Design Pattern for Typical Scientific Applications in DryadLINQ CTP

Hui Li, Yang Ruan, Yuduo Zhou, Judy Qiu, Geoffrey Fox

School of Informatics and Computing, Pervasive Technology Institute
Indiana University Bloomington
{lihui, yangruan, yuduo, xqiu, gcf}@indiana.com

***Abstract*— the design and implementation of higher level language interfaces are becoming increasingly important for data intensive computation. DryadLINQ is a runtime with a set of language extensions that enables programmers to develop applications processing large scale distributed data. It has been successfully used in a wide range of applications for the last five years. The latest release of DryadLINQ was published as a Community Technology Preview (CTP) in December 2010, and it contains new features and interfaces that can be customized to achieve better performances for applications and usability for developers. This paper presents three design patterns in DryadLINQ CTP that are applicable for a large class of scientific applications, exemplified by SW-G, Matrix-Matrix Multiplication and PageRank with real data.**

Keywords-component; Dryad; DryadLINQ; MapReduce; Design Pattern;

I. INTRODUCTION

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

However, its rigid and flat data processing paradigm does not directly support relational operations that have multiple related inhomogeneous input data stream. This limitation causes the difficulties and inefficiency when using Map-Reduce to simulate relational operations like Join which is very common in database. For example, the classic implementation of PageRank is very inefficient due to the simulating of Join with MapReduce causing lots of network traffic for the computation. Further optimization of PageRank requires developers to have sophisticated knowledge on web graph structure.

Dryad [2] is a general purpose runtime that supports data intensive applications on Windows platform. Dryad lies between MapReduce and database, which addresses some of the limitations of MapReduce systems. DryadLINQ [3] is the programming interface for Dryad that aims to enable developers make a wide range of data parallel applications in an easy and efficient way. It automatically translates LINQ programs written by .NET language into distributed computations run on top of Dryad system. For some applications, writing DryadLINQ programs are as simple as writing sequential programs. DryadLINQ and Dryad runtime optimizes the job execution plan and dynamically makes changes during the computation. All this process is handled by DryadLINQ/Dryad but transparent to users. For example, when implementing PageRank with DryadLINQ GroupAndAggregate() operator because it can dynamically construct a partial aggregation tree to reduce the number of intermediate records that transferring across the compute nodes during the computation.

In this paper, we will explore the easiness and efficiency of using DryadLINQ with three classic scientific applications and further classify them into three design patterns. The contributions of this paper are:

1) We studied the task granularity that improve LINQ's support for coarse-grain parallelization with DryadLINQ CTP data model and interface.
2) We demonstrated a hybrid parallel programming model not only utilizes parallelism in multiple nodes but also in multiple cores.
3) We evaluated different distributed grouped aggregation strategies in DryadLINQ CTP and studied the feature of input data that affect the efficiency of partial preaggregation.

The structure of this paper is as follows: section 2 illustrates DryadLINQ basic programming model. Section 3 describes implementation of three classic scientific applications (SW-G, Matrix-Matrix Multiplication and PageRank) with DryadLINQ CTP. Section 4 discusses related work, and section 5 concludes this paper. We note that as the latest LINQ to HPC has published in June 2011 and its interface changes much from DryadLINQ CTP, we will describe programming models in pseudo code.

II. DRYADLINQ PROGRAMMING MODEL

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.

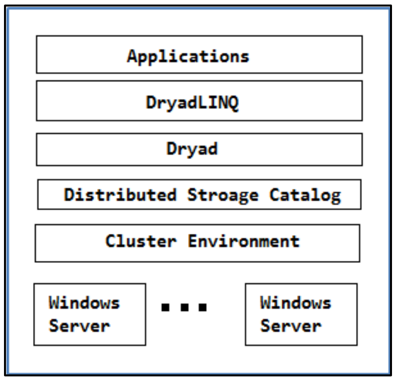

Fig.1 Software Stack for DryadLINQ CTP

A. *Pleasingly Parallel Programming Model*

Many pleasingly parallel applications are of the (Single Program Multiple Data) SPMD model. DryadLINQ supports a unified data and programming model in representation and processing of data. DryadLINQ data objects are collections of strong .NET type objects, which can be split into some partitions and distributed across the computers of cluster. These DryadLINQ data objects are represented as DistributedQuery<T> or DistributedData<T> objects to which the LINQ operators can apply. DryadLINQ applications can create the DistributeData<T> objects from existing data stored in DSC or convert from IEnumerable<T> objects with AsDistributed(), AsDistributedFromPartitions() operators. Then, these DryadLINQ data objects are partitioned and distributed to compute nodes. They can be processed by invoking the user defined function within Select() or ApplyPerPartition() operators. The sample code of this programming model is as follows:

```
Var inputs= inputDataSet.AsDistributedFromPartitions();
//construct DryadLINQ data objects
Var outputs= inputs.Select(distributedObject =>
UserDefinedAppFunction(distributedObject));
//execute DryadLINQ data objects
```

A wide range of pleasingly parallel applications can be implemented with the above DryadLINQ primitives which include CAP3 DNA sequence assembly application, High Energy Physics data analysis application [21] and the all pair gene sequences SW-G computation.

