Design Patterns for 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 data flow programming language interfaces are becoming increasingly important for data intensive computation. DryadLINQ is a declarative, data-centric language that enables programmers to address the Big Data issue in the Windows Platform. DryadLINQ 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 contains new features and interfaces that can be customized in order to achieve better performances within applications and in regard to usability for developers. This paper presents three design patterns in DryadLINQ CTP that are applicable to a large class of scientific applications, exemplified by SW-G, Matrix-Matrix Multiplication and PageRank with real data.

Categories and Subject Descriptors
D.3.2 [**Programming Language**]: Data-flow languages

General Terms
Performance, Design, Languages.

Keywords
Dryad, DryadLINQ, MapReduce, Design Pattern, SW-G, Matrix Multiply, PageRank

1. INTRODUCTION
We are in a Big Data era. The rapid growth of information in science requires the processing of large amounts of scientific data. One proposed solution is to apply data flow languages and runtimes to data intensive applications [1]. The primary function of data flow languages and runtimes is the management and manipulation of data. Sample systems include the MapReduce [2] architecture pioneered by Google and the open-source implementation called Hadoop [3].

The MapReduce systems provide higher level programming languages that express data processing in terms of data flows. These systems can transparently deal with scheduling, fault tolerance, load balancing and communications while running jobs. The MapReduce programming model has been applied to a wide range of applications and has attracted enthusiasm from distributed computing communities due to its ease of use and efficiency in processing large scale distributed data

However, the rigid and flat data processing paradigm of the MapReduce programming model prevents MapReduce from processing multiple, related heterogeneous datasets. A higher level programming language, such as Pig or Hive, can solve this issue to some extent, but is not efficient because the relational operations, such as Join, are converted into a set of Map and Reduce tasks for execution. For example, the classic MapReduce PageRank is very inefficient as the Join step in MapReduce PageRank spawns a very large number of Map and Reduce tasks during processing. Further optimization of MapReduce PageRank requires developers to have sophisticated knowledge of the web graph structure.

Dryad [4] is a general purpose runtime that supports the processing of data intensive applications within the Windows platform. It models programs as a directed acyclic graph of the data flowing between operations. Thus, it is able to addresses some of the limitations that exist in the MapReduce systems. DryadLINQ [5] is the declarative, data flow programming language for Dryad. The DryadLINQ compiler can automatically translate the LINQ (Language-Integrated Query) programs written by .NET language into distributed, optimized computation steps that are run on top of the Dryad cluster. For some applications, writing the DryadLINQ distributed programs are as simple as writing a series of SQL queries. In complex cases, developers can port the application programs or user-defined functions into the lambda expression of the LINQ queries.

In this paper, we investigate the applicability and efficiency of using DryadLINQ to develop scientific applications. Then, we abstracted them into three design patterns. The contributions of this paper are as follows:

1. We studied the task granularity in order to improve LINQ's support for coarse-grain parallelization with the DryadLINQ CTP data model and interface.

2. We demonstrated that a hybrid parallel programming model not only utilizes parallelism in multiple nodes, but also in multiple cores.

3. We investigated the three distributed grouped aggregation approaches and the feature of input data that affects the efficiency of these approaches.

The structure of this paper is as follows. Section 2 illustrates the DryadLINQ basic programming model. Section 3 describes the implementation of the three classic scientific applications (SW-G, Matrix-Matrix Multiplication and PageRank) using DryadLINQ CTP. Section 4 discusses related work, while Section 5 concludes the paper. As the latest LINQ to HPC was published in June 2011, and its interface changed drastically from DryadLINQ CTP, we will describe the programming models using pseudo code.

DataCloud-SC'11, November 14, 2011, Seattle, Washington, USA.

2. DRYADLINQ PROGRAMMING MODEL

Dryad, DryadLINQ and DSC [6] are a set of technologies that support the processing of data intensive applications in the Windows platform. The software stack for these technologies is shown in Figure 1.

Dryad is a general purpose runtime that supports the processing of data intensive applications in Windows platform. A Dryad job is represented as a directed acyclic graph (DAG), which is called the Dryad graph. One Dryad graph consists of vertices and channels. A graph vertex is an independent instance of the data processing program in a certain step. Graph edges are the channels transferring data between the vertices. The Distributed Storage Catalog (DSC) is the component that works with the NTFS in order to provide data management functionalities, such as data sets storage, replication and load balancing within the HPC cluster.

