

DRYADLINQ CTP EVALUATION

Performance of key features and interfaces in DryadLINQ CTP

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

Table of Contents

Introduction	4
Overview	5
Task Scheduling.....	5
Parallel Programming Model	5
Distributed Aggregation.....	6
Task Scheduling.....	7
Introduction	7
Pairwise Alu Sequence Alignment Using Smith Waterman GOTOH	7
DryadLINQ Implementation	7
Scheduling on Homogenous Cluster	8
Task Granularity Study	8
Inhomogeneous Input Data Study	9
Scalability Test.....	11
Scheduling on Inhomogeneous Cluster	13
Task Granularity Study	13
Hybrid Parallel Programming Model	18
Introduction	18
Parallel Matrix-Matrix Multiplication Algorithms.....	18
Row Split Algorithm	18
Row/Column Split Algorithm	20
2D Block Decomposition in Fox Algorithm	21
Core Level Parallelism	24
Multi-Core Technologies.....	25
Performance Analysis in Hybrid Parallel Model.....	28
Performance in Multi Core Parallelism	28
Performance in Multi Node Parallelism.....	29
Performance in Hybrid model with different algorithms	31
Conclusion.....	34
Distributed Aggregation.....	35
Introduction	35
Distributed Grouped Aggregation Approaches	35

Hash Partition	36
Hierarchical Aggregation.....	37
Aggregation Tree.....	37
Performance Analysis:	38
Performance in Different Aggregation Strategies	38
Comparison with other implementations.....	41
Chaining Tasks within BSP Job	42
Conclusion:.....	43
Programming Issues in DryadLINQ CTP	43
Class Path in Working Directory.....	43
Late Evaluation in Chained Queries within One Job	43
Serialization for Two Dimension Array	44
Education Session	44
Concurrent Dryad jobs	45
Acknowledgements.....	46
References:	46
Appendix	47

Introduction

Applying high level parallel runtimes to data and compute intensive applications is becoming increasingly common [1]. Both industry and academia are spending more effort on the design and implementation of higher level language interfaces suitable for data intensive computation runtimes. The MapReduce programming model simplifies the processing of large scale distributed data, and attracts a lot of enthusiasm among distributed computing communities [2]. But its fixed paradigm cannot fit all application scenarios. There is booming discussion about SQL-like and non-SQL-like languages that are designed for data intensive computations, like the Pig Latin and HIVE that work on top of Apache Hadoop, offering a SQL-like programming interface. Dryad/DryadLINQ is designed for data intensive applications and it is able to address some of the limitations of MapReduce and RDBMS.

The first goal of our study is to evaluate the key features and interfaces of the DryadLINQ CTP with the focus on their efficiency and scalability. We achieve this goal by implementing three different scientific applications, which include Pairwise Alu sequence alignment, Matrix Multiplication, and PageRank with large and real-life input data. We show that: 1) task scheduling in the Dryad CTP performs better than the Dryad (2009.11), but its default scheduling plan does not fit for some applications and hardware setting, 2) porting multi-core technologies such as PLINQ and TPL to DryadLINQ task can increase the overall performance greatly, and 3) the choice of distributed aggregation strategies has an effect on performance of communication intensive applications.

Secondly, we explore several programming models for various scientific applications in the DryadLINQ CTP. For example, in pairwise ALU sequence alignment application, we show how to program pleasingly parallel applications that consist of Map and Reduce steps. In Matrix Multiplication, we explore an advanced parallel programming model that requires complex messaging control. In PageRank, we compare the implementation of distributed aggregation in MPI [3], Hadoop [4], Twister [5] and DryadLINQ[6]. This report also covers an education session that illustrates how Prof. Qiu makes use of Dryad/DryadLINQ to improve and advance teaching for graduate students in the Indiana University.

This report is structured as follows: Section 1 introduces the overview of the key features in DryadLINQ CTP we will evaluate. In section 2, we will study the task scheduling in DryadLINQ CTP with the SW-G application. Section 3 will explore hybrid parallel programming models with the Matrix Multiplication application. Section 4 covers the distributed aggregation with the PageRank application. Section 5 will talk about the issues we've found in the DryadLINQ CTP version. Section 6 is an education session that explains the applicability of DryadLINQ to education scenario. In this document, "Dryad/DryadLINQ CTP" refer to Dryad/DryadLINQ CTP version released in 2010.12; "Dryad/DryadLINQ (2009.11)" refer to the version released in 2009.11.11; Dryad/DryadLINQ refers to all Dryad/DryadLINQ version.

Overview

Dryad, DryadLINQ and DSC [5] are a set of technologies support the processing of data intensive applications on Windows HPC cluster. The software stack of these technologies is shown in Fig 1.

Dryad is a general purpose distributed runtime designed to execute data intensive applications on Windows clusters. A Dryad job is represented as a directed acyclic graph (DAG), which is called Dryad graph. The Dryad graph consists of some vertices and channels. A graph vertex is an independent instance of the data processing code in certain stage. Graph edges are channels transferring data between vertices. The Distributed Storage Catalog (DSC) is the component that works with NTFS to provide the data management functionality such as file replication and load balancing for Dryad and DryadLINQ.

DryadLINQ is a library that translates Language-Integrated Query (LINQ) programs written by .NET language into distributed computations run on top of Dryad system. The DryadLINQ API is based on LINQ programming model. It takes the advantage of standard query operators and adds query extensions specific to Dryad. The developers can apply LINQ operators such as Join, GroupBy to a set of .NET objects, which greatly simplify the developing of data parallel applications.

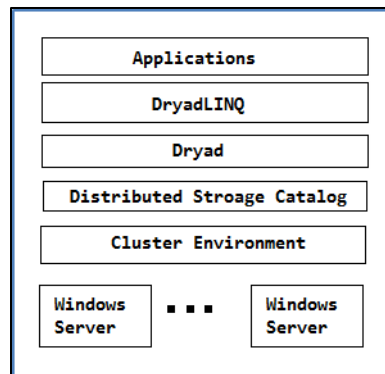


Fig.1 Software Stack for DryadLINQ CTP

Task Scheduling

The Dryad CTP takes more aggressive optimization scheduling strategy than Dryad (2009.11). We draw this conclusion by comparing the performance of skewed distributed input data between Dryad CTP and Dryad (2009.11). The DryadLINQ CTP programmers can leverage this advantage by using the Distributed Storage Catalog (DSC) and related APIs such as AsDistributed(), RangePartitions() without the need to generate partition file as in Dryad (2009.11). Besides, we studied the efficiency and scalability of DryadLINQ CTP Pairwise ALU sequence alignment jobs on a medium-sized HPC cluster, whose results were reasonably good. However, we also found the default scheduling plan does not show good performance for some applications and hardware scenarios. We have designed experiments to find out the reason and explored the possibility to improve these issues.

Parallel Programming Model

Dryad job execution plan is Directed Acyclic Graph(DAG) based which supports many application scenarios. However, Dryad cannot maintain the intermediate state during computation like MPI, which is important for applications that require complex messaging control such as 2D decomposition Matrix Multiplication in Fox algorithm [7]. We explored the applicability of DryadLINQ CTP to several complex parallel programming models for Matrix Multiplication and compared the performance across each other. Furthermore, we implemented a hybrid parallel programming model by porting multi-core technologies in .NET such as PLINQ and TPL to Dryad tasks. Our

experiment results showed that generally this strategy can improve the overall performance significantly for some applications, though performance difference exists while different multi-core technologies are used.

Distributed Aggregation

The GROUP BY operator in parallel database is often followed by the Aggregate functions. It groups the input records into some partitions by keys, and then merges the records for each group by certain attribute values. This common pattern is called Distributed Grouped Aggregation. Sample applications of this pattern include the sales data summarizations, the log data analysis, and social network influence analysis.

MapReduce and SQL in database are two programming models that can perform grouped aggregation. MapReduce has been applied to process a wide range of flat distributed data. However, MapReduce is not efficient to process relational operations which have multiple inhomogeneous input data stream like JOIN. The SQL queries are able to process relational operations of multiple inhomogeneous input data stream. But, the operations in full-feature SQL database has lots of extra overhead which prevents application from processing large scale input data.

DryadLINQ lies between SQL and MapReduce, and it addresses some limitations of SQL and MapReduce. DryadLINQ provides developers SQL like queries to process efficient aggregation for single input data stream and multiple inhomogeneous input stream, but it does not have much overhead as SQL by eliminating some functionality of database (transactions, data lockers, etc.). Further Dryad can build the aggregation tree (some database also provide this kind of optimization) so as to decrease the data transformation in hash partitioning stage. This paper investigates the usability and performance of distributed grouped aggregation in DryadLINQ and compares it with other runtimes including: MPI, Hadoop, Haloop[8], and Twister for the PageRank application.

Task Scheduling

Introduction

Pairwise Alu Sequence Alignment Using Smith Waterman GOTOH

The Alu clustering problem [9][10] is one of the most challenging problems for sequencing clustering because Alus represent the largest repeat families in human genome. There are about 1 million copies of Alu sequences in human genome, in which most insertions can be found in other primates and only a small fraction (~ 7000) are human-specific. This indicates that the classification of Alu repeats can be deduced solely from the 1 million human Alu elements. Notable, Alu clustering can be viewed as a classical case study for the capacity of computational infrastructures because it is not only of great intrinsic biological interests, but also a problem of a scale that will remain as the upper limit of many other clustering problem in bioinformatics for the next few years, e.g. the automated protein family classification for a few millions of proteins predicted from large metagenomics projects.

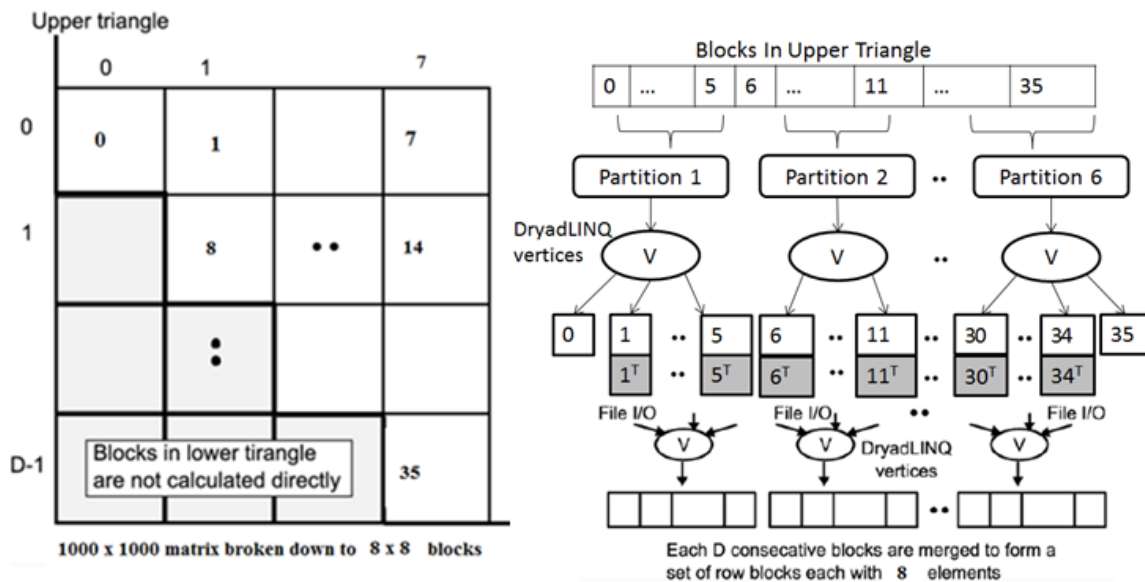


Figure 1: Task Decomposition (left) and the Dryad Vertex Hierarchy (right) of the DryadLINQ Implementation of SW-G Pairwise Distance Calculation Application

We used an open source version, NAligner [11], of the Smith Waterman – Gotoh algorithm (SWG) [12][13] to ensure low start up effects by each task process large numbers (more than a few hundred) at a time. Memory bandwidth needed was reduced by storing data items in as few bytes as possible.

DryadLINQ Implementation

The SWG is a pleasingly parallel application, which consists of two steps: map and reduce. In the map stage, the input data is split into certain blocks which are processed by SW-G program. The calculation of blocks is independent without communication between compute nodes. In the reduce stage, it collects outputs from map tasks and generates the final result. To clarify our algorithm, let's consider an example with 10,000 gene sequences, which produces a pairwise distance matrix of size 10,000 × 10,000. We decompose the computation task by considering the resultant matrix and group the overall computation into a block matrix D of size 8 × 8, each block contains 1250 × 1250 sequences in this case. Due to the symmetry of the distances $D(i,j)$ and $D(j,i)$, we only calculate the distances in the 36 blocks of the upper triangle of the block matrix as shown in Figure 1 (left).

Assuming the blocks in the upper triangle are divided into 6 data partitions, therefore each partition contains 6 blocks. Then the partitions will be distributed to available compute nodes and an ApplyPerPartition operation is used to execute a function to calculate the distances within each block. After computing the distances in each block, the function calculates the transpose matrix of the result matrix which corresponds to a block in the lower triangle, and writes both matrices into two output files in local file system. The names of these files and their block numbers are communicated back to the main program. The main program sorts the files by block numbers and performs another “ApplyPerPartition” operation to combine the files corresponding to a row of blocks in to a single large one as shown in the Figure 1: Task Decomposition (left) and the Dryad Vertex Hierarchy (right) of the DryadLINQ Implementation of SW-G Pairwise Distance Calculation Application. The pseudo code for our implementation is as follow:

Map stage:

```
DistributedQuery<OutputInfo> outputInfo = lists.AsDistributedFromPartitions()
    .WithConfiguration(config).ApplyPerPartition(sublist =>
        PerformAlignments(sublist, _inputFile, _sharepath, _outputFilePrefix,
            _outFileExtension, _seqAlignerExecName, _swgExecName))
    .OrderBy(x => x.Index);
```

The user defined PerformAlignments function will process multiple SW-G blocks within the same partition. In order to make use of the multi-core computation power in each compute node, we create a thread pool to spawn parallel threads. The number of threads is equal to the number of cores in each compute node. Each thread will launch one Operating System (OS) Process to calculate one SW-G block input data at a time. The SW-G experiments in this chapter always make use of this multi-core optimization method, if not claimed otherwise.

Reduce stage:

```
var finalOutputFiles = swgOutputFiles.AsDistributed().WithConfiguration(config).
    ApplyPerPartition(tasks => PerformMerge2(tasks, _dimOfBlockMatrix,
        _sharepath, _mergedFilePrefix, _outFileExtension));
```

Scheduling on Homogenous Cluster

We performed a set of experiments to evaluate the scheduling mechanism and scalability of the DryadLINQ CTP version by running SW-G jobs with different size of input data and number of compute nodes. With the AsDistributedFromPartitions or HashPartition operator, we can manually assign desired amount of workload (SW-G blocks) to each partition, which makes it ideal to track how input data is assigned and if the load on each node is balanced.

Based on the results, the job scheduling mechanism of the DryadLINQ CTP version gives a fairly good performance in job turnaround time and scalability. Besides, its dynamic scheduling mechanism can achieve a balanced workload most of time, which is proved to be better than Dryad (2009.11) for skewed distributed input data in SW-G jobs. The result also shows that as the number of partitions increases, the scheduling cost grows significantly. And its default scheduling plan is sub-optimal for inhomogeneous clusters, which we will illustrate in later paragraphs.

Task Granularity Study

We submitted SW-G jobs on a 32 compute nodes homogeneous HPC cluster named TEMPEST (Appendix B). In this experiment, we executed a set of SW-G jobs with the input length of 10000, and block matrix dimension of 128. We use the AsDistributedFromPartitions() to manually split the input data (8256 SW-G Blocks) into the following

partitions set {31, 62, 93, 124, 248, 372, 496, 620, 744, 992}. As both the compute nodes and input data are homogeneous, workload balancing will not be a severe impact to the performance. As shown in Figure 2, the 31-partition delivers the best performance. Since there are 32 compute nodes in total and Dryad job manager will take one for dedicated usage, leaving only 31 nodes for the actual computation. As it shown in Figure 2 the job turnaround time increases as the number of partition increases. Due to two reasons: 1) the scheduling cost increase as the number of Dryad tasks increase. 2) The partition granularity become finer as number of partitions increases. When number of partitions is more than 372, each partition has less than 24 blocks, which cannot make full utilization of multi-core in each compute node.