B. *Hybrid Parallel Programming Model*

Dryad is supposed to process coarse-granularity tasks for large scale distributed data. And it schedules tasks to the resources in the unit of compute nodes rather than cores. To make high utilization of multi-core resources of a HPC cluster, one approach is to perform the parallel computation with PLINQ on each node. The DryadLINQ provider can automatically transfer PLINQ query to parallel computation. The other approach is to apply the multi-core technologies in .NET like TPL, thread pool to the user-defined function within in the lambda expression of DryadLINQ query.

In above hybrid model, Dryad handles the parallelism between the cluster nodes while the PLINQ, TPL, and thread pool technologies deal with the parallelism on multi-core of each node. This hybrid parallel programing model in Dryad/DryadLINQ has been proved to be successful and applied to data clustering applications like GTM interpolation, MDS interpolation [21]. Most of the pleasingly parallel application can be implemented with this model to increase the overall utilization of cluster. The sample code of this programming model is provided as follows:

```
Var inputs= inputDataSet.AsDistributedFromPartitions();
Var outputs =
inputs.ApplyPerPartition(distributedPartitions =>
  MultiCoreProcessingFunction(distributedPartitions));
```

Further, the performance of applications with hybrid model can be affected not only by the parallel algorithm in node level, but also by the factors in core level like cache and memory bandwidth [17]. This paper will study the hybrid parallel programming model using matrix multiplication with different combinations of algorithms and multi-core technologies.

C. *Distributed Grouped 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 [30].

There are several approaches to implement distributed grouped aggregation. A direct one is to use the hash partition. It uses hash partition operator to redistributes 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 sample code is as follows:

```
Var groups = source.GroupBy(KeySelect);
//redistribute records to some groups by keys
Var reduced = groups.SelectMany(Reduce);
//aggregate records for each group
```

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 preaggregation. 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 [4]. 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.

III. IMPLEMENTATIONS

We implemented SW-G, Matrix-Matrix Multiplication and PageRank with DryadLINQ CTP and evaluated their performance on two Windows cluster (HPC R2 SP1). The hardware resources used in this paper are as follow:

Table 1. 32 nodes homogeneous HPC cluster TEMPEST

	TEMPEST	TEMPEST-CNXX
CPU	Intel E7450	Intel E7450
Cores	24	24
Memory	24.0GB	50.0 GB
Mem/Core	1 GB	2 GB

Table 2. 7 nodes inhomogeneous HPC cluster STORM

	STORM-CN01,CN02, CN03	STORM-CN04,CN05	STORM-CN06,CN07
CPU	AMD 2356	AMD 8356	Intel E7450
Cores	8	16	24
Memory	16GB	16GB	48GB
Mem/Core	2GB	1GB	2GB

A. SW-G Application

The Alu clustering problem [6] [7] 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. Notably, Alu clustering can be viewed as a classical case study for the capacity of computational infrastructures because it is not only of intrinsic biological interests, but also a problem of a scale that will remain as the upper limit of many other clustering problems in bioinformatics for the next few years, e.g. the automated protein family classification for a few millions of proteins predicted from large meta-genomics projects.

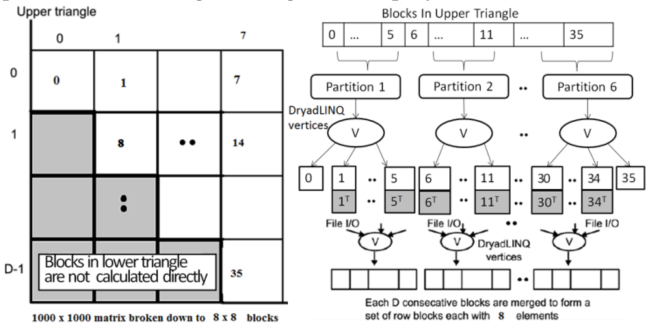

Fig. 2 DryadLINQ implementation of SW-G Application

We implemented the DryadLINQ application to calculate the pairwise SW-G distances in parallel for a given set of gene sequences. 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 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 Fig 2. Assuming there are 6 compute nodes, and we split the 36 blocks into 6 partitions each of which contains 6 blocks. Each Dryad tasks invokes the user defined function PerformAlignments() via ApplyPerPartition to apply Alu clustering computation to the 6 blocks that dispatched to them. The main component of DryadLINQ SW-G code is as follows:

```
DistributedQuery<OutputInfo> outputInfo = inputBlocks.AsDistributed().ApplyPerPartition(subBlocks Set => PerformAlignments4(subBlockSet, values,_inputFile, _sharepath, _outputFilePrefix, _outFileExtension, _seqAlignerExecName, _swgExecName))
```

1) Scheduling for inhomogeneous tasks

The SW-G is pleasingly parallel application, but pairwise SW-G computations are inhomogeneous in CPU time. That task of splitting all SW-G blocks into partitions with even number of blocks still has the workload balance issue when processing those partitions on homogeneous computational resources. We adopt two approaches to address this issue.