DryadLINQ is a high level programming language and compiler for Dryad. The DryadLINQ API is based on the LINQ programming model. It takes advantage of the standard query operators defined within the LINQ and adds query extensions specific to Dryad. Developers can easily apply LINQ operators, such as Join or GroupBy, to a set of .NET data objects, which increase the speed of the development of the data intensive applications.

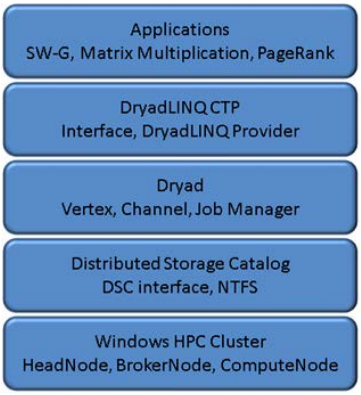

Fig.1: Software Stack for DryadLINQ CTP

2.1. 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 the representation and processing of pleasingly parallel applications. DryadLINQ data objects are collections of strong .NET type objects, which can be split into partitions and distributed across 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 the existing data stored in the DSC or convert it from the *IEnumerable<T>* objects using *AsDistributed()* and *AsDistributedFromPartitions()* operators. Then, these DryadLINQ data objects are partitioned and distributed to the nodes. Developers can deal with these distributed DryadLINQ data objects by invoking the user-defined function within the *Select()* or *ApplyPerPartition()* operators. The pseudo code for this programming model is as follows:

```
Var inputs= inputDataSet.AsDistributedFromPartitions();
//Construct DryadLINQ Distributed Data Objects--inputs
Var outputs= inputs.Select(distributedObject =>
User_Defined_Function(distributedObject));
//Process DryadLINQ Distributed Data Objects with UDF
```

A wide range of pleasingly parallel applications can be implemented using the above DryadLINQ primitives [7], which include the CAP3 DNA sequence assembly application, High Energy Physics data analysis application and the all pair gene sequences SW-G computation.

2.2. Hybrid Parallel Programming Model

Dryad is supposed to process coarse-granularity tasks for large scale distributed data. It usually schedules tasks for the resources in the unit of compute nodes rather than the cores. In order to increase the utilization of the multi-core Windows cluster, one direct approach is to invoke PLINQ (parallel LINQ) queries within the lambda expression of the DryadLINQ query. This approach is not only convenient, but, also, efficient as the LINQ query is naturally built within the DryadLINQ query. The other approach is to apply the multi-core technologies in .NET, such as TPL, and the thread pool to the user-defined function within in lambda expression of the DryadLINQ query. The pseudo code for this programming model is as follows:

```
Var inputs= inputDataSet.AsDistributedFromPartitions();
//Construct DryadLINQ Distributed Data Objects--inputs
Var outputs =
inputs.ApplyPerPartition(distributedObject =>
distributedObject.AsParallel().Select(parallelObject=>
       User_Defined_Function(parallelObject)));
 //Process DryadLINQ Distributed Data Object with PLINQ
```

In the above hybrid model, Dryad handles the parallelism between the cluster nodes, while the PLINQ, TPL and thread pool technologies deal with the parallelism on the multi-core of each node. The hybrid parallel programming model in Dryad/DryadLINQ has been proven to be successful and has been applied to data clustering applications [7], such as the GTM interpolation and MDS interpolation. Most of the pleasingly parallel applications can be implemented using this model.

2.3. Distributed Grouped Aggregation

The GROUP BY operator in the parallel database is often followed by the aggregate function, which groups the input records into partitions by keys and then merges the records for each group using certain attribute values. This common pattern is called the distributed grouped aggregation. Sample applications for this pattern include sales data summarizations, log data analysis and social network influence analysis [8] [9].

Several approaches exist by which to implement the distributed grouped aggregation. A direct approach is to use the hash partition operator to redistribute the records to the compute nodes so that identical records are stored on the same node. After that this approach merges the records of each group on each node.

The implementation of the hash partition is simple, but creates a large amount of network traffic when the number of input records is very large. A common way to optimize this approach is to apply pre-aggregation, which aggregates the local records of each node and then hash partitions the aggregated partial results across a cluster based on their key. This approach is better than the direct hash partition because the number of records transferred across the

cluster becomes much smaller after the local aggregation operation.