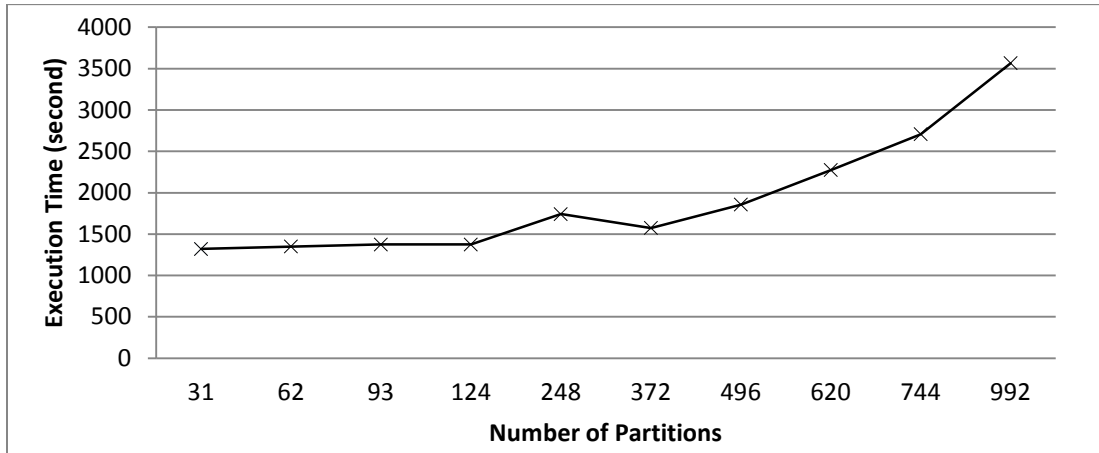


Figure 2: Execution Time Chart for Different Number of Partition on TEMPEST

Table 1: Execution Time Table for Different Number of Partitions on Tempest

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

Inhomogeneous Input Data Study

The input gene sequences for SW-G tasks can be inhomogeneous in length, which can cause imbalanced workload distribution during the task scheduling. The Dryad (2009.11) adapted the static scheduling strategy by writing the tasks execution plan into partition files, which is not suitable for skew distributed input data [1]. We study the same issue in Dryad CTP to see whether it has been solved or alleviated.

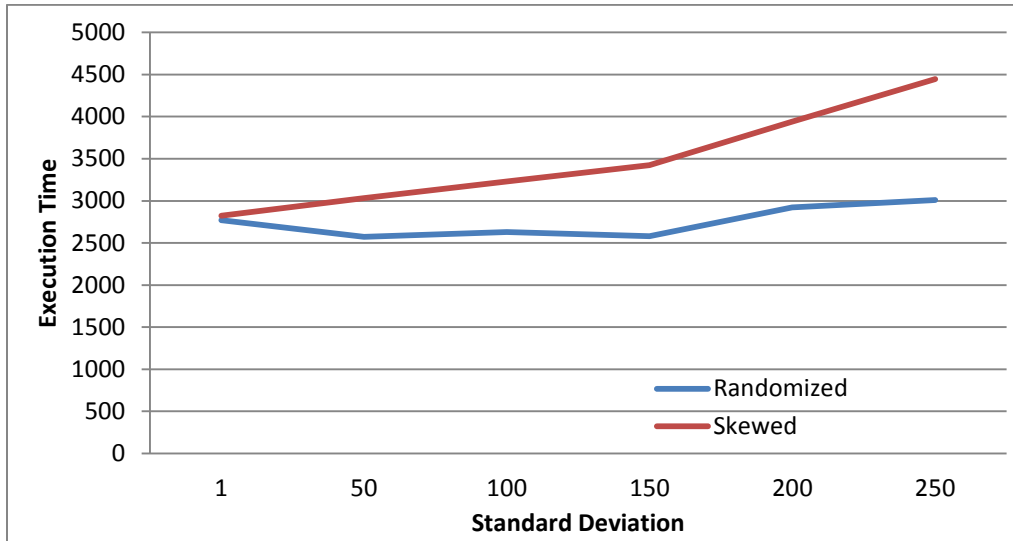


Figure 3: SW-G Execution Time for Skewed and Randomized Distributed Input Data

Table 2: Execution Time for Skewed and Randomized Data

StdDev	1	50	100	150	200	250
Skewed 1	2737	2986	3264	3449	4006	4404
Skewed 2	2910	3076	3198	3396	3878	4488
Average	2823.5	3031	3231	3422.5	3942	4446
Random 1	2803	2592	2616	2608	2793	2989
Random 2	2740	2552	2647	2554	3054	3028
Average	2771.5	2572	2631.5	2581	2923.5	3008.5

We executed a bunch of SW-G jobs on the TEMPEST with input of 10000 sequences. The data sets used were randomly generated with a given mean sequence length (400) with varying standard deviations following a normal distribution of the sequence lengths. We constructed the SW-G blocks input data by randomly selecting sequences from above data set as well as by selecting in a sorted order based on the sequence length. As it shown in Figure 3, the randomly distributed input data can deliver a better performance than skew distributed input data, which is already proved in the Dryad (2009.11) report. Since sequences are sorted by length in skewed scenario, therefore the block sizes are hugely varied among every block. On the other hand, random ordered sequences are expected to give a more balanced workload distribution. Therefore an overall better performance is delivered from random ordered input.

As it shown in Figure 4, in Dryad CTP, the performance gap between skew and randomly distributed input data is not as obvious as the one in Dryad (2009.11). The reason is that in Dryad CTP, it keeps dispatching SW-G blocks partitions to idle compute resources. Furthermore, increasing the number of partitions can further alleviate the workload imbalance issue caused by skew distributed input data.

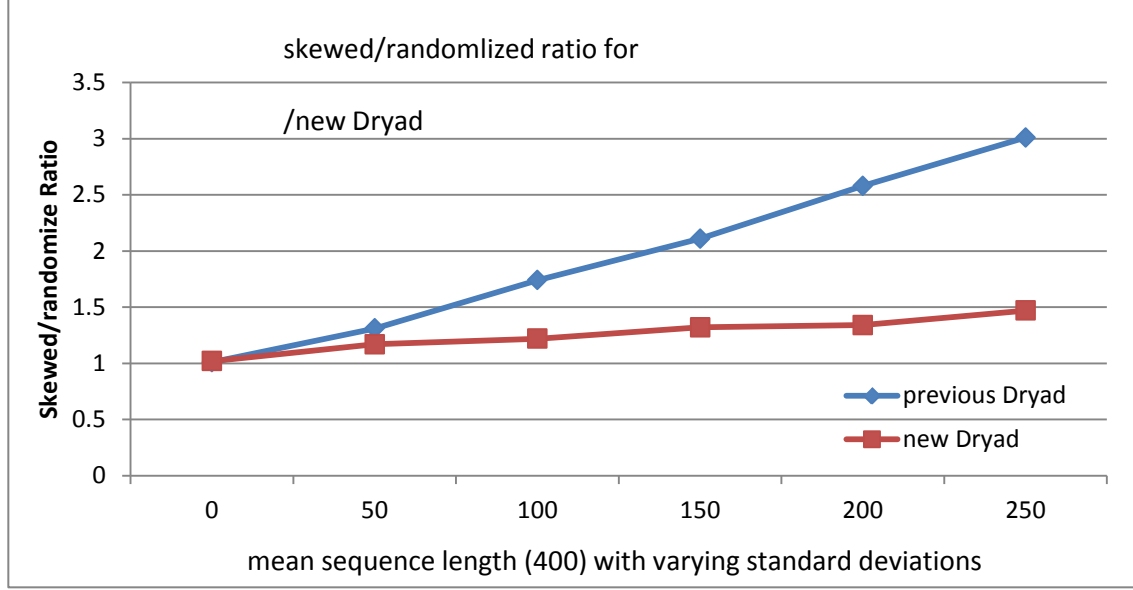


Figure 4: Comparison between Dryad CTP and Dryad (2009.11) in Skewed Data Scheduling

Scalability Test

In this section, we examined the scalability of the Dryad CTP on Tempest Cluster by executing two series of experiments. The first one is to calculate the relative parallel efficiency for different size of input data on 32 compute nodes. The other one is to get the relative speed up for different number of compute nodes.

In the first set of experiments, the input includes gene sequences range from 5,000 to 10,000, with a step length of 2500. The block matrix size is 128×128 , and the number of partition is fixed to 31, which is the optimal value we found in previous experiment. We executed the experiments with same parameters on Tempest Cluster. The result is shown in Figure 5. In the figure, the red line represents to the executions time on 31 compute nodes; the green line represents the execution time on only 1 compute node; and the blue line is the relative parallel efficiency defines as the following:

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

We can observe as the input size increases, naturally the execution times on both scenarios increase. Moreover, the efficiency is over 90% for most of time. This is a considerable good result. To better illustrate the relation between two experiment results, we also included parallel efficiency notion to show how efficient can Dryad CTP achieve by parallel job executing.

The parallel efficiency is getting better as the input size rose according to the experiment. When the input size is 5000, which is a small input for a 32-node cluster, the execution time is about 550 seconds. The cluster is underutilized in this case as job scheduling costs a big portion of the total execution time. However, as the input size keeps increasing, the CPU time leaps, so the scheduling cost becomes less critical to the entire calculation. Hence the efficiency is becoming higher as the workload becomes heavier. In this experiment, the parallel efficiency jumps from 81.23% to 96.65%, this is a noticeable performance.

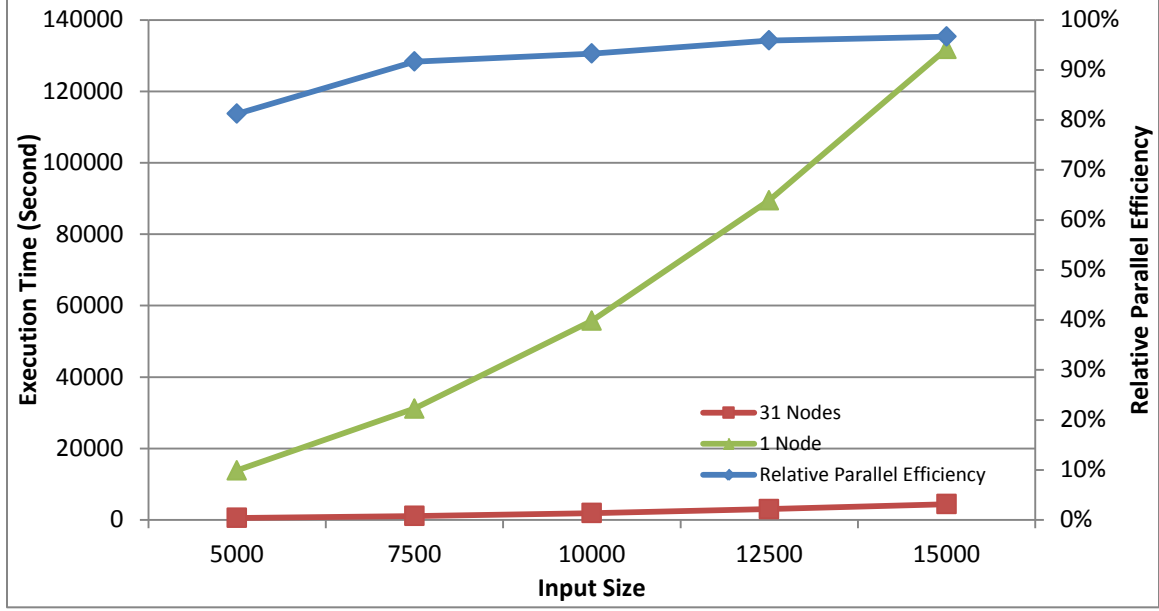


Figure 5: Performances and Parallel Efficiency on Tempest

Table 3: Average Execution Time of Tempest

Cluster	Input length				
	5000	7500	10000	12500	15000
1 Node	13854.71	31169.03	55734.36	89500.57	131857.4
32 Nodes	550.255	1096.925	1927.436	3010.681	4400.221
Parallel Efficiency	81.22%	91.66%	93.28%	95.90%	96.66%

In the second set of experiments, we calculate the relative speed up of DryadLINQ CTP SW-G on Tempest with different number of compute nodes. We chose 10,000 input sequences, 128×128 block matrix, and 31 partitions. The compute nodes involved are 2, 4, 8, 16, and 31 (due to the cluster limitation, we could not extend to 32 compute nodes). The result is shown in Figure 6 and Table 4. The relative speedup used in Figure 6 is defined as following:

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

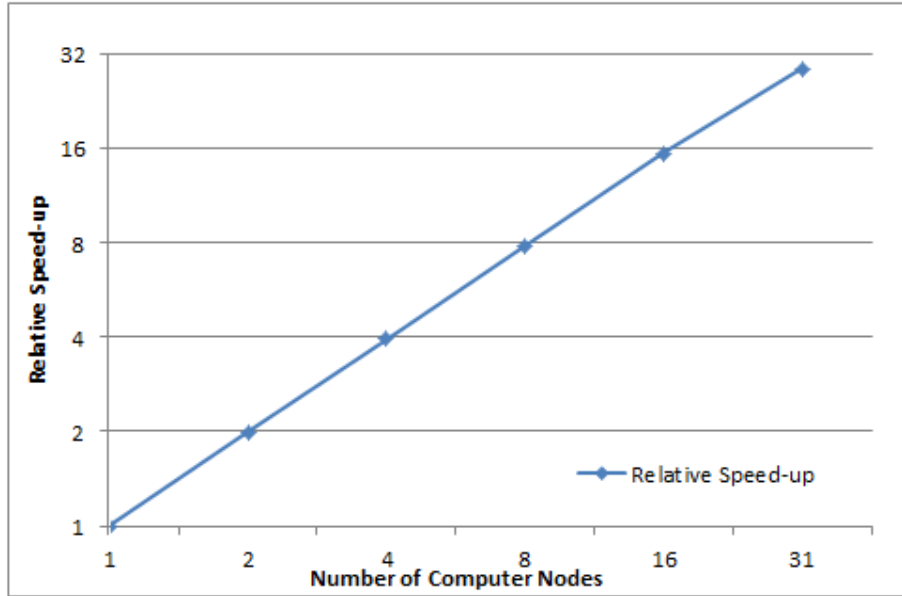


Figure 6: Relative Speed-up for SW-G on Tempest with Varied Size of Compute Nodes

Table 4: Execution Time and Speed-up for SW-G 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

From Figure 6, we can see the speed-up is linear regarding to the number of compute nodes. In table 4, the execution time is also decreasing as more nodes involved, also in a linear trend. The running time across 8 nodes is almost exactly as twice much as the one across 16 nodes. The likewise relationship also exists between other pairs like between 2 nodes and 4 nodes, and so on.

Base on the two series of experiments above, we can see performance of the Dryad CTP doesn't degrade as the computation grows. The execution time is increased linearly as the size of computation boosts. The scale of the experiments is between 40 minutes to 2 days. We can then draw a conclusion that the DryadLINQ CTP SW-G is stable and scales well in a medium sized HPC cluster with 32 nodes.

Scheduling on Inhomogeneous Cluster

The Dryad assumes the hardware resources are homogeneous when it generates the job execution plan, which is not true in some case. The inhomogeneous cluster is becoming popular, as its hybrid hardware resources can deliver better performance/cost ratio for wide range of applications. Besides, even the inhomogeneous hardware resources may deliver inhomogeneous real-time performance in CPU and network. Hence we want to focus on the Dryad CTP scheduling on an inhomogeneous HPC cluster we own.

Task Granularity Study

In this experiment, we executed a set of SW-G experiments on an inhomogeneous HPC cluster named STORM (Appendix A). The parameters for SW-G jobs in experiments are: sequence length of 2048, block matrix dimension of 64. We split all the SW-G input blocks with `AsDistributedFromPartitions()` into following partitions set: {6, 12, 24, 36, 48, 96, 144, 192, 384}

In STORM, as the hardware resources are inhomogeneous in CPU and memory, which make itself an ideal test bed to study workload balance issue in Dryad CTP. In the experiments, as some nodes are twice or three times faster than others, the partitions dispatched to faster compute nodes are finish earlier than the slow ones. At the end of computation, the faster nodes have to stay idle and wait tasks running on slow node to finish. We try to alleviate this issue by splitting the SW-G jobs into many finer partitions, and make Dryad CTP job manager keep dispatching partitions to idle resources. Based on results, some arguments can be made:

- The workload is distributed more balanced, as the number of data partitions increases.
- The execution time increases if the partition number increases beyond a certain point.
- The key point for the best performance is to find the appropriate partition number.