One approach is to construct SW-G blocks of input data by randomly selecting sequences. To verify this strategy, we manually generate a set of gene sequences with a given mean sequence length (400) with a variety of standard deviations following a normal distribution of the sequence lengths. We construct 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 shown in Fig 3, the randomly distributed input data can deliver a better performance than skew distributed one [1]. Note: there is a small performance increase of randomized distributed data when standard deviation increases from 0 to 150 which are due to nature randomness of SW-G program.

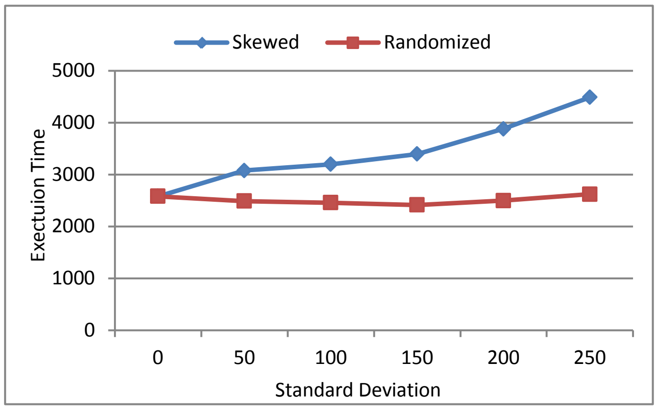

Fig. 3 Performance Comparison for Skewed Distributed and Randomized Distributed Data

The above approach in Fig 3 requires additional work from developer side. Another approach for this issue is to split the skewed distributed input data into many finer granularity tasks. To verify this, we constructed a set of gene sequence with a given mean sequence length (400) with varying standard deviations (50, 150, 250) and run these SW-G data set on our TEMPEST cluster with different number of partitions. As it shows in Fig 4, as the number of partitions increase the overall job turnaround time decrease for the three skewed distributed input data set. This is because the finer granularity tasks can achieve better workload balance among nodes by keeping dispatching available tasks to idle resources. However, when the number of partitions keeps increasing, the scheduling cost becomes the dominant factor on overall performance.

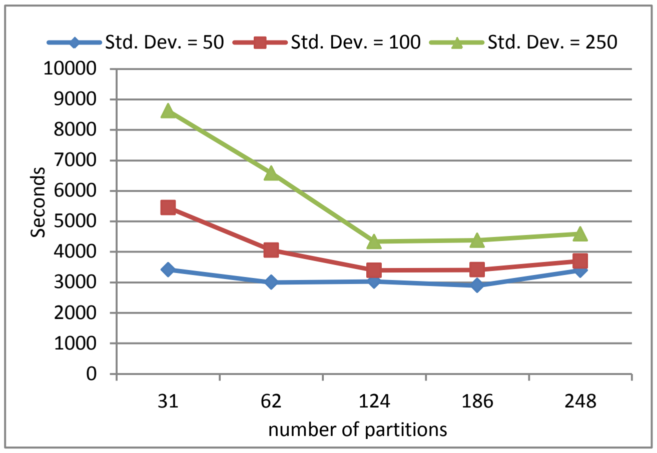

Fig 4. Performance Comparison for Skewed Distributed Data with Different Task Granularity.

2) Scheduling for inhomogeneous cluster

Clustering or extending existing hardware resources may lead to the problem of scheduling tasks on an inhomogeneous cluster with different CPU, memory, network capability between nodes [8]. Allocating work load to resources according to their computational capability is a solution, but it requires the runtimes to know the resources requirement of each job and availability of hardware resources. One other solution is to split the entire job into many finer granularity tasks and keep dispatching available tasks to idle computational resources.

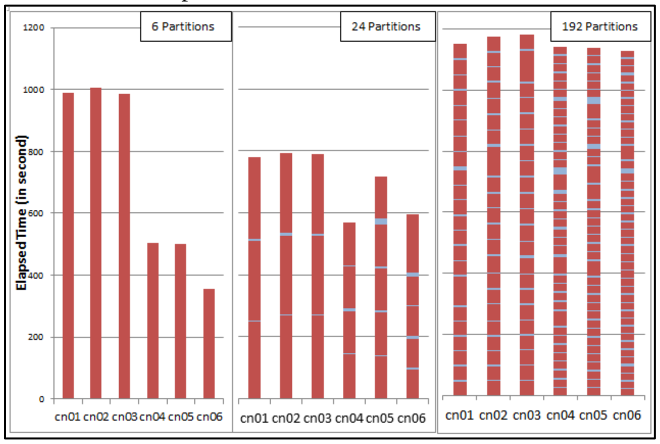

Fig. 5 CPU and scheduling time of same SW-G job with various partition granularities