Two additional ways exist by which to implement the pre-aggregation: 1) hierarchical aggregation and 2) an aggregation tree [10]. A hierarchical aggregation usually contains two or three aggregation layers, each having an explicit synchronization phase. An aggregation tree is a tree graph that guides a job manager to perform the pre-aggregation for the many subsets of the input records. The workflow of the three distributed grouped aggregation approaches is shown in Figure 2.

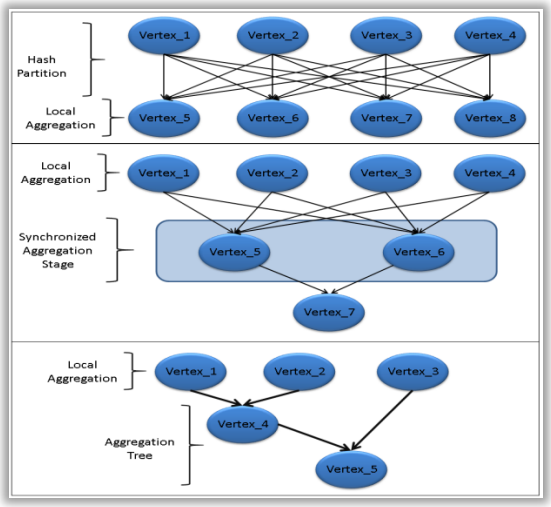

Fig. 2: Three Distributed Grouped Aggregation Approaches: Hash Partition, Hierarchical Aggregation, and Aggregation Tree.

3. IMPLEMENTATIONS

We implemented SW-G, Matrix-Matrix Multiplication and PageRank using the DryadLINQ CTP and then evaluated their performances on two Windows HPC clusters and one Linux cluster. The hardware resources used in this paper are as follows:

Table 1: 32 Nodes Homogeneous HPC Cluster TEMPEST

	TEMPEST	TEMPEST-CNXX
CPU	Intel E7450	Intel E7450
Cores	24	24
Memory	24.0 GB	50.0 GB
Memory/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	16 GB	16 GB	48 GB
Memory/Core	2 GB	1 GB	2 GB

Table 3: 230 Nodes Homogeneous Linux Cluster Quarry

	Head Node	PG-XX
CPU	Intel E5335	Intel E5335
Cores	8	8
Memory	8 GB	16 GB
Memory/Core	1 GB	2 GB

3.1. Pleasingly Parallel Application

The Alu clustering problem [11] [12] is one of the most challenging problems when sequencing clustering because Alus represent the largest repeat families in the human genome. About one million copies of the Alu sequence exist in the human genome. Most insertions can be found in other primates and only a small fraction (~7000) are human-specific. This feature indicates that the classification of Alu repeats can be deduced solely from the one 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 interest, but, also, a problem on 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 million proteins predicted from large meta-genomics projects.

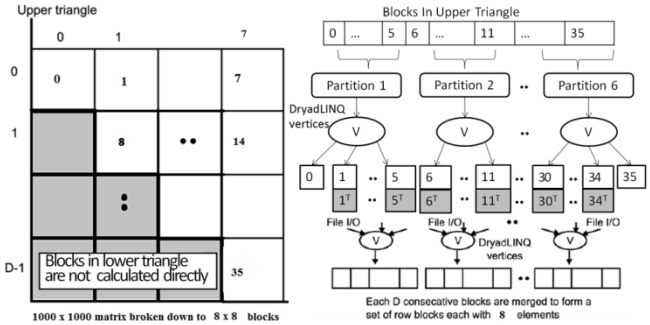

Fig. 3: DryadLINQ Implementation of SW-G Application

We implemented the DryadLINQ application in order to calculate the pairwise SW-G distances in parallel for a given set of gene sequences. In order to clarify our algorithm, we considered an example with 10,000 gene sequences, which produced a pairwise distance matrix of 10,000 × 10,000. We decomposed the overall computation into a block matrix D of 8 × 8, each block containing 1250 × 1250 sequences. Due to the symmetry of the distances D(i,j) and D(j,i), we only calculated the distances in the 36 blocks of the upper triangle of the block matrix as shown in Figure 3. These 36 blocks were constructed as 36 DryadLINQ distributed data objects. Then, our program split the 36 DryadLINQ objects into 6 partitions, which spawned 6 DryadLINQ tasks. Each Dryad task invoked the user-defined function *PerformAlignments()* in order to process the six blocks that were dispatched to each Dryad task. One should bear in mind that different partition scheme will cause different task granularity. The DryadLINQ developers can control task granularity by simply specify the number of partition with *RangePartition()* operator.