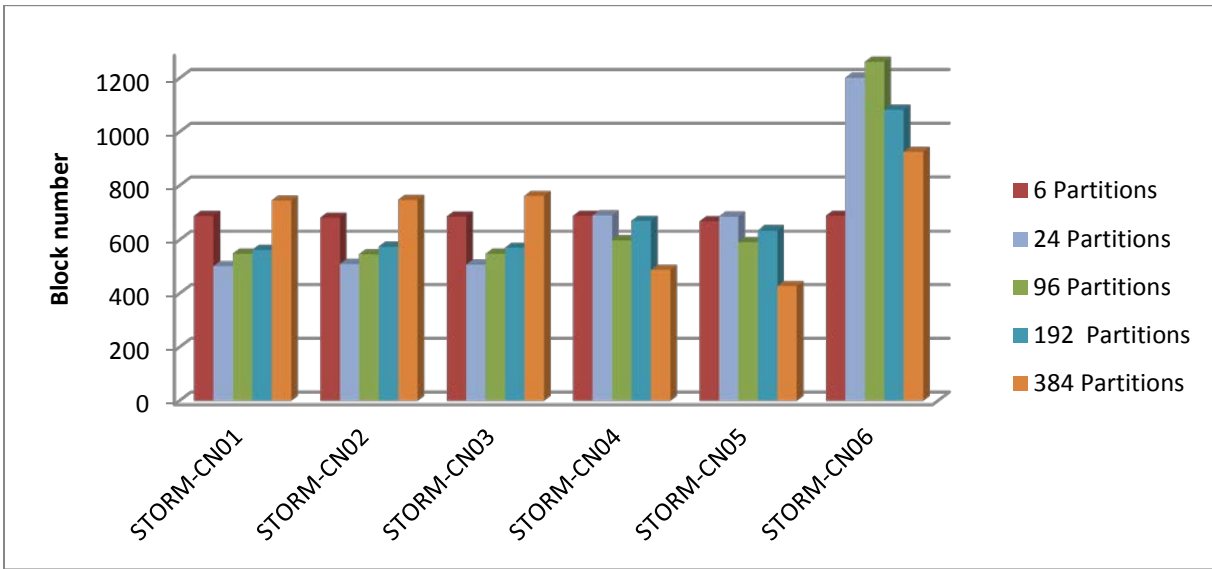


Figure 7: SW-G blocks distribution across inhomogeneous compute nodes

As it shown in Figure 7, the bigger the number of partitions is the better the workload distribution will be fit with inhomogeneous computing powers of STORM. The default partition scheme, twice as the number of compute nodes 12, may not deliver the best distribution in workload balance.

Table 5: Block Number for Each Compute Node in 6 Experiments

Node Name	Partition number						
	6	24	48	96	144	192	384
STORM-CN01	687	502	502	549	500	563	745
STORM-CN02	681	510	423	547	504	575	747
STORM-CN03	685	508	511	548	493	571	762
STORM-CN04	688	689	775	599	846	669	488
STORM-CN05	667	685	679	592	791	635	428
STORM-CN06	688	1202	1206	1261	962	1083	926
Standard Deviation	7.41	269.3382	262.02	259.52	190.48	183.08	171.41

Since Dryad CTP is doing dynamic scheduling, it's quite logical that more data partitions will lead to a more evenly workload distribution. With adequate data partitions, the job manager can assign a new partition to a compute node that finishes before others. Therefore every node will finish in a similar time. However, on the other hand, the operation to warp up finished data and retrieve new data is relatively time consuming. If data is divided overly scattered, the advantage we gain from a balanced distribution will be surpassed by the overwhelming scheduling cost, which could double the total execution time in our experiment.

Performance analysis with different partition granularities

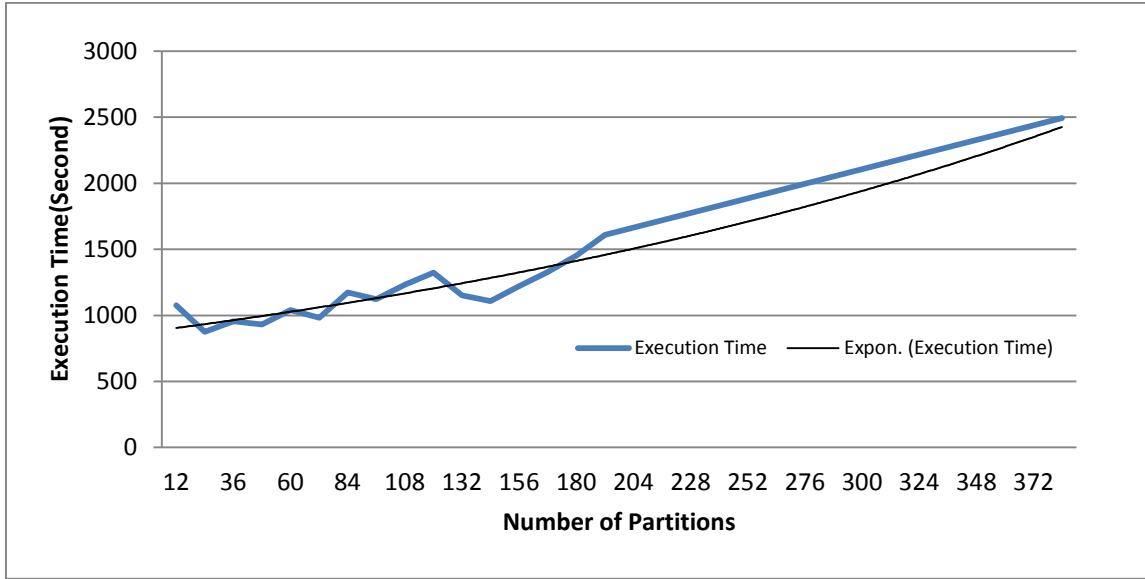


Figure 8: SW-G job Turnaround Time for Different Partition Granularities

We executed the 4096 sequences SW-G jobs on STORM with different number of partitions. As it shown in Figure 8, it's obvious that the execution time is doubled with 384 partitions comparing to only 6 partitions. The reason is as we mentioned above: the time cost for scheduling 384 partitions is increased and the pure computation time remains the same (4096 sequences in total). Therefore the execution time increases as the number of partitions grows.

Except for large amount of partitions, the execution time fluctuates when the number of partitions is relatively small. This is because of the workload balancing issue. Since the experiment is running on a non-uniform cluster, some compute nodes are faster than others. Therefore distributing all absolute evenly to each compute node will not give the best performance. The faster nodes will do nothing other than waiting for slower nodes to finish. This is why dividing data into 6 partitions doesn't return the best performance. Dividing data into more partitions will provide a more balanced distribution. Nevertheless, it will also trigger more scheduling cost. Thus the critical point for achieving the best performance is to divide data into appropriate number of partitions which brings a balance between workload balance and scheduling cost. In this experiment, it should be 3 or 4 times of the number available compute node. If the actual partition number is less than this, the workload is not balanced so the faster nodes will stay idle after finishing their job; in the other way, the scheduling cost for handling more partitions will considerably slow the whole computation down.

To better describe this phenomenon, the start and end time for processing each partition is showed in the following figures. In Figure 9, 10, and 11, x-axis stands for the elapsed time in seconds since the calculation started. Each value along y-axis presents one compute node. In the plot area, each blue point means a calculation starts at

this exact time on this corresponding compute node. Similarly, a red point means one calculation finishes at this time on this node. Through Figure 9, 10, and 11, the busy/idle status of each node is clearly represented.

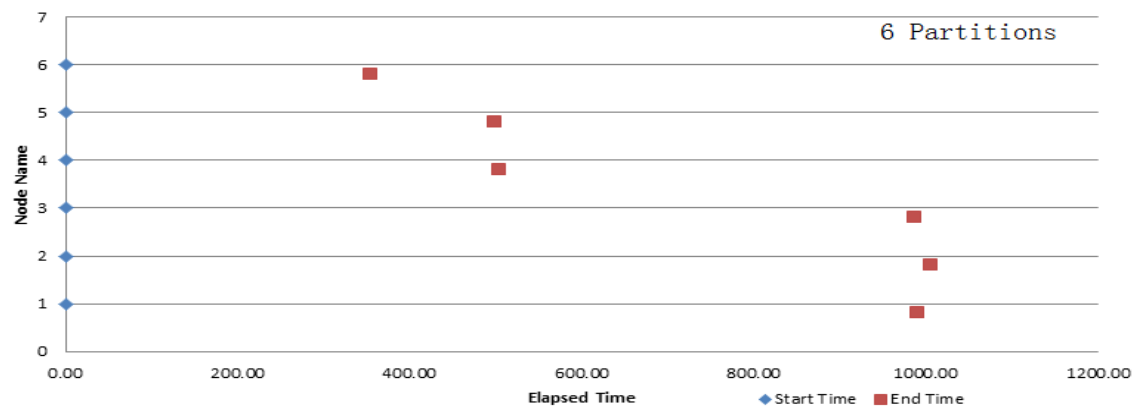


Figure 9: Time Log for 6 Partitions in SW-G

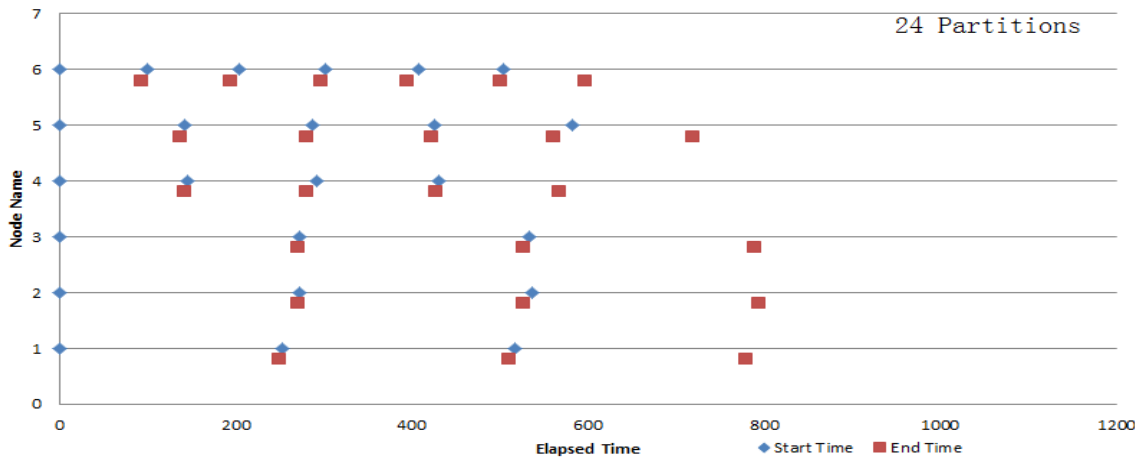


Figure 10: Time Log for 24 Partitions in SW-G

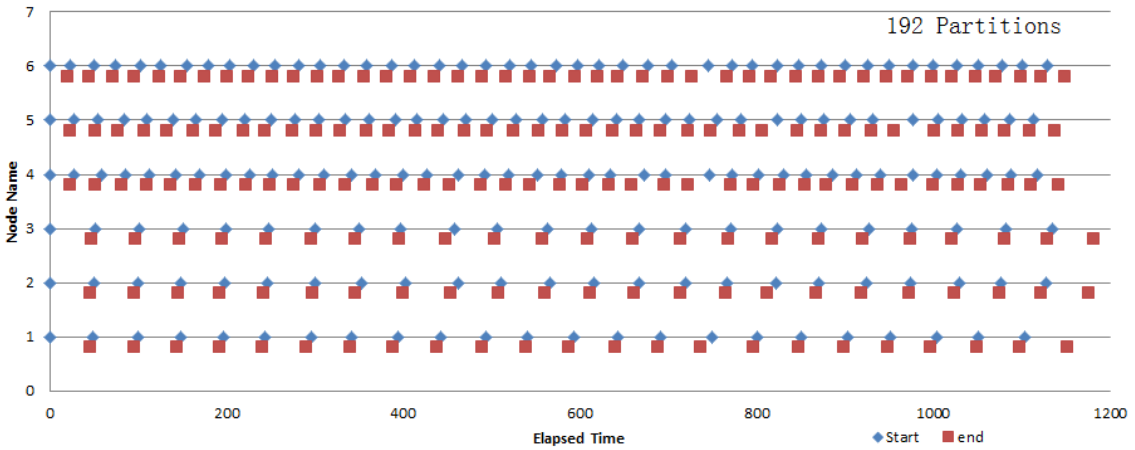


Figure 11: Time Log for 192 Partitions in SW-G

In the 6 partitions scenario, every node was assigned with one equally sized partition. Node 6 has 24 cores but node 1, 2, and 3 only have 8 cores each. Naturally node 6 will finish the calculation first. Without extra data partitions, node 6 could only stay idle until the whole job is finished. This is a highly unbalanced workload distribution.

In the 24 partitions scenario, node 6 didn't stay idle after finished its first partition. It retrieves another partition and started calculating while other nodes were still processing their first partitions. At the end, node 6, which has 24 cores, processed 7 partitions while node 4 and 5, each has 16 cores, processed 4 partitions respectively, and node 1, 2, and 3 only processed 3 partitions each. Although the partitions were not evenly across nodes, they finished calculation almost in the same time. Besides, handling 24 partitions didn't take up too much time. This experiment was an efficient combination of both balanced workload and fast execution.

The 192 partitions experiment showed another trend. Although the workload is distributed very balanced, the total schedule cost for all 192 partitions outweighed the advantage gained by balancing load. Dryad CTP was busy handling the start and wrap up for the 192 partitions. The actual CPU usage across the cluster was considerably lower than the first two scenarios.



Figure 12: Cluster CPU Usage for Different Partition Numbers

It's obviously in Figure 12, the overall CPU usage drops significantly as the number of partitions increases. On the other hand, too few partitions will harm the load balance therefore also slow down the performance. So it's very necessary to decide an appropriate number of partitions for any application in Dryad CTP.

Inhomogeneous aware scheduling

We have illustrated that the default Dryad CTP partition granularity may not be suitable for the inhomogeneous cluster. To solve this issue, we write the following code which try to generate partition scheme that considering the inhomogeneous compute nodes with the Storm case

```
int numPartition = 6;
int numElements = 1280;
int numTotalCores = 8 + 8 + 8 + 16 + 16 + 24;
// 6 inhomogeneous compute nodes in reference cluster
int[] numCoresStorm = newint[] { 8, 8, 8, 16, 16, 24 };
int[] accumulateCores = newint[numPartition];
int[] rangPartitionSeperator = newint[numPartition-1];

accumulateCores[0] = numCoresStorm[0];
for (int i = 1; i < numPartition; i++)
```

```

        accumulateCores[i] = accumulateCores[i - 1] + numCoresStorm[i];
for (int i = 0; i < numPartition - 1; i++)
    rangPartitionSeperator[i] = (int)(accumulateCores[i] * numElements /
numTotalCores);

```

The above partition scheme consider the inhomogeneous compute nodes, however, this partition scheme is not supported in Dryad CTP, as it does not provide interface that allow user assign the tasks to specific compute nodes. The Dryad (2009.11) supports the static scheduling by writing the job execution plan into partition file, which maybe still useful when dispatching homogeneous tasks like Matrix Multiplication, parameters sweeping on inhomogeneous cluster. Besides, other job execution like, the Condor support both dynamic and static scheduling by providing various job execution scenarios and “match maker” mechanism.

Hybrid Parallel Programming Model

Introduction

Dryad job execution plan is a DAG which supports large application scenarios. However, Dryad cannot maintain the intermediate state during computation like MPI, which is important for applications that require complex messaging control such as 2D decomposition Matrix Multiplication in Fox algorithm. We explored the applicability of DryadLINQ to several complicated parallel programming model in Matrix Multiplication and compare the performance on them. Further, we explored the hybrid parallel programming by porting multi-core technologies in .NET such as PLINT, TPL[14] to Dryad tasks. Our experiments results show that though there is some performance difference when using different multi-core technologies, but in general this strategy can improve the overall performance significantly for some application.

The Matrix-matrix multiplication is a fundamental kernel [15][16], one which can achieve high efficiency in both theory and practice. The matrix multiplication is basically $A * B = C$ where we have matrix A and matrix B as input matrixes and matrix C as result matrix. The equation for the calculation is:

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

The p in Equation 3 is the number of columns in A Matrix and number of rows in B Matrix. The Matrices in the experiments are square Matrix by default in this paper. The Matrix Multiplication 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.

Parallel Matrix-Matrix Multiplication Algorithms

We implemented DryadLINQ Matrix-Matrix Multiplication with three different Algorithms: 1) row split algorithm, 2) row/column split algorithm, 3) 2 dimension block decomposition split in Fox algorithm [12]. We make the performance comparison between them for different size of input data. Besides, we analyze the computation cost of serialization of input data with DryadLINQ provider and DSC.

Row Split Algorithm

In this model, we split the A matrix by rows, the program scatters the split rows of A matrix onto compute nodes. The whole B matrix is broadcasted or copied to every compute node. Each Dryad task multiply the row blocks of A

Matrix by whole B matrix, and retrieve the output results to user main program to aggregate the complete output C matrix. The program flow of the Row Split Algorithm is shown in Figure 13.