We verify the second approach by executing the 4096 sequences SW-G jobs on the inhomogeneous HPC STORM with different partition granularity. Fig 5 shows the CPU time and task scheduling time of same SW-G job with different number of partitions: 6, 24, and 192. In the first SW-G job, an entire job is split into 6 partitions. The difference in CPU time for each task is caused by the difference in computational capability among nodes. It is clearly illustrated that finer partition granularity can deliver a better load balance on inhomogeneous computational nodes. However, it also shows that the task scheduling cost increases as the number of partitions increases.

B. Hybrid Parallel Programming Model

To explore the hybrid parallel programming model, we implemented DryadLINQ Matrix-Matrix Multiplication with three different algorithms and four multi-core technologies. The three matrix multiplication algorithms are: 1) row split algorithm, 2) row/column split algorithm, 3) two dimension block decomposition split in Fox algorithm [9]. The multi-core technologies are: PLINQ, TPL, thread pool, and parallel for.

In the experiments, we port multi-core technologies to different algorithms and study their overall performance. We use square matrix in this section by default, where each element is double number. The basic equation to calculate matrix-matrix multiplication is:

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}$$

1) Matrix-Matrix Multiplication algorithm

The row split algorithm splits matrix A by its rows. It scatters the rows blocks of matrix A across compute nodes, and then copies the whole matrix B to every compute node. Each Dryad task multiplies some rows blocks of A by entire B, and retrieves the output results to main program and combine them into matrix C.

The row/column split algorithm [18] splits matrix A by rows and split matrix B by columns. The column blocks of B are scattered across the cluster in advance. Then the whole computation is divided into several iterations, each of which multiplies the one row block of A to all the column blocks of B on compute nodes. The output of tasks within the same iteration will be retrieved to the main program to aggregate one row block of matrix C. The main program collects results in multiple iterations to generate the final output of matrix C.

The two dimensional block decomposition in Fox algorithm splits matrix A and matrix B into squared sub-blocks. These sub-blocks are dispatched to a squared process mesh with same scale. For example, let's assume to run the algorithm on a 2X2 processes mesh. Accordingly, matrix A and matrix B are split by both rows and columns and construct a 2X2 block mesh respectively. In each computation step, every process holds a block of matrix A and a block of matrix B and computes a block of matrix C. The data flow of the algorithm is shown in Fig 6.

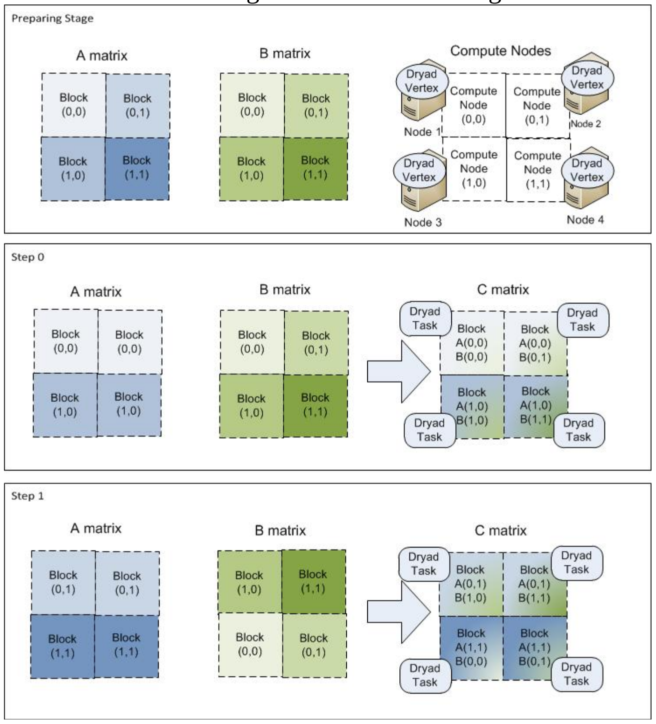

Fig. 6 Program Flow for DryadLINQ Matrix-Matrix Multiplication in Fox Algorithm

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

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

We evaluate the three algorithms by running matrix-matrix multiplication jobs with various input data size from 2400 to 19200 with only one core per node on 16 compute nodes (4X4 mesh). Fig 7 shows that the Fox and RowSplit algorithm can achieve the better speed up than RowColumnSplit. Comparing with the other two algorithms, the Fox algorithm has finer granularity tasks as it only calculate one sub-block of matrix A and matrix B. This will cause the high cache hitting rate during the computation. The row/column algorithms perform the worst due to cost to launch Dryad vertex in every iteration.