3.1.1. *Workload Balance for Inhomogeneous Tasks*

The workload balance is a common issue when scheduling inhomogeneous tasks on homogeneous resources or vice versa. In order to solve this issue, Dryad provides a unified data model and flexible interface for developers to tune task granularity. The following experiments will study workload balance issue in DryadLINQ SW-G application.

The SW-G is a pleasingly parallel application, but the pairwise SW-G computations are inhomogeneous in CPU time. The task of splitting all of the SW-G blocks into partitions with an even number of blocks still experiences a workload balance issue when processing the partitions on the homogeneous computational resources. One approach for this issue is to split the skewed distributed input data into many finer granularity tasks. In order to verify this approach, we constructed a set of gene sequences with a

given mean sequence length (400) using varying standard deviations (50, 150, 250). Then, we ran the SW-G dataset on the TEMPEST cluster using a different number of data partitions. As shown in Figure 4, as the number of partitions increased, the overall job turnaround time decreased for the three skewed distributed input datasets. This phenomenon occurs because the finer granularity tasks can achieve the better overall system utilization by dynamically dispatching available tasks to idle resources. However, when the number of partitions continually increases, the scheduling costs become the dominant factor in regard to overall performance.

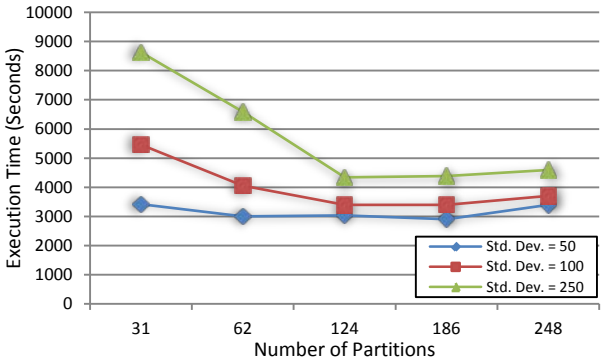

Fig. 4: Performance Comparison for Skewed Distributed Data with Different Task Granularity.

3.1.2. *Workload Balance for Inhomogeneous Cluster*

Clustering or extending existing hardware resources may lead to the problem of scheduling tasks on an inhomogeneous cluster with different CPUs, memory and network capabilities between nodes [13]. Allocating the workload to resources according to their computational capability is a solution, but requires the runtimes to know the resource requirement of each job and availability of hardware resources. Another solution is to split the entire job into many finer granularity tasks and dispatch available tasks to idle computational resources.

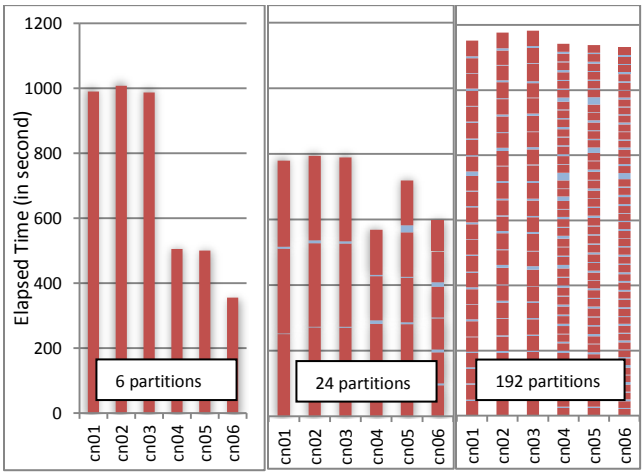

Fig. 5: CPU and Scheduling Time of the Same SW-G Job with Various Partition Granularities

We verified the second approach by executing 4,096 sequences for SW-G jobs on the inhomogeneous HPC STORM using different partition granularities. Figure 5 shows the CPU and task scheduling times of the same SW-G job with a different number of partitions: 6, 24 and 192. In the first SW-G job, the entire job was split into six partitions. The difference in CPU time for each task was caused by the difference in the computational capability of the nodes. The second and third jobs in Figure 5 clearly illustrate that finer partition granularity can deliver a better load balance on the inhomogeneous computational nodes. However, it also showed that the task scheduling cost increased as the number of partitions increased.