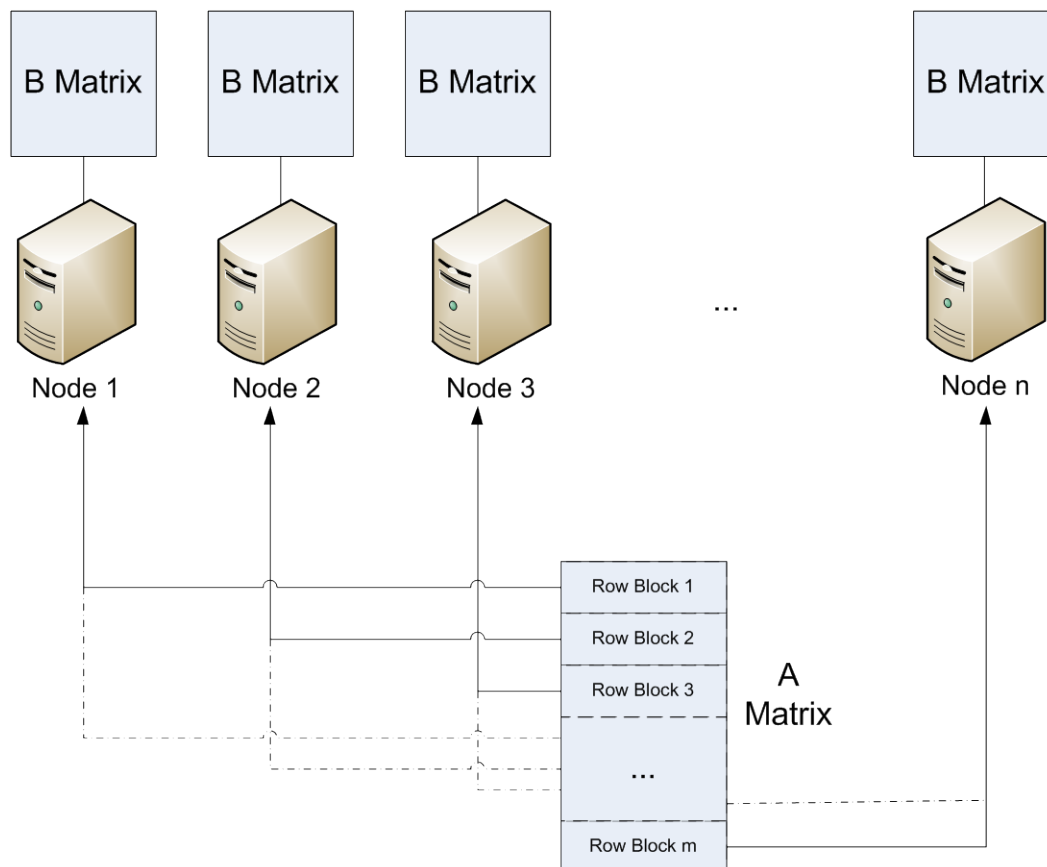


Figure 13: Row Split Algorithm

We split the A Matrix by rows, which can be stored as both DistributedQuery and DistributedData objects defined in DryadLINQ. In the first method, we write a readBlockFromAMatrix which load A Matrix data to memory, and store the row blocks in A Matrix as DistributedQuery objects which are serialized and transferred to remote compute nodes by DryadLINQ provider. In the second method, we split A Matrix into several partition files, and store their network path into DSC. During computation, the real data will be loaded into memory by reading these files through network file system.

Stage in input data with DryadLINQ provider

```
DistributedQuery<blockWrapper> inputs = readBlockFromAMatrix(rowFiles).AsDistributed();
```

Stage in input data with DSC

```
DistributedData<string> inputs = Directory.GetFiles(rowFiles, "*.txt", SearchOption.AllDirectories).AsDistributed();
```

After the partition, we use ApplyPerPartition operator to invoke user-defined function rowsXcolumnsMultiCoreTech to perform the sub task calculation. In the DSC model, we the compute node will get the file names for each input block and read the matrix remotely. We've studied the partition granularity issue in task scheduling chapter. For Matrix Multiplication application, as the sub tasks within Row Split Algorithm are homogeneous in CPU time, the ideal partition number equals to the number of compute nodes.

```
var results = inputs.ApplyPerPartition<rowPartition, double[]>(rows => rowsXcolumnsMultiCoreTech(rows));
```

Row/Column Split Algorithm

The Row/Column split algorithm was brought up in [17]. It splits both A matrix and B matrix. The A matrix is split by rows and the B matrix is split by columns. The whole computation consists of several iterations. In the initial stage, we scatter the column block of B matrix onto each compute node. In each iteration, we broadcast row block of A matrix to each compute node. One Dryad task within each iteration multiply the row block of A matrix by the column block of B matrix. The output of Dryad tasks within the same iteration will be retrieved to the main program to aggregate one row block of C matrix. The main program collects the row block results of C matrix in multiple iterations to generate final output C matrix.

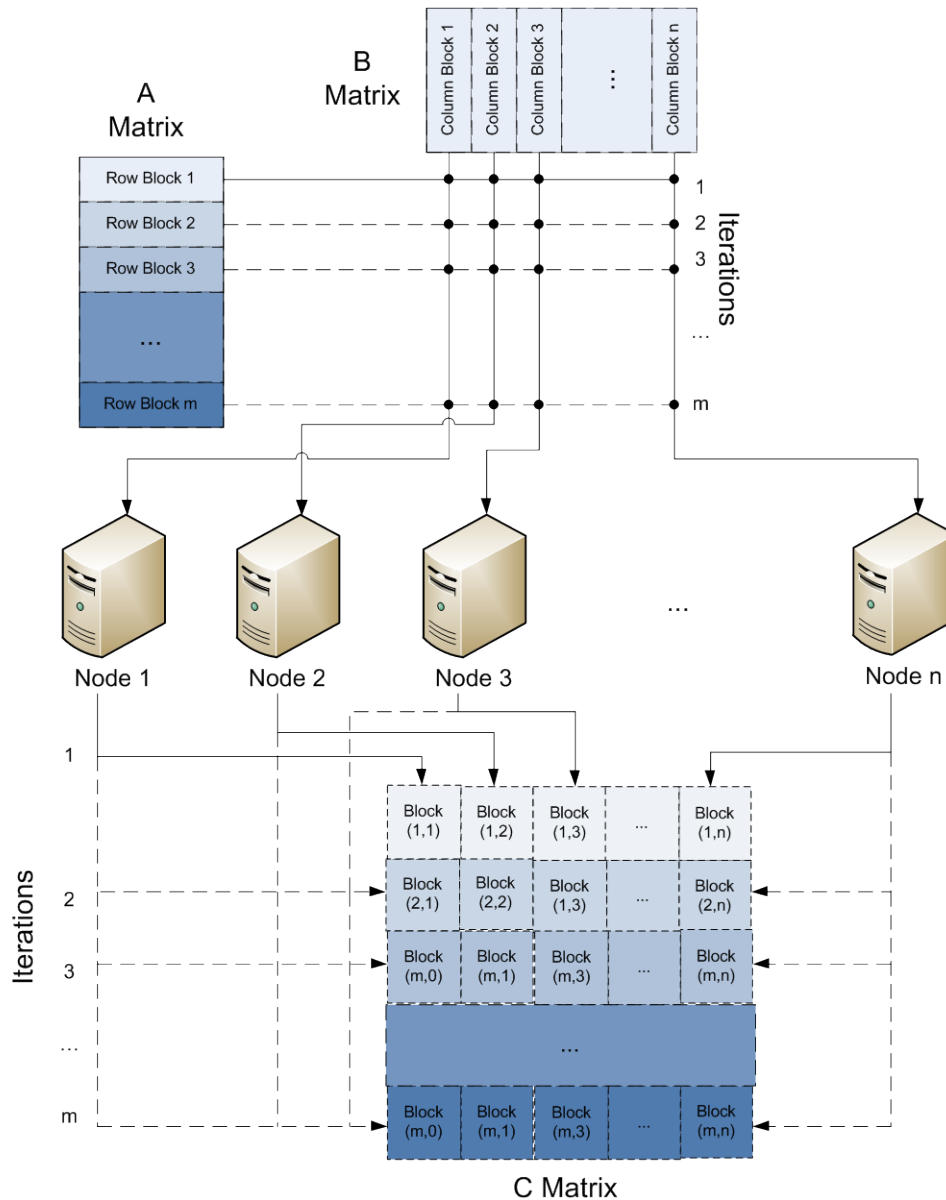


Figure 14: Row Column Split Algorithm

In the implementation, the number of partitions of row blocks of A matrix equals to number of iterations in the computation. And the number of partitions of column blocks of B matrix equals to number of compute nodes used in computation. The DistributedQuery objects of columns blocks of B matrix are distributed across the compute nodes first. Within each iteration, we use the ApplyPerPartition operator to invoke the user-defined function aPartitionMultiplybPartition to multiply the one column block in B Matrix and one row block in A Matrix.

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

```
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 Fox algorithm uses a 2 dimension block decomposition approach with a square processes mesh. Let us assume the total number of processes available is 4. This leads to a processes mesh of 2X2. Accordingly, both the A matrix and the B matrix are split into a block mesh of 2X2 respectively. In the Fox algorithm, each 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 l with A(i, (i+k)mod s) broadcasts it to all other processes l the same row i.
- 2) Processes in row l receive A(l, (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:
$$B(l,j) \leftarrow B((i+1) \bmod s, j)$$

End

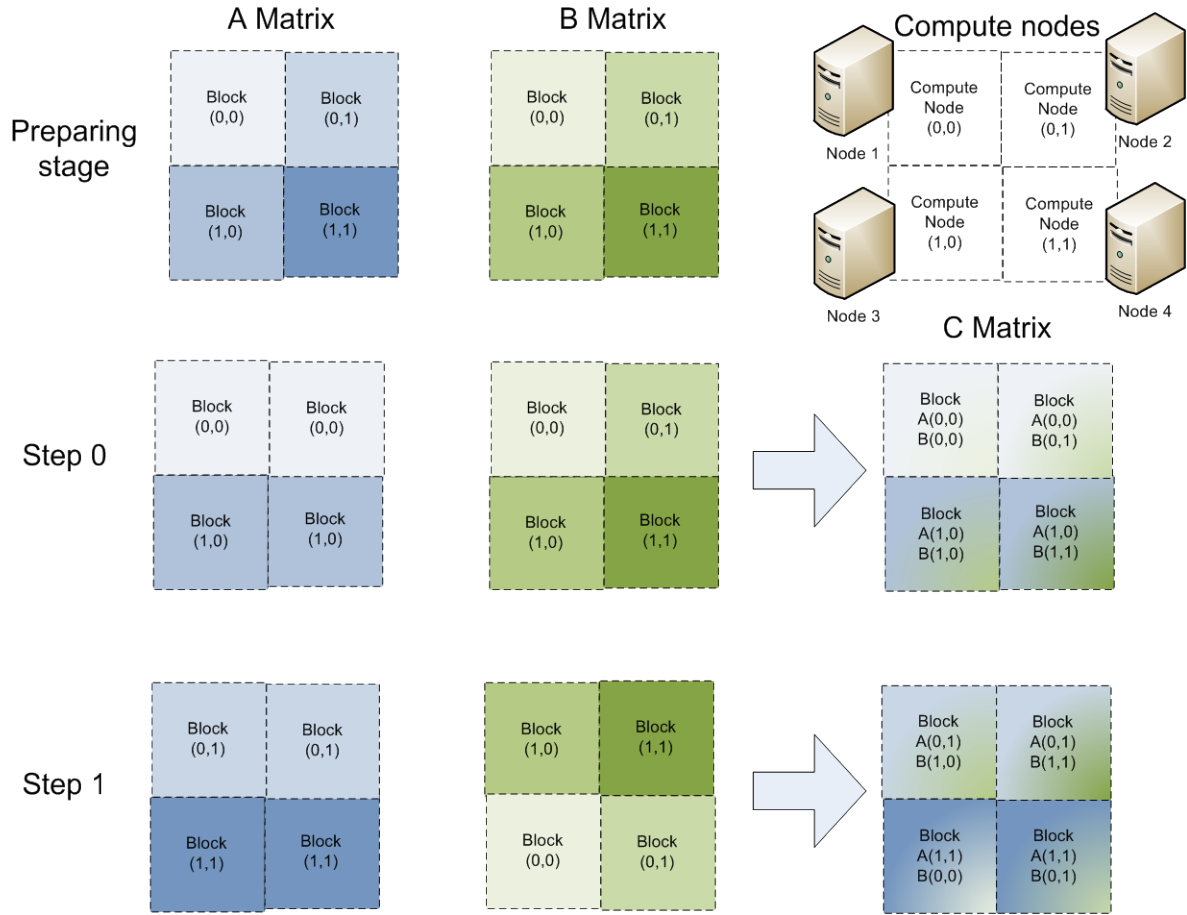


Figure 15: Different stages of an order 2 2D block decomposition illustrated

Figure 15 is an example of an A matrix and B matrix 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 A matrix 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 B matrix will be scattered onto each compute node, and perform an upward circular shift in each column. Then, the algorithm compute the $C_{ij} = AB$ on each compute node. In the Step 1, the A matrix 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 B matrix 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 $C_{ij} += AB$.

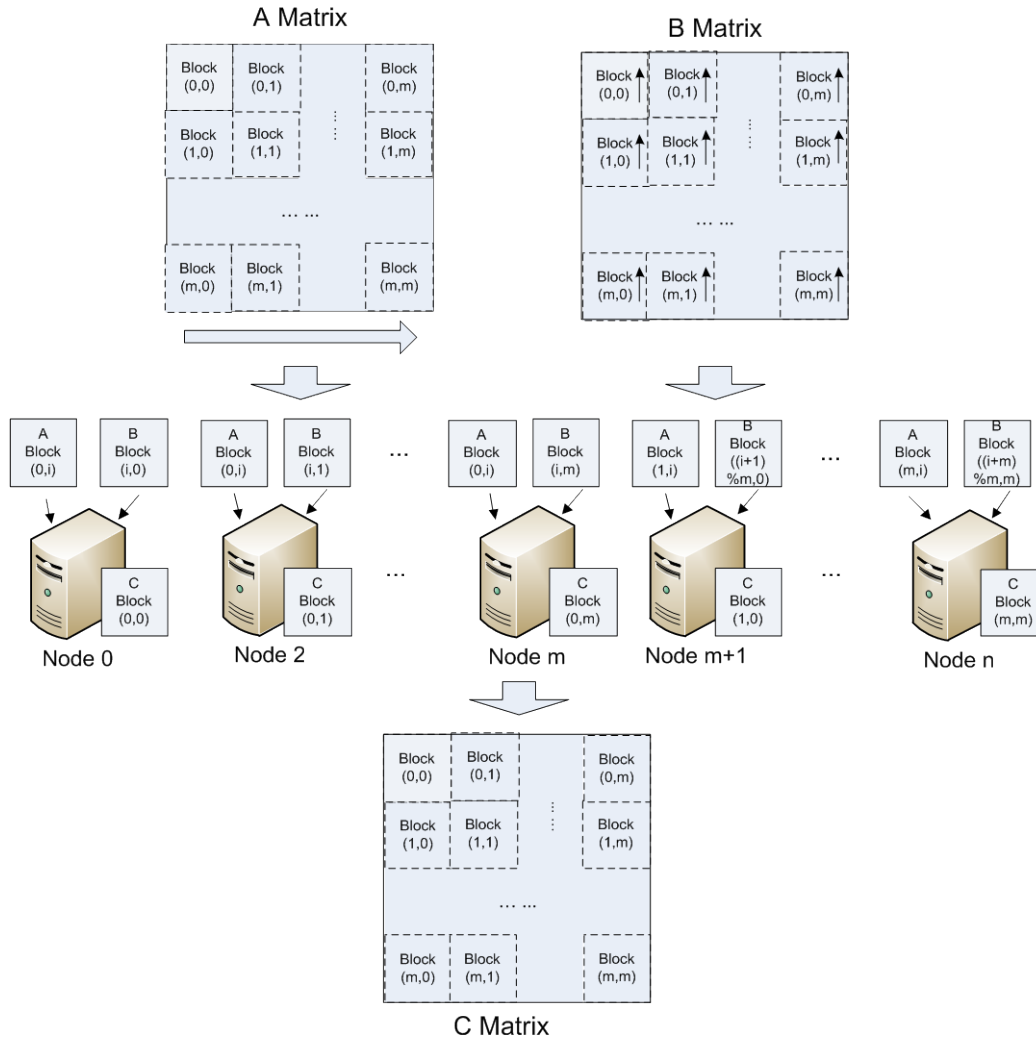


Figure 16: Structure of 2D decomposition algorithm on iteration i with $n+1$ nodes

Figure 16 is the structure of an $M+1$ order 2D block decomposition algorithm for $N+1$ nodes. So in each step, the blocks of A matrix and B matrix will be scattered onto each compute node; therefore, each compute node will have one block of A matrix and one block of B matrix, where the memory usage will be $2/(N + \text{sqrt}(N))$ compare to the row-split algorithm.

The decomposed block can also be saved as either `DistributedQuery` or `DistributedData`. We have also implemented it in both ways. We can directly read the matrix into blocks then distribute them to the compute node.