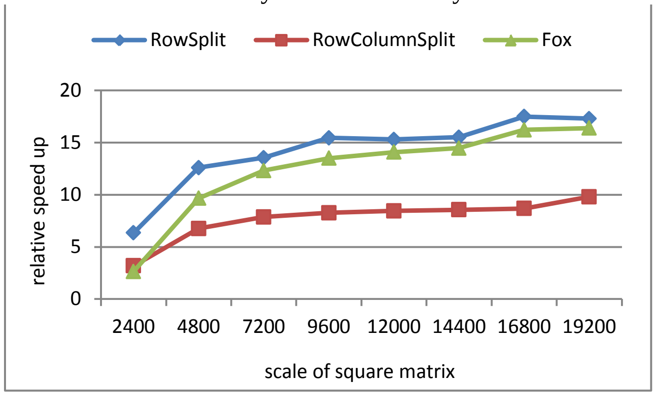

Fig 7 Speed-up of three algorithms with various sizes of data

2) *Parallelism in core level*

We evaluated the multi-core technologies in .NET 4 by running matrix-matrix multiplication jobs with various size of input data from 2400 * 2400 to 19200 * 19200 on a 24-core Windows server. Fig 8 shows the performance results of matrix-matrix multiplication jobs for three different multi-core technologies. As illustrated in Fig 8, the PLINQ has the best performance compare with other approaches.

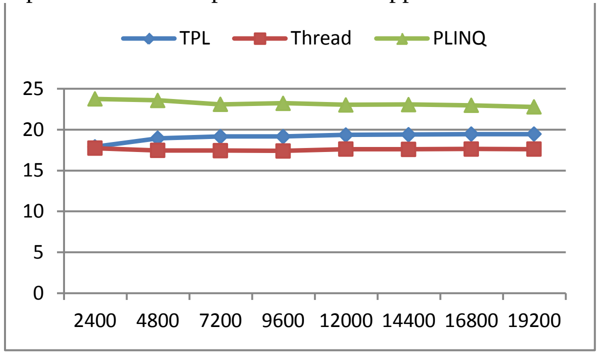

Fig. 8 Speed up for Different Method of Multi-Core Parallelism on 24 cores Compute Node

3) *Port multi-core tech into Dryad task*

We port the above multi-core technologies into the three matrix-matrix multiplication algorithms. Fig 9 shows the relative speed up for three algorithms combined with different multi-core technologies run on 16 compute nodes with 24 cores per node. Fig 9 shows that the Fox algorithm does not perform well as RowSplit in Fig 7. In matrix-matrix multiplication, the computation cost O(n^3) increases faster than the communication cost O(n^2). Thus after porting multi-core into Dryad task, the task granularity for the row split and row/column split algorithm becomes finer as well, which alleviates the low cache hit rate issue for coarse-granularity task.

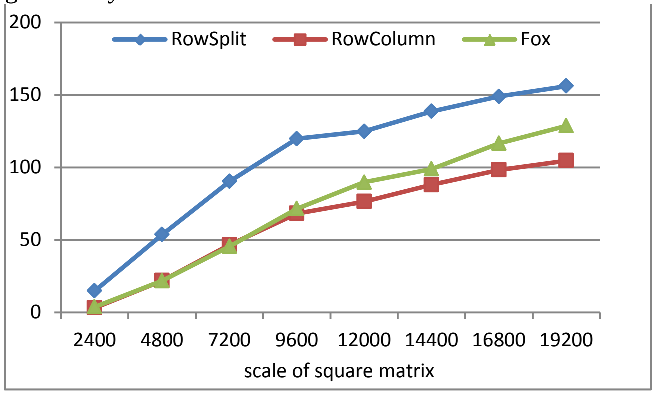

Fig 9. Relative Speed up for Different Combination of Algorithms and Multi-core Technologies

Fig 10 is the CPU and network utilization static data of the three parallel algorithms from HPC resource manager.

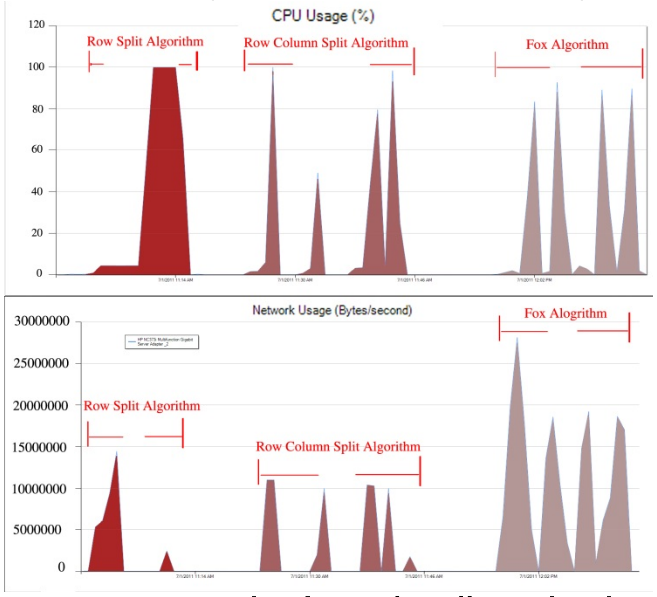

Fig 10. CPU & Network Utilization for Different Algorithms

C. *Distributed Grouped Aggregation*

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.