3.1.3. *Compare with Hadoop*

As shown in Figures 4 and 5, the task granularity is important for the workload balance issue in DryadLINQ. Further, we compared the task granularity issue of DryadLINQ with that of Hadoop. The DryadLINQ/PLINQ SW-G experiments were run with 24 cores per node on 32 nodes in TEMPEST. The input data was 10,000 gene sequences. The number of DryadLINQ tasks per vertex ranged from 1 to 32. The Hadoop SW-G experiments were run with 8 cores per node on 32 nodes in Quarry. Eight mappers and one reducer were deployed on each node. The number of map tasks per mapper ranged from 1 to 32. As shown in Figure 6, when the number of tasks per vertex was bigger than 8, the relative parallel efficiency of DryadLINQ jobs decreased noticeably. This decrease occurred because the number of tasks per vertex was bigger than 8 and the number of SW-G blocks allocated to each DryadLINQ task was less than 12, which is only half of the number of cores in each node in TEMPEST. Dryad can run only one DryadLINQ task on each compute node. Thus the relative parallel efficiency was low for fine task granularity in DryadLINQ.

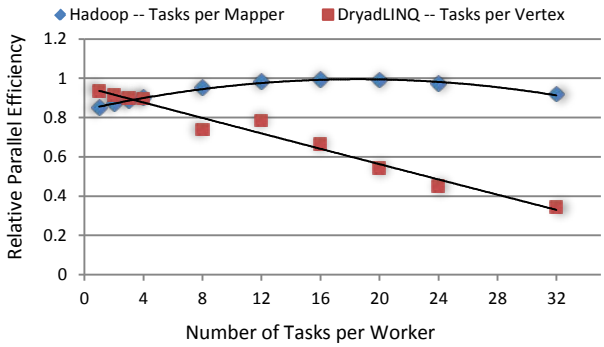

Fig. 6: Relative Parallel Efficiency of Hadoop and DryadLINQ with Different Task Granularity.

3.2. Hybrid Parallel Programming Model

In order to explore the hybrid parallel programming model, we implemented the DryadLINQ Matrix-Matrix Multiplication using three algorithms and three multi-core technologies. The three matrix multiplication algorithms were: 1) row partition algorithm, 2) row/column partition algorithm and 3) two dimension block decomposition in the Fox algorithm [14]. The multi-core technologies were: 1) PLINQ, 2) TPL, and 3) Thread Pool.

3.2.1. *Matrix-Matrix Multiplication Algorithms*

The row partition algorithm split matrix A by rows. It scattered the rows of the blocks of matrix A across the compute nodes, and then copied all of matrix B to every compute node. Each Dryad task multiplied some of the rows of the blocks of matrix A by the entire matrix B, and then retrieved the partial output results to the main program and added these results to the corresponding rows of blocks of matrix C.

The row/column partition algorithm [15] split matrix A into rows and matrix B into columns. The column blocks of matrix B were scattered across the cluster in advance. Then, the entire computation was divided into several steps. In each step, one task multiplied one row of blocks of matrix A by one column of blocks of matrix B on the compute node. The output results were sent back to the main program and aggregated into one row of blocks of matrix C. The main program collected the results in all of the steps in order to generate the final results of matrix C.

The two dimensional block decomposition in the Fox algorithm is called the Broadcast-Multiply-Roll (BMR) algorithm. In the initial stage of the Fox algorithm, it splits matrix A and matrix B into squared sub-matrices. These sub-matrices were scattered into a square mesh of processors. Figure 7 is the work flow of the Fox algorithm on a mesh of 2*2 nodes.

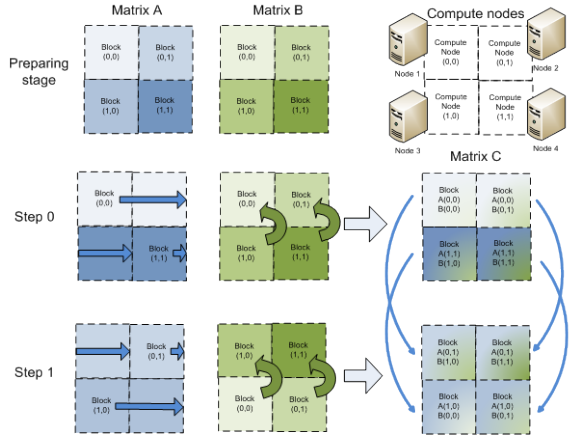

Fig. 7: Work Flow of the Fox Algorithm on a Mesh of 2*2 Nodes

We evaluated the three algorithms by running matrix multiplication jobs with various matrices sizes whose scales ranged from 2,400 to 19,200. By default, we used a square matrix in the experiments and the elements of the matrices were double numbers. The experiments were run with one core per node on the 16 compute nodes in TEMPEST.