```
matrixBlock[][] aBlocks2D = buildBlocksFromFile(aMatrixPath, nProcesses);
matrixBlock[][] bBlocks2D = buildBlocksFromFile(bMatrixPath, nProcesses);
aMatrixcMatrixInput[] inputAC = buildBlocksInputOfAMatrixCMatrix(rowNum, colNum, 0,
nProcesses);
DistributedQuery<aMatrixcMatrixInput> inputACquery = inputAC.AsDistributed();
matrixBlock[] bBlocks1D = buildBlocksInputOfBmatrix(bBlocks2D, 0, nProcesses);
bMatrixInput[] inputB = buildbMatrixInput(bBlocks1D, 0, nProcesses);
```

```
DistributedQuery<bMatrixInput> inputBQuery =  
inputB.AsDistributed().WithConfiguration(config);
```

When we do the computation, the calculation can be done by directly using the DistributedQuery and the join operation.

```
matrixBlock[] aBlocks1D = buildBlocksInputOfAMatrix(aBlocks2D, iterations, nProcesses);  
    inputACQuery = inputACQuery.Join(aBlocks1D.AsDistributed(),  
acBlocks => acBlocks.ci, aBlocks => aBlocks.rowIndexOfBlocksMatrix, (acblocks, ablocks)  
=> acblocks.updateAMatrixBlock(ablocks));  
inputACQuery = inputACQuery.Join(inputBQuery, x => x.key, y => y.key, (x, y) =>  
x.multiplyBlock(y.bMatrix));  
inputBQuery = inputBQuery.Select(x => x.updateIndex(nProcesses));
```

By using the DSC, we can avoid the serialization cost by distributing the blocks' file names to each node during its steps.

```
string[] aPartitionsFile = splitInputFile(aMatrixPath, nProcesses);  
string[] bPartitionsFile = splitInputFile(bMatrixPath, nProcesses);  
IEnumerable<aMatrixcMatrixInput> inputAC = buildBlocksInputOfAMatrixCMatrix(rowNum,  
colNum, 0, nProcesses);  
DistributedQuery<aMatrixcMatrixInput> inputACQuery =  
inputAC.AsDistributed().WithConfiguration(config).HashPartition(x => x, nProcesses *  
nProcesses);  
DistributedQuery<bMatrixInput> inputBQuery =  
bPartitionsFile.AsDistributed().WithConfiguration(config).Select(x =>  
buildbMatrixInput2(x, 0, nProcesses)).SelectMany(x => x);
```

And inside each step, the blocks will be read from the files onto each node; we will use ApplyPerPartition and join operation to do the calculation and update the index of the blocks to put on each node in the next step.

```
inputACQuery = inputACQuery.ApplyPerPartition(sublist => sublist  
    .Select(acblock =>  
acblock.updateAMatrixBlockFromFile(aPartitionsFile[acblock.ci], iterations,  
nProcesses))).Execute();  
inputACQuery = inputACQuery.Join(inputBQuery, x => x.key, y => y.key, (x, y) =>  
x.task_multiplyBlock(y.bMatrix));  
inputBQuery = inputBQuery.Select(x => x.updateIndex(nProcesses));
```

Core Level Parallelism

The Dryad is coarse task granularity scheduler which deploys one Dryad vertex on each compute node. To leverage the computation power of multi core system, the programmers need to manually port multi core programming technologies, such as PLINQ, TPL, and thread pool into Dryad tasks.

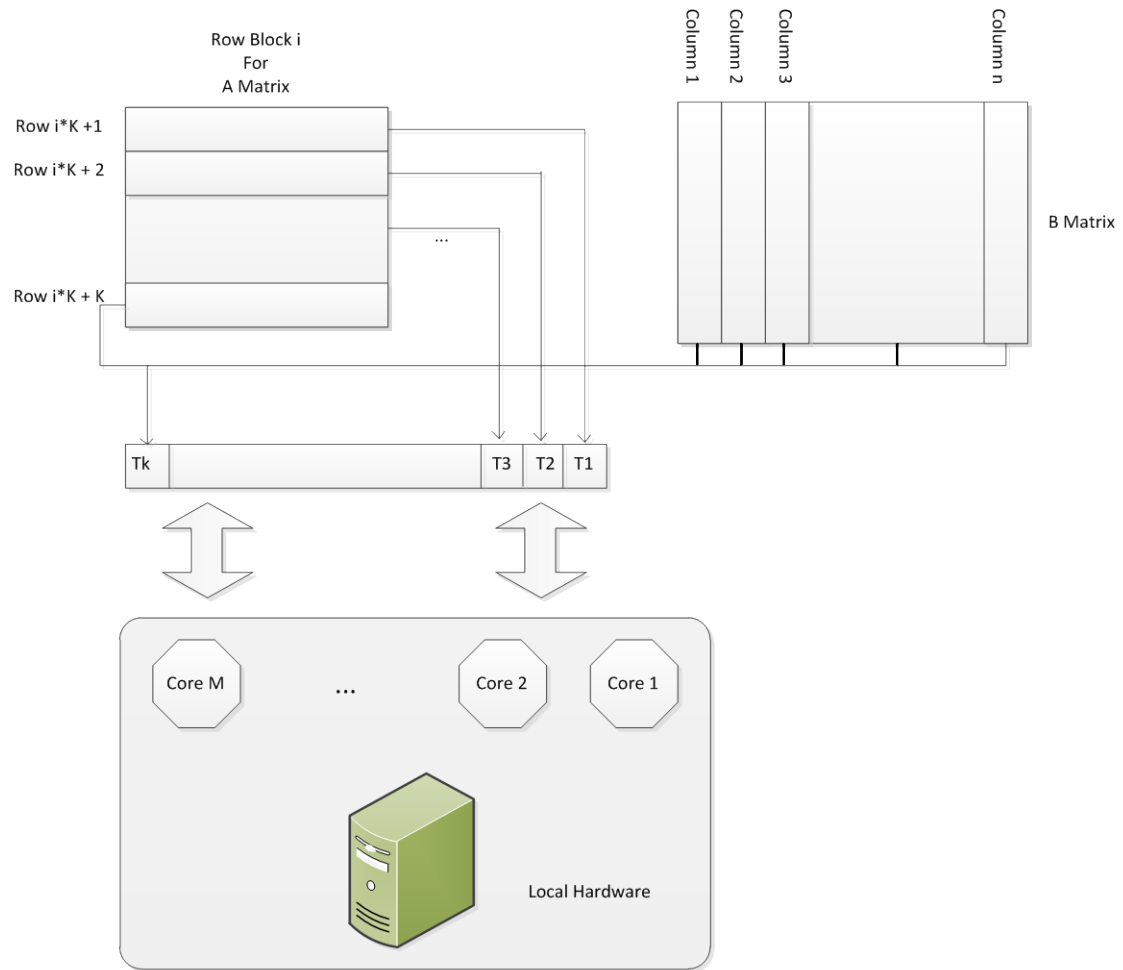


Figure 17: Coarse parallel task granularity

Figure 17 shows the coarse parallel task granularity each calculates one row of result of C Matrix. The pseudo code of coarse parallel task granularity implementation is as follow:

```

Read B matrix
Parallel foreach row in rowPartition
    foreach element in C matrix
        Use row in a row block and column in B to get element c

```

We implemented both fine and coarse parallel granularity task with following multi core technologies to increase the CPU utilization on each compute node: Parallel.For, Thread Pool, task pool and PLINQ, here the code is for the Row Split Algorithm, and it is similar to apply these technologies to other matrix multiplication algorithms:

Multi-Core Technologies

Inside the function rowXcolumns, the sequential code without using any multicore technology should be like this:

```

double[,] columns = readFromBMatrix(_localBMatrixPath);
IEnumerator<rowWrapper> localRows = rows.GetEnumerator();
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 is the most light weighted function and easy to use.

```

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

```

2) The tasks is like a thread pool but easier to use, and according to the author, tasks were implemented with the worker queue steal function, so it should be faster.

```

while(localBlocks.MoveNext()){
    double[] rows = localBlocks.Current.block;
    int size = localBlocks.Current.rowNum;
    blockWrapper rows_result = newblockWrapper();
    rows_result.rowNum = size;
    rows_result.colNum = colNum;
    rows_result.block = newdouble[size * colNum];
    Task[] tasks = newTask[size];
    for (int n = 0; n < size; n++)
    {
        int k = n;
        tasks[n] = Task.Factory.StartNew(() =>
        {
            for (int i = 0; i < colNum; i++)
            {
                double tmp = 0;
                for (int j = 0; j < rowNum; j++)
                    tmp += rows[k * rowNum + j] * columns[i][j];
            }
        });
    }
    yieldreturn rows_result;
}

```

```

        rows_result.block[k * colNum + i] = tmp;
    }
});
}
Task.WaitAll(tasks);
yieldreturn rows_result;
}

```

3) The thread pool is the most popular way for multicore level programming, so we implemented it as well.

```

while (localBlocks.MoveNext())
{
    double[] rows = localBlocks.Current.block;
    int size = localBlocks.Current.rowNum;
    blockWrapper rows_result = newblockWrapper();
    rows_result.rowNum = size;
    rows_result.colNum = colNum;
    rows_result.block = newdouble[size * colNum];
    int iters = size;
    ManualResetEvent signal = newManualResetEvent(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 += 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;
}

```

4) DryadLINQ provider the in-built AsParallel() operator to run tasks in parallel as follows:

```

while (localBlocks.MoveNext())
{
    double[] serializedRows = localBlocks.Current.block;
    int size = localBlocks.Current.rowNum;
    blockWrapper rows_result = newblockWrapper();
    //rows_result.start = DateTime.Now.Date;
    rows_result.rowNum = size;
    rows_result.colNum = colNum;
    rows_result.block = newdouble[size * colNum];
    double[][] rows = newdouble[size][];
    for (int i = 0; i < size; i++)
    {
        rows[i] = newdouble[rowNum];
        for (int j = 0; j < rowNum; j++)
        {
            rows[i][j] = serializedRows[i * rowNum + j];
        }
    }
    IEnumerable<double[]> result = rows.AsEnumerable().AsParallel().AsOrdered()

```

```

        .Select(x => oneRow(x, columns));
IEnumerator<double[]> result_iter = result.GetEnumerator();
int index = 0;
while (result_iter.MoveNext())
{
    for (int i = 0; i < colNum; i++)
        rows_result.block[index * colNum + i] = result_iter.Current[i];
    index++;
}
yieldreturn rows_result;
}

```

Performance Analysis in Hybrid Parallel Model

Performance in Multi Core Parallelism

We executed the Matrix Multiplication jobs on TEMPEST with various multi core technologies in .NET. As the platform is based on .NET 4, we used tasks and parallel for inside Task Parallel Library (TPL), thread pool and PLINQ to do the parallelism.

We have tested it on the one of tempest compute node which has 24 cores, the size of A matrix and B matrix are from 2400 * 2400 to 12000 * 12000. In this way, the computation for each thread will increase as the data size increase, and the result is shown below:

We calculate the speed up of Matrix Multiplication with various Matrix size on TEMPEST nodes 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.

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

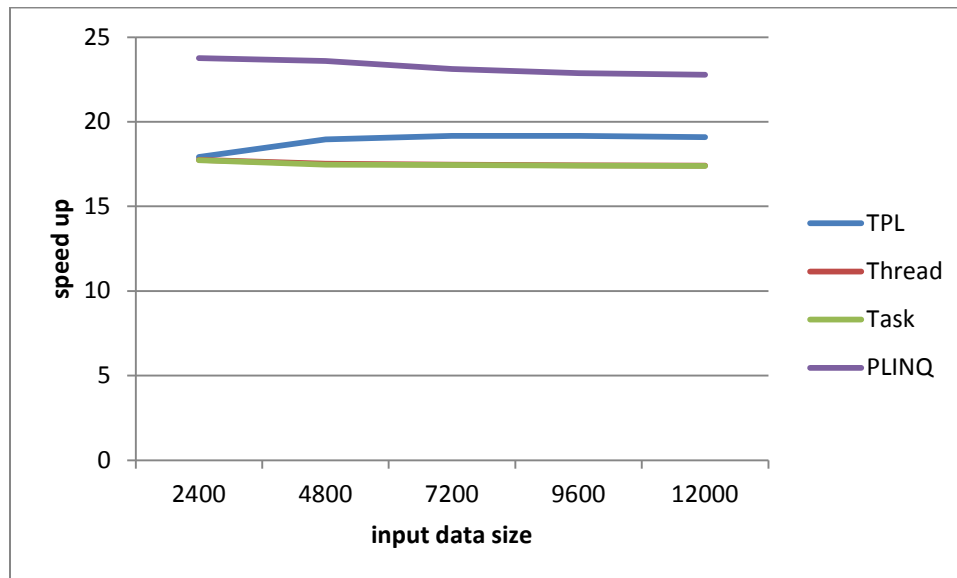


Figure 18: the speed up chart for different method of multi-core parallelism on 1 node

As we can see from Figure 18, the efficiency of parallelism remains around 17 to 18 for TPL, Task and Thread. However, while the data size becomes larger, the TPL do performs better than the Task and Thread where the latter two has nearly identical performance. And the 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, thread and Task seems low should due to the memory cache issue as the data is large. And obviously, PLINQ has some optimization for large data size to make it very fast on multicore system.

Performance in Multi Node Parallelism

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

We evaluate the 3 different Matrix Multiplication algorithms without using any multicore technology on 16 nodes from TEMPEST. The 2D blocks decomposition algorithm and row split algorithm use DSC or NO-DSC approaches to stage in A matrix and B matrix, and the data size of Matrix is from 2400 x 2400 to 14400 x 14400.

We calculated the speed-up approximant of Matrix Multiplication jobs with 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.

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

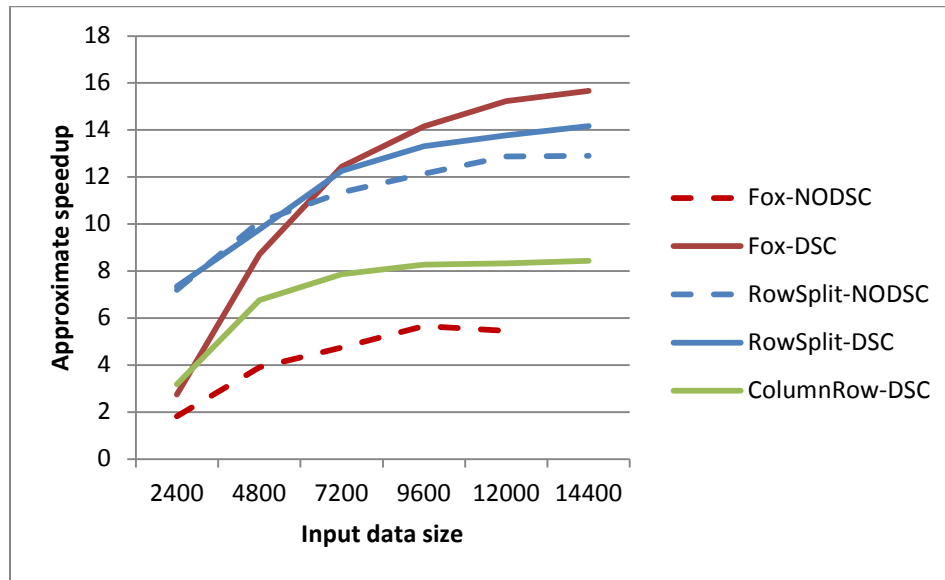


Figure 19: the speed up for different Matrix Multiplication models using one core per node

As shown in Figure 19, the Fox algorithm's performance increases dramatically by using the DSC to avoid the serialization cost. 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. The parallel efficiency for Fox-DSC, Row Split-DSC, Row Split-NODSC is shown in the following Figure 20.

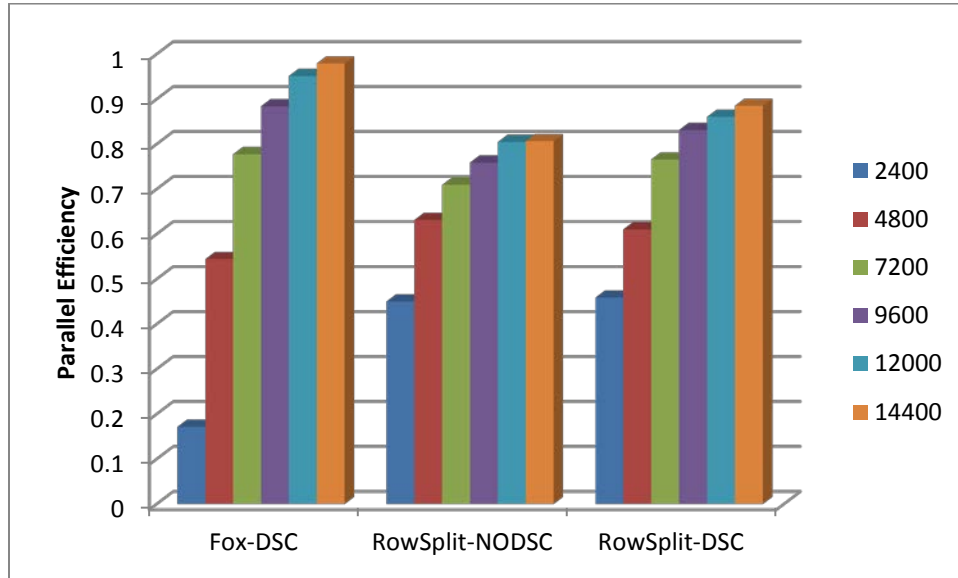


Figure 20: the parallel efficiency for different Matrix Multiplication models using one core per node

We study these 3 different algorithms since they have the best parallel efficiency. As we can see from the Figure 20, the highest parallel efficiency for each model is when data size is around 12000X * 12000. And the parallel efficiency for Fox-DSC is around 0.9 while the Row Split-DSC and NO-DSC's parallel efficiency is around 0.8.