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.

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

1) *The Hash Partition Approach*

A simple way to implement PageRank with hash partition approach in DryadLINQ CTP is to use GroupBy() and Join() as follows:

```
for (int i = 0; i < maxNumIteration; i++)    {
newRanks  =  pages.Join(ranks,  page  =>  page.key,  rank  =>
rank.key,(page, rank) => page.links.Select(key => new Rank(key,
rank.value / (double)page.links.Length())))
.SelectMany(ranks => ranks).GroupBy(rank => rank.key).
    Select(group => new Rank(group.Key, group.Select(rank =>
      rank.value).Sum() * 0.85 + 0.15 / (double)numUrls));
    ranks = newRanks;    }
```

The **Page** objects are used to store the structure of web graph. Each element **Page** in collection pages contains a unique identifier number page.key and a list of identifiers specifying all the pages in the web graph that **page** links to. We construct the DistributedQuery<Page> **pages** objects from the AM files with function BuildPagesFromAMFile(). The **rank** object is a pair specifying the identifier number of a page and its current estimated rank value. In each iteration the program JOIN the **pages** with **ranks** to calculate the partial rank values. Then GroupBy() operator 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].

2) *The Hierarchical Aggregation Approach*

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

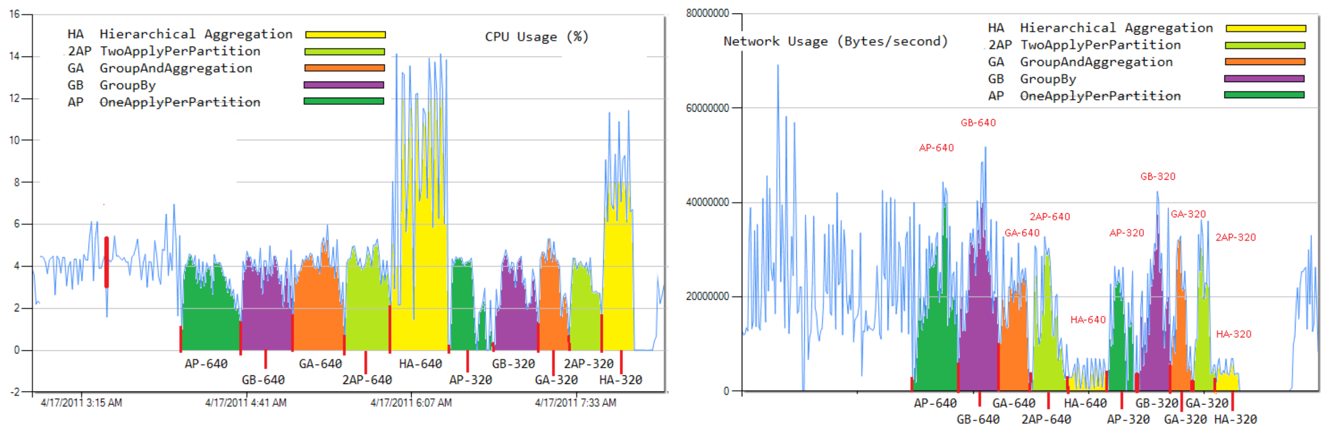

Fig. 12 CPU and Network Utilization for Different Aggregation Strategies

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.

3) *The Aggregation Tree Approach*

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. We also implemented PageRank with GroupAndAggregate() operator that enable aggregation tree optimization.

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.

4) *Performance Analysis*

We evaluate performance of the three approaches by running PageRank jobs with various sizes of input data on 17 compute nodes on TEMPEST. Fig 11 shows that the aggregation tree and hierarchical aggregation approaches outperform hash partition approach. Fig.12 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.

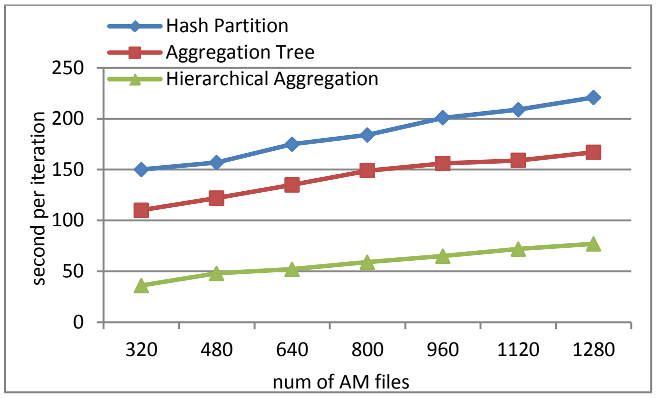

Fig. 11 Time in sec to compute PageRank per iteration with three aggregation approaches with clue-web09 data set on 17 nodes of TEMPEST