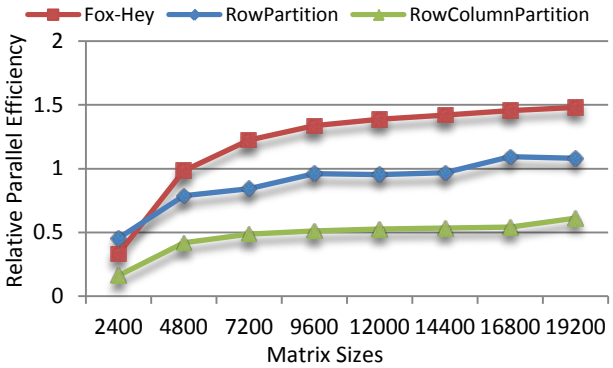

Fig. 8: Relative Parallel Efficiency of the Three Algorithms

As shown in Figure 8, the Fox algorithm performed better than the other two algorithms for the large matrices. The RowPartition algorithm had the simplest logic and least amount of scheduling costs. It did not perform as well as the Fox algorithm due to the fact that it did not parallelize the communications when broadcasting matrix A and scattering sub-matrices B over the cluster. The RowColumnPartition algorithm performed worse than the RowPartition as it had additional startup costs in the multiple steps.

3.2.2. *Parallelism in the Core Level*

We evaluated the multi-core technologies in .NET 4 by running matrix-matrix multiplication jobs with various matrices sizes whose scales ranged from 2,400 * 2,400 to 19,200 * 19,200 on a 24-core machine. Figure 9 shows the performance results for the three multi-core technologies. As illustrated in Figure 9, the PLINQ had the best performance compared to the other technologies.

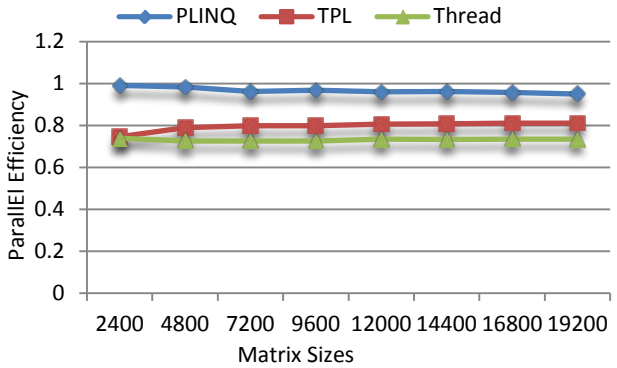

Fig. 9: Parallel Efficiency for Different Technologies of Multi-Core Parallelism on the 24 Core Compute Node

3.2.3. *Porting Multi-core Tech into Dryad Tasks*

We investigated the overall performance of the three matrix multiplication algorithms when porting PLINQ to the DryadLINQ tasks. The experiments were run with 24 cores per node on 16 nodes in TEMPEST. The matrices sizes ranged from 2,400 * 2,400 to 19,200 * 19,200. As shown in Figure 9 and 10, 1) the PLINQ version was much faster than the sequential version, 2) each of the three algorithms scaled out for the large matrices and 3) the Fox algorithm performed better than the other algorithms for the large matrices.

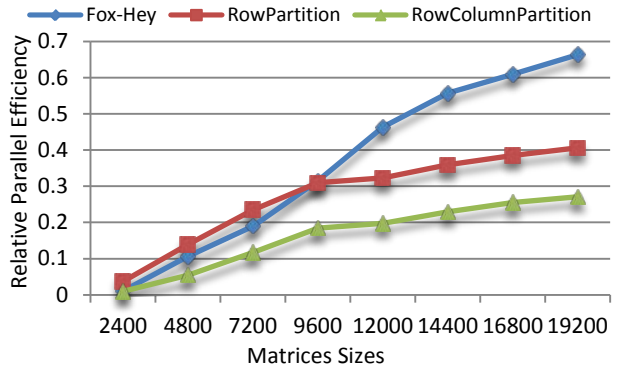

Fig. 10: Relative Parallel Efficiency for the PLINQ Version of the Three Matrix Multiplication Algorithms