Serialization Cost with DryadLINQ provider and DSC

In the multi-node level, we used RowSplit-DSC and RowSplit-NODSC approaches to read the A matrix. The test is done before we do the calculation. Basically we have recorded the time with DSC and the time without DSC, which showed the time it would be consumed by loading the data in to Distributed Query. The performance result is shown below:

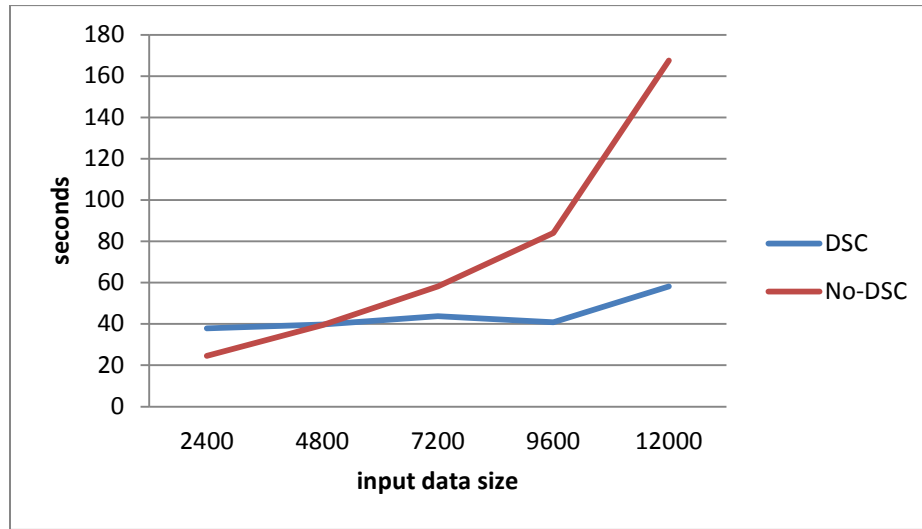


Figure 21: the time consumed for different data loading method

As shown in Figure 21, by using DSC the data loading time would be greatly reduced compare to the NO-DSC method. Both of the methods have partitioned matrix A into 31 blocks, to fit our 32-node cluster, as one node is used for scheduling. And the DSC's loading time remains the stable as data size increases. The performance gap is caused by serialization cost between the two implementations. In NO-DSC implementation, the input data (Matrix A) is serialized by DryadLINQ provider before sending out to compute node. In DSC implementation, we split Matrix A into several txt files and store them in Windows Network File System. The DSC will manage the data locality information of input files which will be used by Dryad vertex when it staging input file in via network shares.

Performance in Hybrid model with different multi-core technologies

As we declaimed previously, porting multi-core technologies into Dryad tasks can increase the overall performance greatly. Here we verified our conclusion by applying single-core and various multi-core technologies (TPL, Thread, Task, PLINQ, Sequential) into row split Matrix Multiplication algorithm. We performed the experiments with various matrix sizes (from 2400x2400x2400 to 12000x12000x12000) on a 32 compute nodes HPC cluster TEMPEST (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, task, PLINQ to perform the Matrix Multiplication calculation in parallel. As shown in Figure 21, 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 way we program multi-core application. We may make further study on this issue in future.

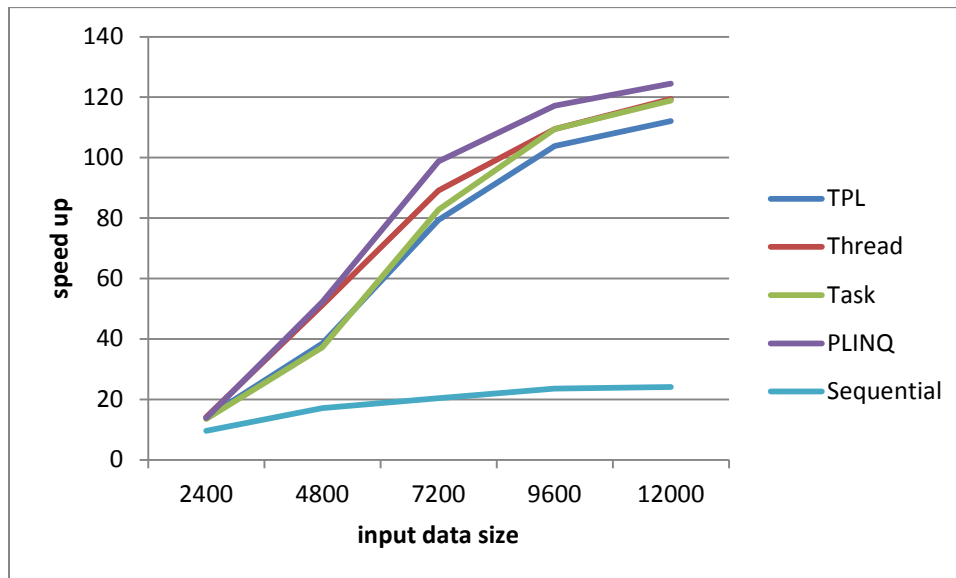


Figure 22: the time consumed for Row Split Algorithm using different core methods

Performance in Hybrid model with different algorithms

We chose 4 models from the multi node parallelism section which performs the best to run the different multi core technology integrated with these models on large scale data. The data size is 12000x12000x12000. Here is the performance chart for these models. The ideal speed up should be 16 nodes * 24 cores/node = 364. However, as we can see from Figure 23, the PLINQ always does best compare to the other TPL technology and using thread. In the 2D block decomposition algorithm, by using the PLINQ technology, the speed up can achieve 60 while the

other technologies' speed up is around 40. In the row split algorithm, the speed up for PLINQ is 127 for using DSC and 78 for the NO-DSC, which is still larger than the speed up of other 3 technology. As for Row Column Split Algorithm, the result shows the same as PLINQ has a speed up of 70 while the other technology has the speed up around 50.

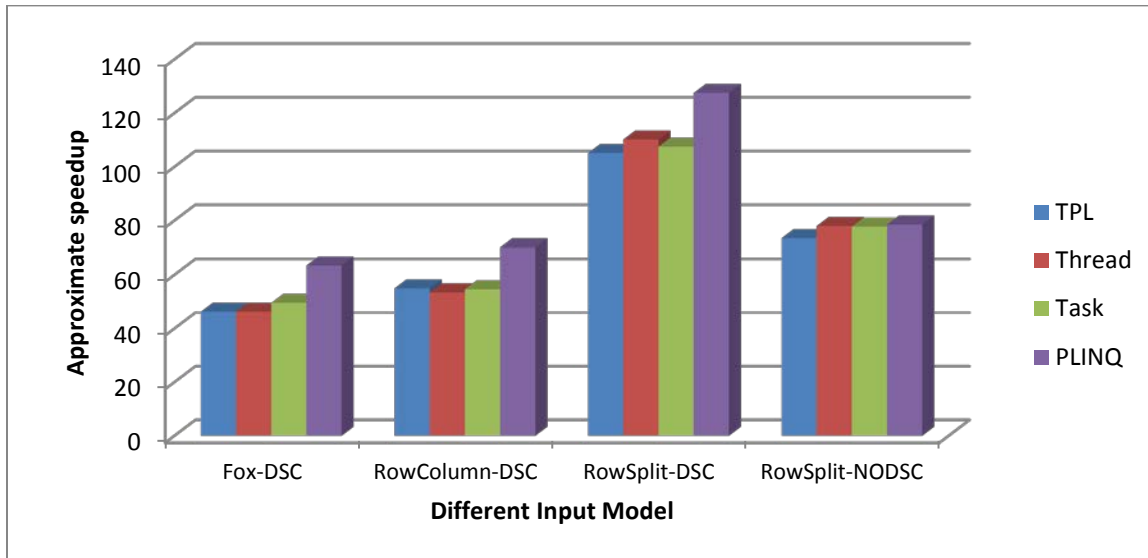


Figure 23: The speed up chart for different multicore technology apply on Matrix Multiplication

The speed up of row split algorithm increase greatly while importing the multicore technology on row split matrix multiplication-DSC algorithm. The reason for the speed up is around 127 instead of the 16 nodes * 24 cores/node = 384 is the communication cost during each computation is too high.

When we use the NODSC way to read the data, basically the A matrix is read into 16 blocks, and serialized to send to each compute node, there is also the B matrix being directly copy to the compute node. Let's assume N is the input data size. The communication cost for a size 1 matrix multiplication is T_1 . So the communication cost for a size N matrix multiplication is: compute node number (16) * T_1 * N^2 .

Assume the computation cost for a size 1 matrix multiplication is T_0 , so the computation cost for a size N matrix multiplication is $T_0 * N^3 / 16$. So the computation / communication for matrix multiplication should be $O(N)$. However, it's also depends on T_0/T_1 . If T_0 becomes larger and larger while doing the communication, the computation/communication is hardly increased. And as we have mentioned before in the [Performance in Multi Node Parallelism](#) section, the parallel efficiency for one core per node is around %80 to %90 for every model. So when we using the multicore technology, the parallel efficiency would decrease dramatically.

The following is chart of the CPU utilization and network activity on one compute node while 3 jobs are submitted to do a PLINQ 12000 x 12000 x 12000 matrix multiplication using Row Split Algorithm, Row Column Split Algorithm and Fox Algorithm. The lower graph is the CPU usage rate and the higher graph is the network activity.



Figure 24: The CPU usage and network activity on one compute node for multiple algorithms

From this chart we can see that only the Row Split Algorithm can reach a CPU utilization rate at %100, but only for 2 minutes. The whole job for Row Split Algorithm is finished in 6 minutes, and the network activity for it is the lowest. As for Row Column Split Algorithm, the computation can reach 30 to 40 percent for each iteration, and the network activity is higher than the Row Split Algorithm since within every iteration, the scheduler has to send out the new row block to every node and at reduce stage, to collect all the result matrix blocks.

The Fox algorithm performs best in the sequential version, but now it performs worst with the multicore technology. The CPU utilization for it can reach only at 30 percent, and the network activity is extremely high, which is 3 times of the Row Column Split Algorithm and 4 times of the Row Split Algorithm.

The Fox algorithm takes 4 steps as we have an order 4 decomposition for the input matrixes. In each step, the CPU utilization cannot reach 100% because of the computation granularity is too small for 1/16 block multiply 1/16 block. The computation only takes a few seconds and most of the time the Dryad spends was on how to allocate the blocks and schedule the job during each step.

So the premise should be that if we increase the size of the input matrix, the CPU utilization of Fox algorithm should be higher. Here is an example of test on 24000 x 24000 x 24000 compared with 12000 x 12000 x 12000 of Fox algorithm using PLINQ as multicore technology.

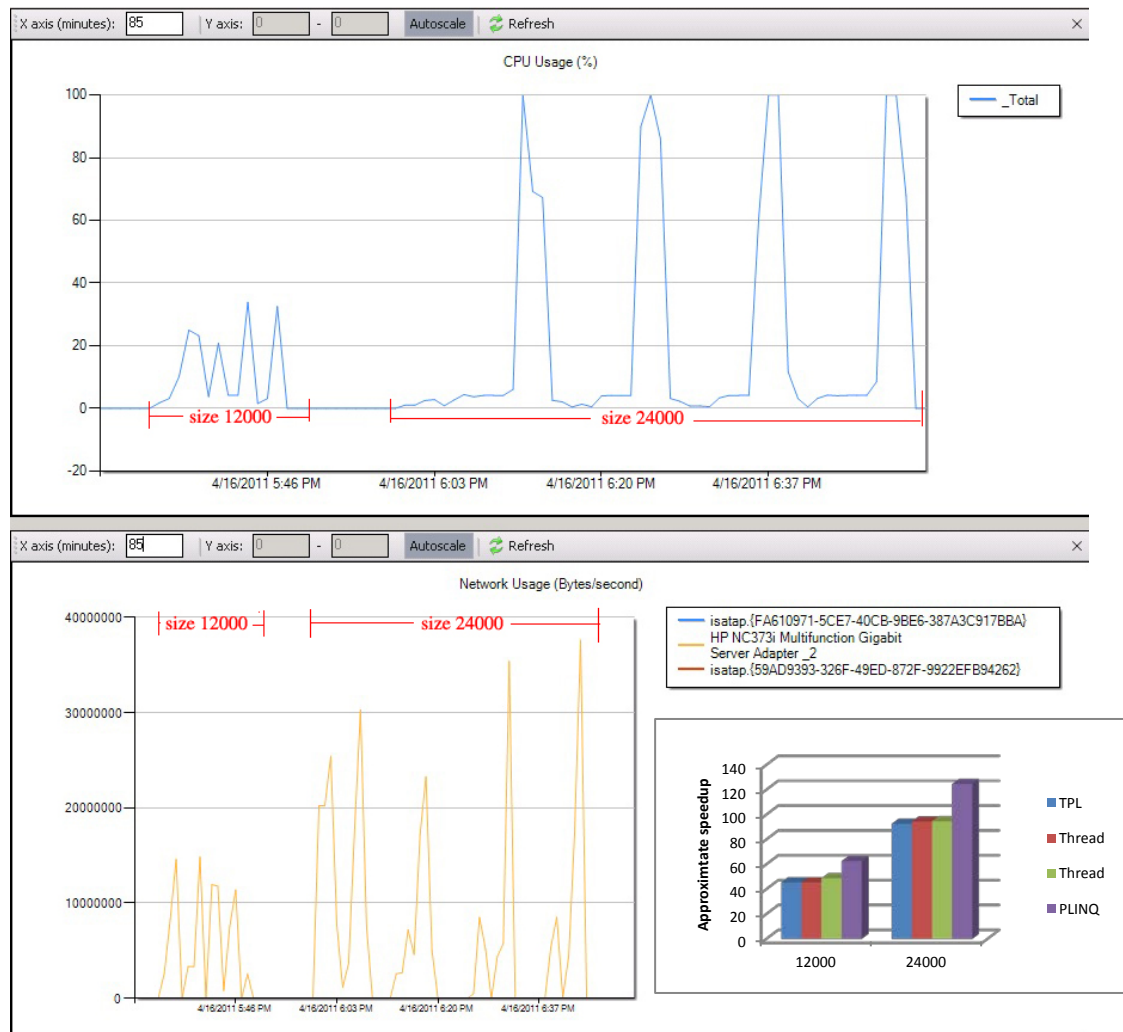


Figure 25: The CPU usage and network activity for Fox Algorithm-DSC by different sizes

As we can see from the Figure 25, the CPU utilization can reach %100 while doing the computation, this increases the speed up of Fox algorithm integrate with PLINQ from 60 to 130. However, the overall CPU utilization is still low because of the communication time. The communication time increases as the input data size increases. And we can also see that clearly from the size 24000 case that the network activity only happens when the CPU utilization is low. It clearly states that during each step, the communication cost is so high that the overall parallel efficiency is low.

Conclusion

Except PLINQ, the Dryad user has to manually port the multi-core technologies, such as TPL, threads, to Dryad tasks so as to increase the CPU utilization of HPC cluster. However other runtimes like, Hadoop, MPI, Twister can make use of multi-core computing power by deploying multi task daemons on each compute node. This process is transparent to programmers so that they can more focus on the logic implementation.

Distributed Grouped Aggregation

Introduction

We have studied the distributed grouped aggregation in DryadLINQ CTP with PageRank with real data. Specifically, we investigated the programming interface and evaluate performance of three distributed grouped aggregation approaches in DryadLINQ which include: Hash Partition, Hierarchical Aggregation and Aggregation Tree. Further, we studied the features of input data that affect the performance of distributed grouped aggregation implementations. At the end we compare the performance of distributed grouped aggregation between DryadLINQ and 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 [16] which contains 50 million web pages.

Distributed Grouped Aggregation Approaches

We study the three distributed grouped aggregation approaches and Fig. 26 is their work flow.