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:

$$DRP = \frac{number\ of\ output\ tuples}{number\ of\ input\ tuples} \quad (1)$$

Table 3. Data reduction ratios for different PageRank approaches with Clue-web09 data set

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

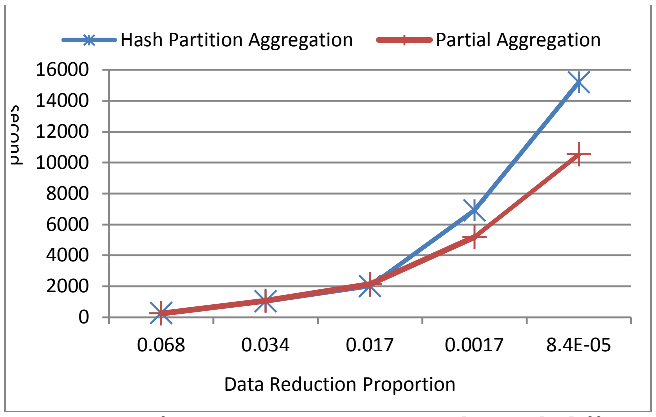

Fig. 13 Time for two aggregation approaches with different DRP values.

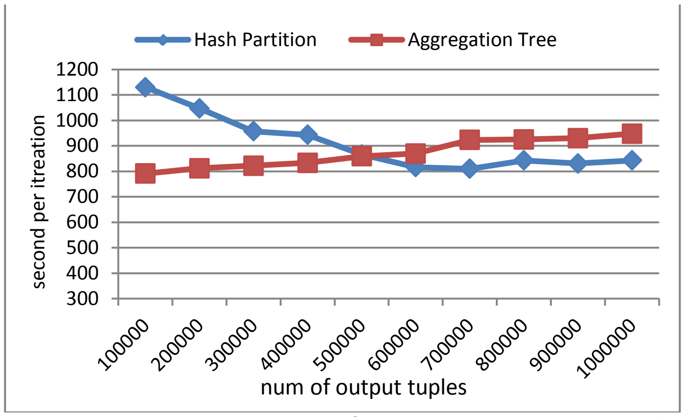

Fig. 14 Time per iteration for two aggregation approaches with different number of output tuples (from 100000 to 1000000) when number of input tuples is 4.3 billion

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:

$$\begin{cases} O\left(\frac{M}{M}\right) = O(1) \\ O\left(\frac{M+N}{M}\right) = O(1 + N * DRP) \end{cases} \qquad (2)$$

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 thousands 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 deliever performance as well as word count [4].

To quantatively analysis of how DRP affects the aggregation performance, we compare two aggregation approraches with a set of web graphes with different DRP by fixing the number of output tuples and changing that of input tuples. It is illustrated in Fig. 13 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. 14 shown the time per iteration of PageRank jobs of web graphes with different number of output tuples when that of input tuples fixed. Fig.13 and 14 show that different grouped aggregation approaches fit well with different DRP range of input data.

IV. Related Work

In this paper, we illustrate three design patterns in DryadLINQ CTP for classic scientific applications with the focus on easiness of programming and the performance of applications. To our knowledge, these patterns have covered a wide range of distributed scientific applications.

A. *Pleasingly Parallel Application*

We have shown that the developers can easily control the partition granularity with DryadLINQ interface to solve work load balance issue. In batch job scheduling systems like PBS, programmers have to manually group/un-group or split/combine input data to control task granularity. Hadoop provides the interface that allows developers to control task granularity by defining the size of input records in HDFS. This is a good improvement, but it still requires developers to know and define the logic format of input data in HDFS. DryadLINQ provides a simplified data model and interface for this issue based on existing .NET platform.

B. *Hybrid Parallel Programming*

The hybrid parallel programming must combine inter node distributed memory parallelization with intra node shared memory parallelization. MPI + MPI/OpenMP/Threading are the hybrid programming model utilized in the high performance computing. Paper [31] discusses the hybrid parallel programming paradigm using MPI.NET and TPL, CCR (Concurrency and Coordination Runtime) on Windows HPC server. It shows that the efficiency of hybrid parallel programming model have to do with the task granularity, while parallel overhead is mainly caused by synchronization and communication. Our paper is focus on Dryad, which is intended for the data intensive computation.

Twister [14] and Hadoop can also make use of multiple core system by launching multiple task daemons on each compute node. Typically the number of task daemons is equal to that of cores on each compute node, but it can be less or more than number of cores on each node as well. The developers do not need to know the difference of underline hardware resources, because the runtime provide the identical programming interface to dispatch tasks to task daemons across cluster automatically.

