryadLINQ provider, and found it may not be able to unserialize two dimension array objects correctly where the reason need further study.

```
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]);
   }
}
```

7 Education Session

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

In CSCI B649 Cloud Computing for Data Intensive Science, 8 master students selected topics related with Dryad/DryadLINQ as a term long project. The following were three projects completed by the end of 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 in the CloudCom2010 conference hosted in Indianapolis, IN.

In CSCI B534 Distributed Systems, in 2011 spring semester, 2 students studied Dryad/DryadLINQ as a term long project and contributed small but solid results for this report.

Concurrent Dryad jobs

From the experiences of the two courses, a few observations on how students would utilize a cluster to understand the theories and applications in large scale computing are mentioned:

1. Ordinary classes will contain 30-40 students, which can form 10 – 15 groups.
2. Student groups will not execute extremely large-scale computations and will require either 1 or 2 nodes to finish a job within an hour.
3. Students may not submit jobs until the due date is approaching. In another words, when a deadline is forthcoming, there can be many jobs in the queue waiting to be executed, other than this time the cluster may be in an idle state.

Based on the above observations, it's critical to run multiple Dryad jobs simultaneously on a HPC cluster, especially in an education setting. In the Dryad CTP, we managed by allowing each job to reside in a different node group as shown in Figure 32. 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 both the Programming and Guides.

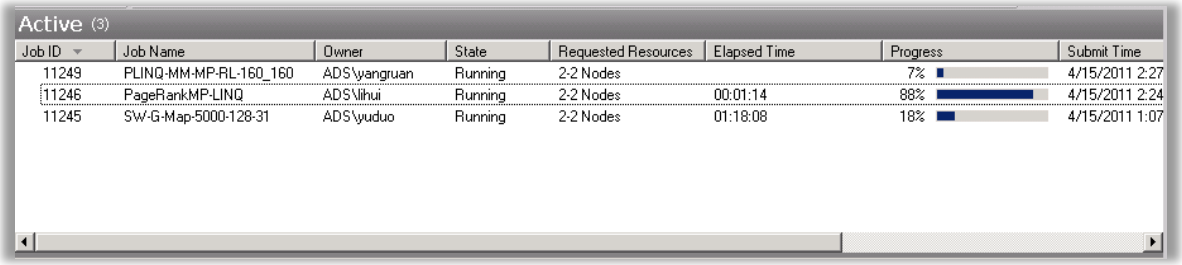


Figure 32: Concurrent Job Execution in Dryad CTP Version

On the other hand, though concurrent job running is enabled, the overall resource utilization is not perfect. Figure 33 shows the CPU usage on each node while the jobs in Figure 32 are executing. Dryad jobs are assigned to compute node STORM-CN01 through STORM-CN06. In this case, each compute node group contains 2 compute nodes, though only one of them does the actual computation. The reason is every Dryad job requires an extra node act as the job manager role. However the CPU usage of this particular node seldom exceeds 3%. This 8 nodes cluster is fully occupied by three concurrent jobs. However, the overall usage is only about 37%. A great fraction of the computation power is wasted. If the multiple job managers can reside on one same node, the resource for actual computation will increase. Also a better utilization can be achieved this way.

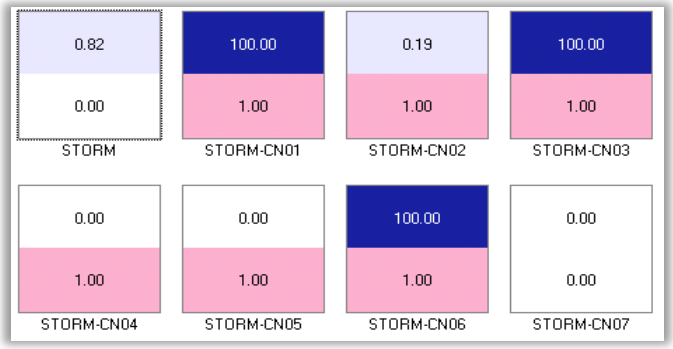


Figure 33: CPU Usage for Concurrent Job Execution

Acknowledgements

First, we want to thank John Naab and Ryan Hartman for dispensing their system administration knowledge for the STORM, TEMPEST and MADRID HPC clusters, which were critical for our experiments. Second, we thank Thilina Gunarathne and Stephen Wu for their generous ability to share the SWG application and data in Dryad (10.16.2009) report, which is important for task scheduling analysis. Also,,we thank Ratul Bhawal and Pradnya Kakodkar, two Master's students enrolled in Professor. Qiu's B534 course in Spring 2011 for their small but solid contribution to this report. This work is partially funded by Microsoft.

References:

[1] 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.

[2] 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.

[3] 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.

[4] Argonne. *MPI Message Passing Interface*. Retrieved November 27, 2010, from http://www-unix.mcs.anl.gov/mpi/.

[5] Apache Hadoop. Retrieved November 27, 2010, from http://hadoop.apache.org/.