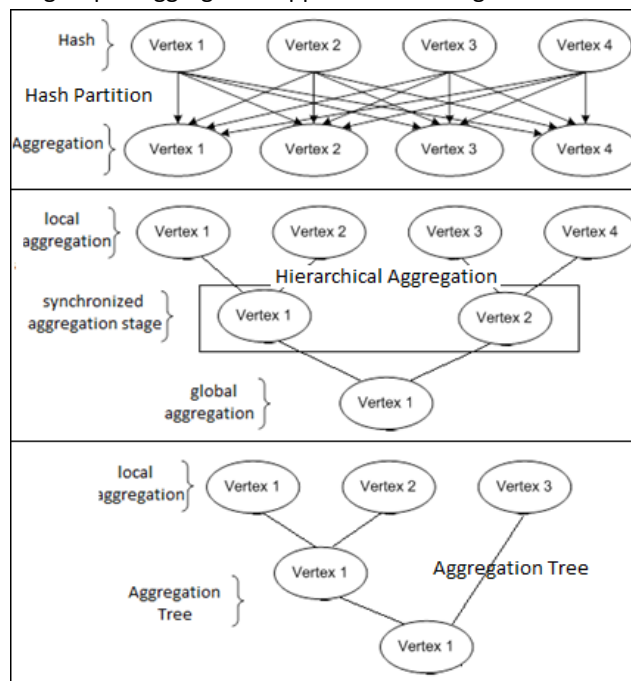


Fig 26 Three distributed grouped aggregation approaches

The Hash Partition uses hash partition operator to redistribute the records to compute nodes so that identical records store on the same node. Then it merges the records of each group on each node. The hash partition 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 aggregates the local records of each node, and then hash partition 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 ways 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.

DryadLINQ can automatically translate the distributed grouped aggregation query and its combine functions satisfy the associative and commutative rules into optimized aggregation tree. During processing, 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 efficiency of grouped aggregation applications.

Hash Partition

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

PageRank implemented with GroupBy() and Join()

```
DistributedQuery<Page> pages = Directory.GetFiles(inputDir, inputFilePattern,
SearchOption.AllDirectories).AsDistributed().SelectMany(fileName =>
buildPagesFromAMFile(fileName));
.....

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.Execute();//you can comment out Execute() to speed up program
}
```

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 hash partition 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 updated with power method [20].

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 => newVertex(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 hash partition PageRank is not efficiency when the number of output tuples is small. Thus we also implemented PageRank with hierarchical aggregation approach which has tree fixed 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 rank 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 code and the understanding of web graph from developer side.

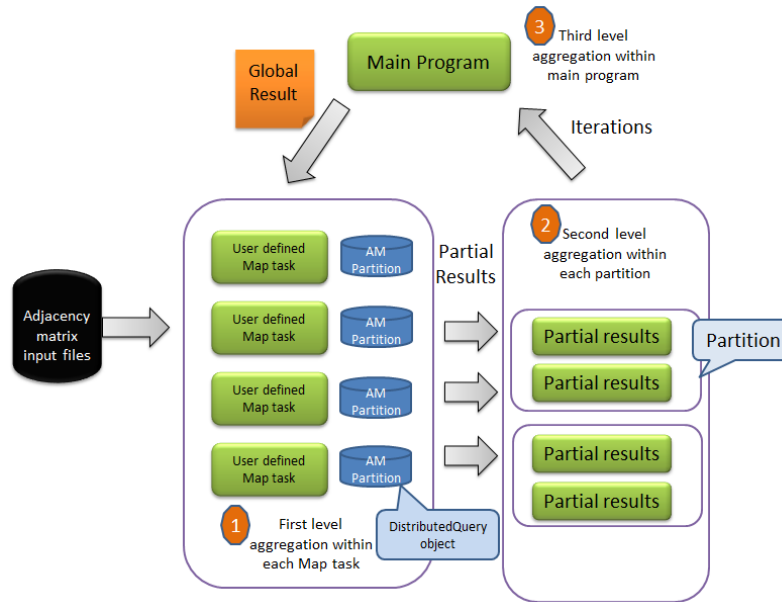


Figure 27: Hierarchical aggregation in DryadLINQ PageRank

Hierarchical Aggregation with User Defined Aggregation function

```
DistributedQuery<amPartition> WebgraphPartitions = Directory.GetFiles(_inputDir, _inputFilePattern,
SearchOption.AllDirectories).AsDistributed().WithConfiguration(config).Select(fileName =>
buildWebGraphPartition2(fileName));
for (int i = 0; i < _iteration; i++){
    DistributedQuery<double[]> partialRVTs = null;
    partialRVTs = WebgraphPartitions.ApplyPerPartition(subPartitions =>
        calculateMultipleWebgraphPartitionsWithPLINQ(subPartitions, rankValueTable, _numUrIs));
    rankValueTable = mergeResults(partialRVTs);
}
publicstaticIEnumerable<double[]> calculateMultipleWebgraphPartitionsWithPLINQ(IEnumerable<amPartition>
webGraphPartitions, double[] rvt, int numUrIs) {
return webGraphPartitions.AsParallel().Select(amp =>
calculateSingleWebgraphPartition(amp,rvt,numUrIs)); }
```

Aggregation Tree

The hierarchical aggregation approach may not perform well for computation environment which is inhomogeneous in network bandwidth, CPU, memory capability, because it has several synchronization stages. 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 in 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. We implemented PageRank with GroupAndAggregate() operator as follows:

PageRank implemented with GroupAndAggregate

```
for (int i = 0; i < _iteration; 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 query perform the sub-GroupBy for each partition on compute node. The second ApplyPerPartition aggregate the aggregated partial results produced by first ApplyPerPartition.

PageRank implemented with two ApplyPerPartition steps

```
for (int i = 0; i < _iteration; 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 =>newTuple<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();
    SaveResults(ranks);
}
```

Performance Analysis:

Performance in Different Aggregation Strategies

We evaluate performance of the three approaches by running PageRank jobs with various sizes of input data on 17 compute nodes on TEMPEST. Fig. 29 shows that the aggregation tree and hierarchical aggregation approaches outperform hash partition approach. Fig. 28 is the CPU utilization and network utilization statistic data obtained from HPC cluster manager for the three aggregation approaches. It shows that the partial aggregation requires less network traffic than hash partition in the cost of CPU overhead. The hierarchical aggregation approach outperforms aggregation tree because it has the coarser granularity processing unit. Besides, our experiment environment of TEMPEST cluster has homogeneous network and CPU capability.

We split the entire ClueWeb graph into 1280 partitions, each of which is saved as Adjacency Matrix (AM) file. The characteristics of the input data are described as below:

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

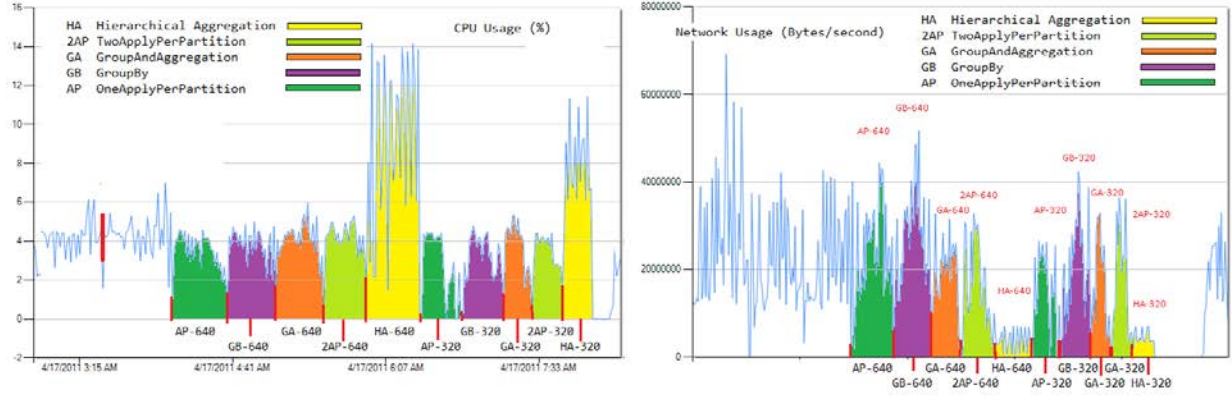


Fig. 28 CPU and Network Utilization for Different Aggregation Strategies

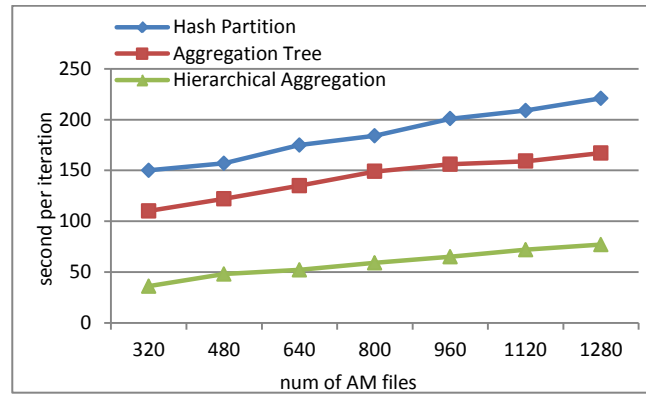


Fig. 29 Time in sec to compute PageRank per iteration with three aggregation approaches with clue-web09 data set on 17 nodes of TEMPEST

In summary, the hierarchical aggregation and aggregation tree approaches have different trade-offs on memory and CPU overhead vs. 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. We design a mathematics model to describe how the ratio between input and output tuples affects the performance of aggregation approaches. First, we define the data reduction proportion (DRP) to describe the ratio as follows:

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

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

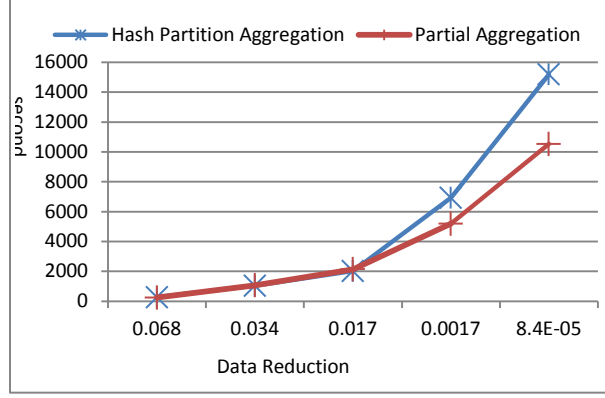


Fig. 30 Time for two aggregation approaches with different DRP values.

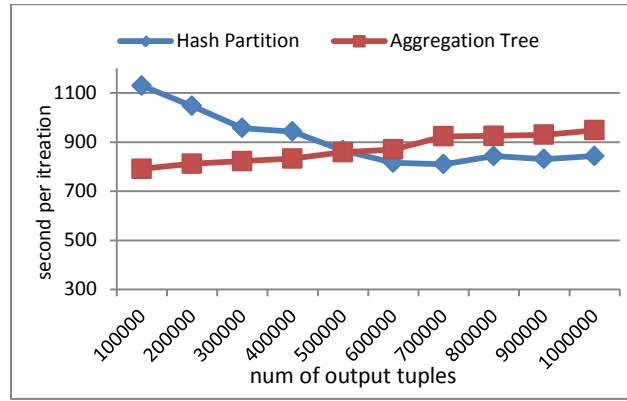


Fig. 31 Time per iteration for two aggregation approaches with different number of output tuples (from 100000 to 1000000) when number of input tuples is 4.3 billion

Further, we define a mathematic model to describe how DRP will affect the efficiency of different aggregation approaches. Assume the average number of 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 on the N nodes. In hash partition approach, the M tuples with same key are hashed into same group on one node, which require M aggregation operations. In partial aggregation approaches, the number of local aggregation operations is M/N on each node, which produces N partial aggregated results and need N more aggregation operations. Thus the total number of aggregation operations for the M tuples is $(M/N)*N+N$. Then the average number of aggregation operations of each record of the two approaches is as follows:

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

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 several thousand common 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 [4].

To quantitatively analysis of how DRP affects the aggregation performance, we compare two aggregation approaches with a set of web graphs with 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 per iteration of PageRank jobs of web graphs with different number of output tuples when that of input tuples fixed. Fig.30 and 31 show that different grouped aggregation approaches fit well with different DRP range of input data.

Comparison with other implementations

We have implemented the PageRank with DryadLINQ, Twister, Hadoop, Haloop, and MPI on ClueWeb data set [17] with the Power method [22]. We perform experiments of the five implementations with same 1280 AM input files but with different hardware and software environment illustrated in Table. 7. To remove those inhomogeneous factors as much as possible, we calculate the parallel efficiency 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 across the cluster. $T(S)$ means the job turnaround time of sequential PageRank with only one core.

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

Figure 32 shows the parallel efficiency are noticeably lower than 1%, as PageRank is communication intensive application, and the computation do not take large proportion of overall PageRank job turnaround time. The bottleneck 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 across the cluster. 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 that 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. We found that Dryad is faster than Hadoop, but is still much slower than MPI, 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.

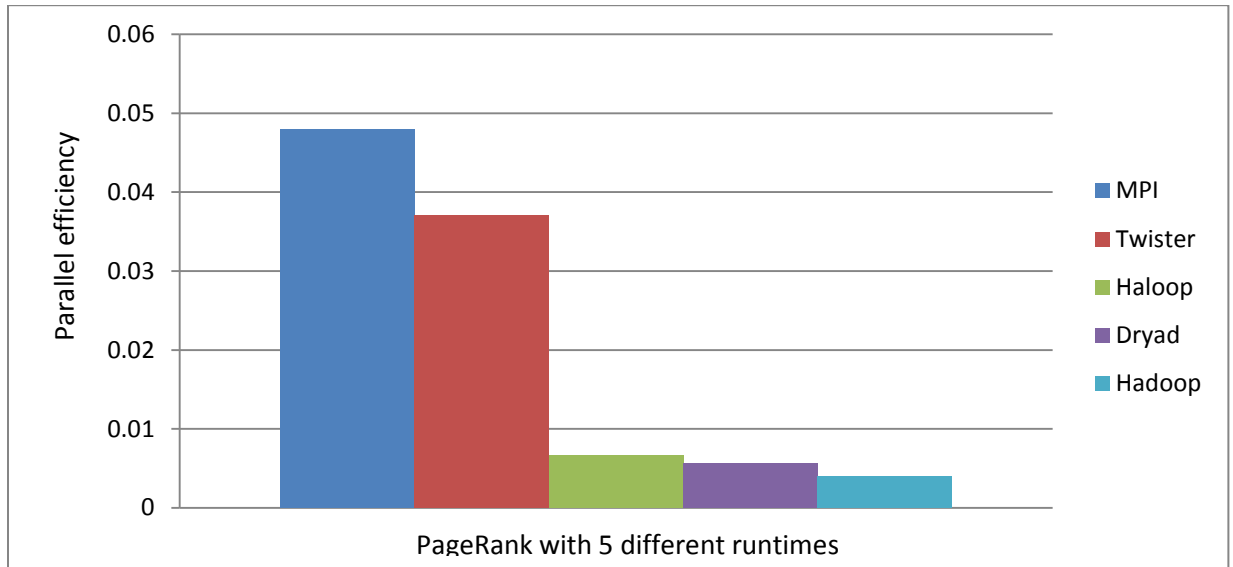


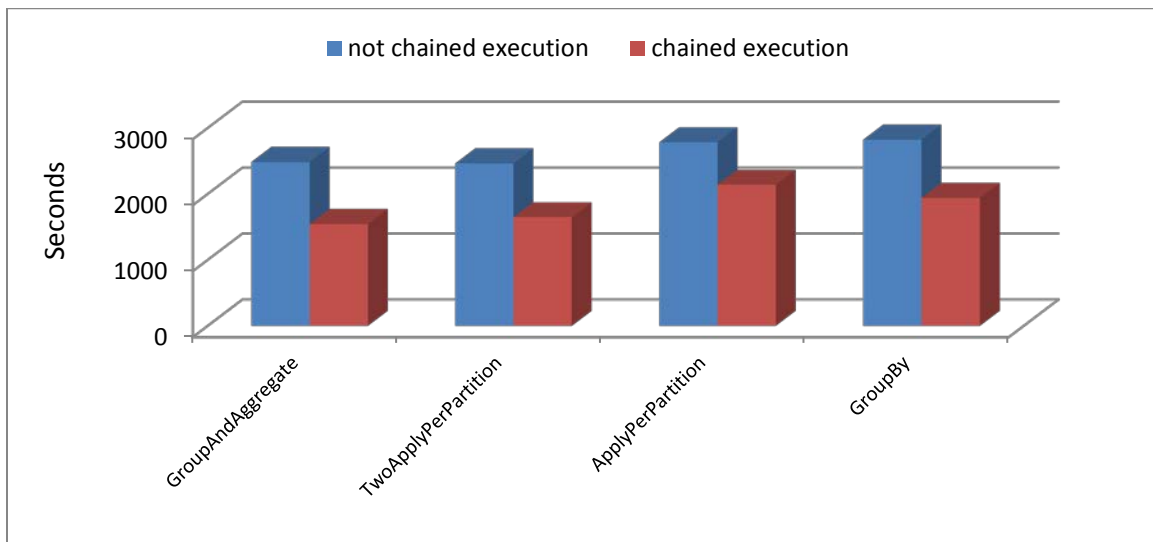
Figure 32: Parallel efficiency of five different PageRank implementations

Table 7: 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

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 perform 1280 AM PageRank input files on TEMPEST for 10 iterations. However, DryadLINQ does not efficiently support chaining the execution of multiple tasks within BSP style job due to the explicitly evaluation of sub query in each execution step. Thus DryadLINQ does not efficiently support our hierarchical aggregation strategy for PageRank. In last aggregation stage, the PageRank code have to merge all the partial rank values to calculate the global rank value table which is used as input in next iteration. Thus, the calculation of global rank value table will trigger the evaluation of DryadLINQ query in each iteration. Each evaluation will trigger reboot of Dryad vertex on each compute node, while this situation does not happen in MPI, Twister, and Haloop.

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

Generally, the parallel computation model of hierarchical aggregation PageRank belongs to the BSP, Bulk Synchronous Parallel [23], which consists of three stages: 1) Concurrent computation, 2) Communication, 3) Barrier synchronization. In Dryad, the evaluation in each barrier synchronization stage will trigger the resubmit of Dryad

jobs to HPC cluster. But in MPI, Twister, there is no need to restart task daemons in each barrier synchronization stage. This is one of the main reasons lead to the performance gap between Dryad, MPI, and Twister.

Conclusion:

At last, we studied the different aggregation approaches in DryadLINQ CTP. And the experiment results showed that different approaches fit well with different range of data reduction proportion (DRP). We designed a simple mathematics model to describe the overhead of aggregation approaches. For a complete analysis of the performances of aggregation approaches, one has to take in consideration several factors: 1) The size of memory on each node. Partial pre-aggregation requires more memory than hash partition. 2) The bandwidth of network. Hash partition has larger network traffic 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 cost of distributed grouped aggregation.

DryadLINQ support the iterative tasks by chaining the LINQ queries in different iteration together. However, for BSP style applications that need explicitly evaluate LINQ query in each iteration, it has to resubmit Dryad job to HPC job manager for each iteration, which do harm to the overall performance. System such as MPI, Twister and Hadoop can performance iterative computation by chaining the iterative task without restarting the task daemons.

Programming Issues in DryadLINQ CTP

Class Path in Working Directory

We found this issue when running DryadLINQ CTP SW-G jobs o Dryad CTP. Dryad can automatically transfer necessary files of user program to remote working directory on each compute. To save disk space, Dryad [26] does not copy all DLL and other share files in sub task working directory, rather it stores only one copy of shares files in the job working directory, because the Dryad vertex can add the job working directory into the class path of task working directory. 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 will throw error says 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 allow DryadLINQ provider to optimize the execution plan of multiple DryadLINQ queries. However, we found a program issue when running chained DryadLINQ queries in iterative PageRank jobs. As show in following code, the DryadLINQ query within different iteration are chained together and not get evaluated until invoke the Execute() operator explicitly. The parameter "iterations" should have different value in different query. However, the DryadLINQ provider just evaluates "iterations" parameter only once and assigns this value to all the queries. DryadLINQ compiler should save the parameters in different queries with different values box during compiling.