C. *Distributed Grouped Aggreagtion*

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.

V. Discussion and Conclusion

We have presented in this paper three typical programming models, which are applicable to a large class of applications in science domain using DryadLINQ CTP. Further, we discussed the issues that affect the performance of applications implemented with these programming models.

We investigated the hybrid parallel programming model with the matrix-matrix multiplication. We have shown that porting multi-core technology can increase the overall performance significantly. And we observed that different combination of parallel algorithm in nodes level and multi-core technology in core level will affect overall performance of application. In matrix-matrix multiplication, the CPU cost O(n^3) increase faster than the memory and bandwidth cost O(n^2). Thus the CPU cache and memory paging is more important than network bandwidth to scale up matrix-matrix multiplication.

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 preaggregation requires more memory than hash partition. 2) The bandwidth of network. Hash partition has larger network traffic overhead than partial preaggregation. 3) The choice of implementation of partial preaggregation 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.

Acknowledgment

We want to thank John Naab and Ryan Hartman from IU PTI for setting up the Windows HPC cluster, and Thilina Gunarathne, Stephen Tak-lon Wu from IU CS for providing the SW-G application and data. This work is partially funded by Microsoft.

References

[1] Salsa Group, Applicability of DryadLINQ to Scientific Applications. 2010.

[2] 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.

[3] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsso, P. Gunda, and J. Currey, “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computig Using a High-Level Language”, Symposium o Operating System Design and Implementation (OSDI), CA, December 8-10, 2008.

[4] Y. Yuan, P. Gunda, M. Isard, “Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations”, SOSP’09, October 11-14, 2009, Big Sky, Montana, USA.

[5] Introduction to Dryad, DSC and DryadLINQ. 2010

[6] 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.

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

[8] H. Li, H. Yu, X. Li, “A Lightweight Execution Framework for Massive Independent Tasks” MTAGS’08 workshop at SC08, October 2008.

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

[10] J. Qiu, G. Fox, H. Yuan, S. Bae, “Parallel Data Mining on Multicore Clusters”, gcc, pp.41-49, 2008 Seventh International Conference on Grid and Cooperative Computing, 2008

[11] Y. Yu, P. Kumar, M. Isard, “Distributed Aggregation for Data Parallel Computing”, SOSP’ 09, October 11-14, 2009, Big Sky, Montana, USA.

[12] Haloop, http://code.google.com/p/haloop/

[13] Hadoop, http://hadoop.apache.org/

[14] Twister, http://www.iterativemapreduce.org/

[15] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, G. Fox, “Twister: A Runtime for Iterative MapReduce”

[16] ClueWeb09: http://boston.lti.cs.cmu.edu/Data/clueweb09/

[17] J. Qiu, G. Fox, H. Yuan, S. Bae, “Performance of Multicore Systems on Parallel Dataming Services” CCGrid 2008.

[18] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, “Data cube: A relational aggregation operator geeralizig group-by, cross-tab, and sub-totals. Data Mining and Knowledge Discovery, 1(1), 1997.

[19] Jaliya Ekanayake, “Architecture and Performance of Runtime Environments for Data intensive Scalable Computing”, Dec 2010

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

[21] J. Ekanayake, A. Soner, T. Gunarathen. and G, Fox, C. Poulain, “DryadLINQ for Scientific Analysis”.

[22] Partial pre-aggregation in relational database queries, Per-Ake Larson, Redmond, WA (US) Patent No.: US 7,133,858,B1

[23] 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.

[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar and A. Goldberge “Quincy: Fair Scheduling for Distributed Computing Clusters”

[25] OpenMPI http://www.open-mpi.org/

[26] R. Chen, X. Weng, B. He, M. Yang “Large Graph Processing in the Cloud” SIDMOD’10 June 6-11,2010, Inidanapolis, Indiana, USA

[27] S. Hee, J. Choi, J. Qiu and G. Fox “Dimension Reduction and Visualization of Large High-dimensional Data via Interpolation” HPDC 2010, Chicago, Illinois, Jun. 2010.

[28] S. Helmer, T. Neumann, G. Moerkotte, “Estimating the Output Cardinality of partial Preaggregation with a Measure of Clusteredness” Proceeding of the 29th VLDB Conference, Berlin, Germany, 2003

[29] T. Gunarathne, T. Wu, J. Qiu, G. Fox, “Cloud Computing Paradigms for Pleasingly Parallel Biomedical Applications” Conference on eScience (eScience ’08), Indianapolis, IN 2008

[30] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski. “Pregel: A System for Large-Scale Graph Processing” SIGMOD’10, June 6-11, 2010, Indianapolis, Inidana, USA.

[31] J. Qiu, S. Beason, S. Bae, S. Ekanayake, G. Fox, “Performance of Windows Multicore System on Threading and MPI” GCC 2010