[6] 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.

[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] 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.

[10] Batzer MA and Deininger PL (2002). *Alu repeats and human genomic diversity*. Nature Reviews Genetics 3(5): 370-379.

[11] *JAligner*. Retrieved December, 2009, from http://jaligner.sourceforge.net.

[12] Smith, T. F. and M. S. Waterman (1981). *Identification of common molecular subsequences*. Journal of Molecular Biology 147(1): 195-197.

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

[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] 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.

[17] Jaliya Ekanayake (2010). ARCHITECTURE AND PERFORMANCE OF RUNTIME ENVIRONMENTS FOR DATA INTENSIVE SCALABLE COMPUTING. School of Informatics and Computing. Bloomington, Indiana University.

[18] ClueWeb Data: http://boston.lti.cs.cmu.edu/Data/clueweb09/

[19] PageRank wiki: http://en.wikipedia.org/wiki/PageRank

[20] BSP, Bulk Synchronous Parallel http://en.wikipedia.org/wiki/Bulk_Synchronous_Parallel

[21] 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.
[22] 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.
[23] Yang, H.-c., A. Dasdan, Ruey-Lung Hsiao, D. Stott Parker. (2007). *Map-reduce-merge: simplified relational data processing on large clusters*. Proceedings of the 2007 ACM SIGMOD international conference on Management of data. Beijing, China, ACM: 1029-1040.
[24] Ekanayake, J., S. Pallickara, Shrideep, Fox, Geoffrey. *MapReduce for Data Intensive Scientific Analyses*. Fourth IEEE International Conference on eScience, IEEE Press: 277-284. 2008.

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

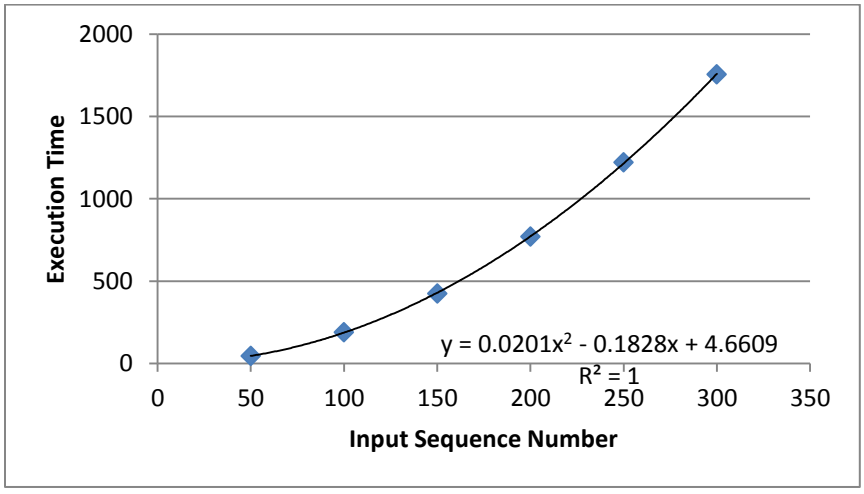
9-node homogeneous HPC cluster

	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

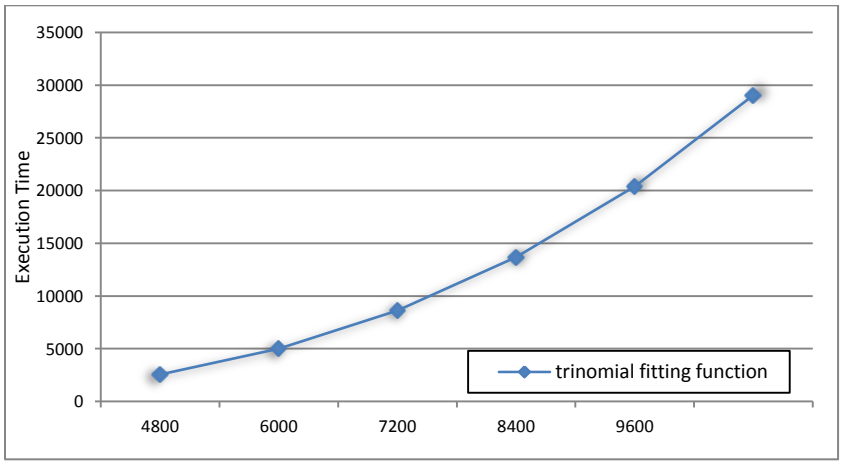
Binomial fitting function for sequential SWG jobs

$$Seq(x) = 0.0201x^2 - 0.1828x + 4.6609$$
$$R^2 = 1$$



Appendix E

Trinomial fitting chart for sequential Matrix Multiplication jobs



Appendix F

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 Dryad CTP and Dryad 2009 on same input data

Dryad 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%**
Dryad 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	79.78688	77.95291	95.89923	87.89089	90.10821	80.7075	85.47434	71.52603	74.79243