```

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.multiplybBlock(y.bMatrix));
    inputBQuery = inputBQuery.Select(x => x.updateIndex(nProcesses));
} //for iterations

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 object. 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 automatically generated by DryadLINQ provider, and found it may not be able to unserialize two dimension array objects correctly.

```

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]);
    }
}

```

Education Session

Dryad/DryadLINQ has shown its practical applicability in wide range of applications in both industry and academia, which include: image processing in WorlWideTeleScope, data mining in Xbox, HEP in physical, and SW-G, CAP3, PhyloD bioinformatics applications. An increasing number of graduate students, especially master students, have shown their interest and would like to learn Dryad/DryadLINQ. Here in Indiana University, Prof. Qiu made use of Dryad/DryadLINQ to teach graduate students the knowledge about large scale coming in the B649: Cloud Computing for Data Intensive Science in 2010 fall semester and B534: Distributed System in 2011 spring semester.

In the B649 Cloud Computing for Data Intensive Science class, 8 master students selected topics related with Dryad/DryadLINQ as their term long projects. They completed three following Dryad projects at the end of 2010Fall 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

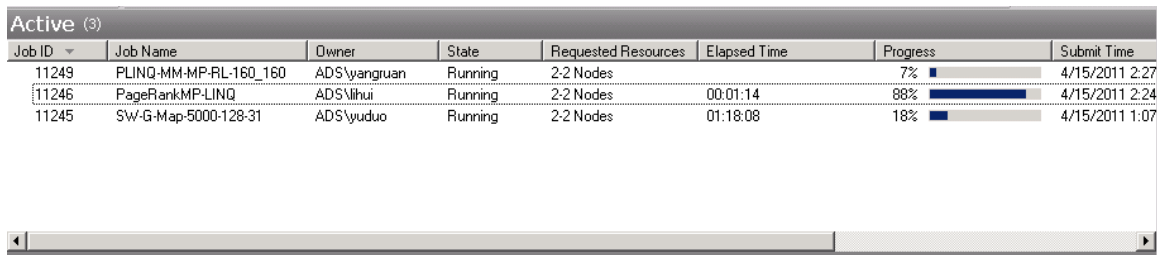
Project 2 and 3 were accepted as posters in the CloudCom2010 conference hosted in Indianapolis, IN. In the B534 “Distributed System” class in 2011 spring semester, 2 students took Dryad/DryadLINQ as their term long projects and contributed some small but solid results for this report.

Concurrent Dryad jobs

From the experience of the two classes using Dryad/DryadLINQ, we got a few observations on how student would utilize clusters to understand the theories and applications in large scale computing:

1. Ordinary class will contain 30-40 students, which can form 10 – 15 groups.
2. Student groups will not run extremely huge scale computation. Most of their jobs require only 1 or 2 nodes and will finish within 1 hour.
3. Students may not submit jobs until the due date is approaching. In another word, when a deadline is coming, there can be many jobs in the queue waiting to be executed, other than this time the cluster may be in an idle state.

Base on the above observation, we can see it’s critical to enable simultaneous job execution on a HPC cluster, especially in education respect. In the Dryad CTP, we manage to do this by make each job reside in a different node group like in Figure 32. In this way, a middle sized cluster with 30 compute nodes can hold up to 15 concurrent jobs. However, this feature is not mentioned in both the Programming Guide and User Guide.



Job ID	Job Name	Owner	State	Requested Resources	Elapsed Time	Progress	Submit Time
11249	PLINQ-MM-MP-RL-160_160	ADS\yangruan	Running	2-2 Nodes		7%	4/15/2011 2:27
11246	PageRankMP-LINQ	ADS\lihui	Running	2-2 Nodes	00:01:14	88%	4/15/2011 2:24
11245	SW-G-Map-5000-128-31	ADS\yuduo	Running	2-2 Nodes	01:18:08	18%	4/15/2011 1:07

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 as the job manager role. However the CPU usage of this particular node seldom exceeds 3%. This 8 node 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.

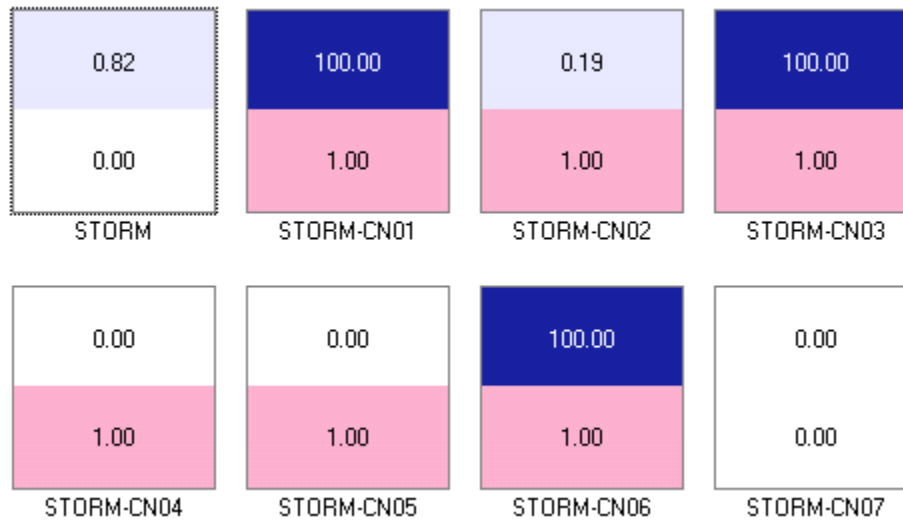


Figure 33: CPU Usage for Concurrent Job Execution

Acknowledgements

First, we want to thank John Naab and Ryan Hartman for their professional system administrator job for the STORM and TEMPEST HPC cluster, which is critical for our experiments. Second, we thank Thilina and Stephen for their generous to share the SW-G application and data in Dryad (2009.11) report, which is import for task scheduling analysis. At last, but not least, we want to thank Ratul and Pradnya, two master students taking Prof. Qiu's B534 class this semester, for their small but solid contribution to this report.

References:

- [1] Salsa Group, Applicability of DyradLINQ to Scientific Applications. 2010.
- [2] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, G. Fox, "Twister: A Runtime for Iterative MapReduce"
- [3] MPI (Message Passing Interface), <http://www-unix.mcs.anl.gov/mpi/>
- [4] Apache Hadoop, <http://hadoop.apache.org/core/>
- [5] Twister, <http://www.iterativemapreduce.org/>
- [6] Introduction to Dryad DSC and DryadLINQ. 2010
- [7] G. Fox, A. Hey, and Otto, S., Matrix Algorithms on the Hypercube I: Matrix Multiplication, Parallel Computing,4,17,1987
- [8] Haloop, <http://code.google.com/p/haloop/>
- [9] C.Moretti, H.Bui, K. Hollingsworth, B.Rich, P.Flynn, D. Thain, "All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids", IEEE Transactions on Parallel and Distributed System, 13, Mar. 2009.
- [10] M.A. Batzer, P.L. Deininger, 2002. "Alu Repeats And Human Genomic Diversity." Nature Reviews Genetics 3, no. 5: 370-379. 2002
- [11] Source Code. Smith Waterman Software. <http://jaligner.sourceforge.net>
- [12] T.F. Smith, M.S.Waterman,. Identification of common molecular subsequences. Journal of Molecular Biology 147:195-197, 1981.
- [13] O. Gotoh, An improved algorithm for matching biological sequences. Journal of Molecular Biology 162:705-708 1982.

- [14] Daan Leijen, Wolfram Schulte and Sebastian Burckhardt, “The Design of a Task parallel Library”, Microsoft Research, OOPSLA’09
- [15] G. Fox, A. Hey, and Otto, S., Matrix Algorithms on the Hypercube I: Matrix Multiplication, Parallel Computing, 4, 17, 1987
- [16] Johnsson, S. L., T. Harris, et al. 1989, Matrix Multiplication on the connection machine. Proc of the 1989 ACM/IEEE conference on Supercomputing. Reno, Nevada, United States, ACM.
- [17] Jaliya Ekanayake, “Architecture and Performance of Runtime Environments for Data intensive Scalable Computing”, Dec 2010
- [18] Y. Yu, P. Kumar, M. Isard, “Distributed Aggregation for Data Parallel Computing”, 2009.
- [19] Source Code. Smith Waterman Software. <http://jaligner.sourceforge.net>
- [20] ClueWeb Data: <http://boston.lti.cs.cmu.edu/Data/clueweb09/>
- [21] New DryadLINQ Programming Guide Dec. 2010.
- [22] PageRank wiki: <http://en.wikipedia.org/wiki/PageRank>
- [23] BSP, Bulk Synchronous Parallel http://en.wikipedia.org/wiki/Bulk_Synchronous_Parallel
- [24] J. Ekanayake, A. Soner, T. Gunarathne, and G. Fox, C. Poulain, “DryadLINQ for Scientific Analysis”. 2009.
- [25] Workshop on Data-Intensive Scientific Computing Using DryadLINQ, <http://research.microsoft.com/en-us/events/dryadling2010/>
- [26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks”, European Conference on Computer Systems, March 2007
- [27] J. Ekanayake, S. Pallickar, and G. Fox, “MapReduce for Data Intensive Scientific Analysis”, Fourth IEEE International Conference on eScience, 2008, pp. 277-284.
- [28] 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.
- [29] H. Yang, A. Dasdan, R. Hsiao, D. Stott, “Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters”, SIGMOD’07, 2007

Appendix

Appendix A

STORM

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

33-node homogeneous HPC R2 cluster

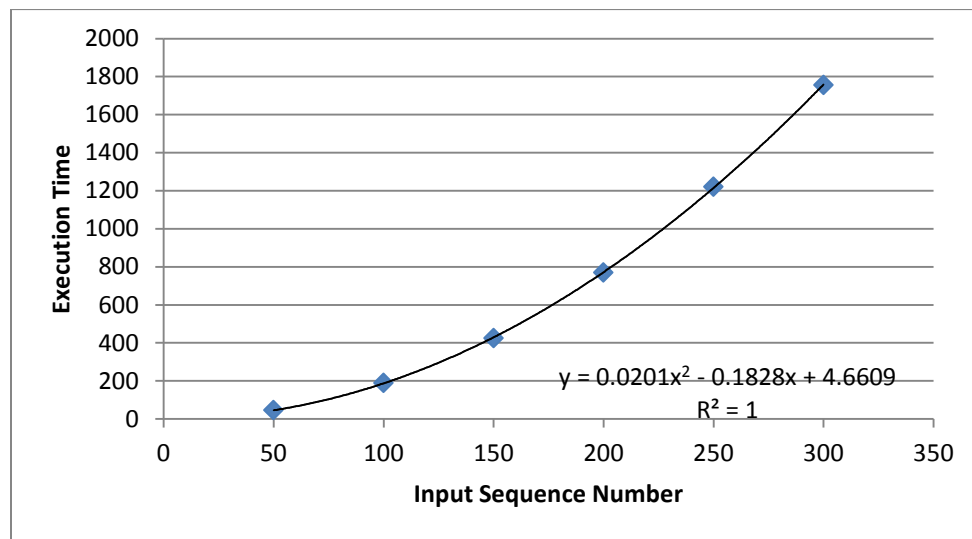
	TEMPEST	TEMPEST-CNXX
CPU	Intel E7450	Intel E7450
Cores	24	24
Memory	24.0GB	50.0 GB
Mem/Core	1 GB	2 GB
NIC (Enterprise)	HP NC 360T	n/a
NIC (Private)	HP NC373i	HP NC373i
NIC (Application)	Mellanox IPoIB	Mellanox IPoIB

Appendix C

Binomial fitting function for sequential SW-G jobs

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

$$R^2 = 1$$



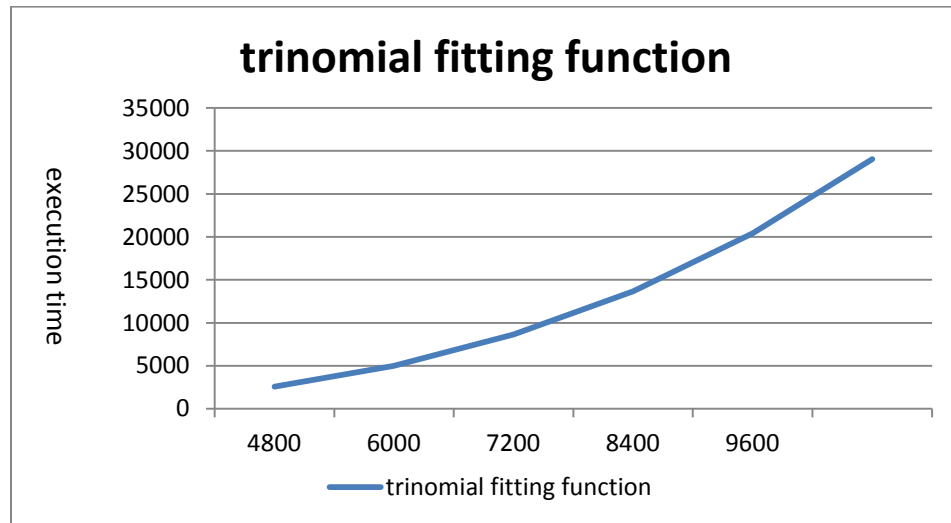
Appendix D

Aggregation methods	GroupAndAggregate	TwoApplyPerPartition	ApplyPerPartition	GroupBy	HierarchicalAggregation
4 nodes	4468		5674	6224	8505

8 nodes	2330	2313	2877	2902	4003
12 nodes	1897	1920	2371	2329	3200
16 nodes	1651	1671	2167	2078	3389

Appendix E

Trinomial fitting chart for sequential Matrix Multiplication jobs



Appendix F

Table 1: 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	79.7 8688	77.952 91	95.899 23	87.89 089	90.108 21	80.707 5	85.47 434	71.526 03	74.7924 3