3.2.4. *Compare with OpenMPI and Twister*

We compared the scalability of the Fox algorithm of DryadLINQ/PLINQ with that of the OpenMPI/Pthread and Twister/Thread. The DryadLINQ experiments were run with 24 cores per node on 16 nodes in TEMPEST. The OpenMPI and Twister experiments were run with 8 cores per node on 16 nodes in Quarry. The matrices sizes ranged from 2,400*2,400 to 31,200*31,200. As shown in Figure 11, the parallel efficiency of

the Fox algorithm of the DryadLINQ/PLINQ was smaller than that of the OpenMPI/Pthread and Twister/Thread for the small matrices sizes. The super linear speed up in Twister is due to the cache behaving better in the parallel case. The experiment results also indicate that the DryadLINQ implementation is able to scale out for large matrices.

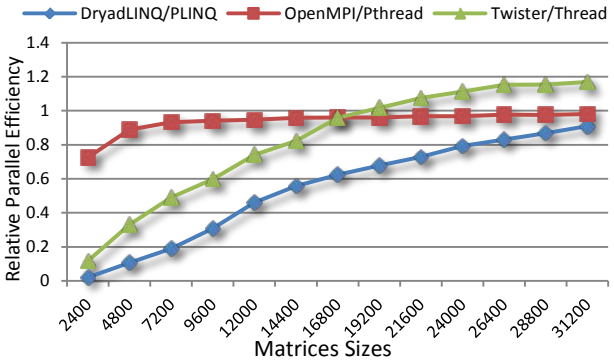

Fig. 11: Parallel Efficiency of the Fox Algorithm Using the DryadLINQ/PLINQ, OpenMPI/Pthread and Twister/Thread

3.3. Distributed Grouped Aggregation

We studied the distributed grouped aggregation in the DryadLINQ CTP using PageRank with real data. Specifically, we investigated the programming interface and performance of the three distributed grouped aggregation approaches in the DryadLINQ, which included the Hash Partition, Hierarchical Aggregation and Aggregation Tree. Further, we studied the features of the input data that affected the performance of the distributed grouped aggregation implementations.

PageRank is already a well-studied web graph ranking algorithm. It calculates the numerical value of each element of a hyperlinked set of web pages in order to reflect the probability that a random surfer will access those pages. The PageRank process can be understood as a Markov Chain, which needs recursive calculations in order to converge to the final results. An iteration of the algorithm calculates the new access probability for each web page based on the values calculated in the previous computation. The iterations will not stop until the Euclidian distance between the two subsequent rank value vectors becomes less than a predefined threshold. In this paper, we implemented the DryadLINQ PageRank using the ClueWeb09 dataset [16], which contained almost 50 million web pages.

We split the entire ClueWeb graph into 1,280 partitions, each saved as an Adjacency Matrix (AM) file. The characteristics of the input data are described below:

No of Am Files	File Size	No of Web Pages	No of Links	Ave Out-degree
1280	9.7 GB	49.5 million	1.40 billion	29.3

3.3.1. *PageRank using Three Distributed Grouped Aggregation Approaches*

PageRank is a communication intensive application that requires joining two input data streams and then performing the grouped aggregation over partial results.

First, we implemented PageRank with the hash partition approach with three main functions [17]: *Join()*, *GroupBy()*, and user-defined aggregation function. In the Join stage, we constructed the *DistributedQuery<Page>* objects that represented the web graph structure of the AM files. Then, we constructed the *DistributedQuery<Rank>* objects each of which represent a pair that contains the identifier number of a page and its current estimated rank value. After that, the program joins the pages within the ranks in order to calculate the partial rank values. Then, the *GroupBy()* operator hash partition calculated the partial rank values to some groups, where each group represented a set of partial ranks with the same source page pointing to them. At last, the partial rank values in each group were aggregated using the user-defined aggregation function.

Second, we implemented PageRank using the hierarchical aggregation approach, which has tree fixed aggregation stages: 1) the first pre-aggregation stage for each user-defined aggregation function, 2) the second pre-aggregation stage for each DryadLINQ partition and 3) the third global aggregation stage to calculate the global PageRank rank values.

The hierarchical aggregation approach may not perform well in the computation environment which is inhomogeneous in network bandwidth, CPU and memory capability due to the existence of its global synchronization stages. In this scenario, the aggregation tree approach is a better choice. It can construct a tree graph in order to guide the job manager to make the optimal aggregation operations for many of the subsets of the input tuples so as to decrease the intermediate data transformation. We implemented PageRank using the aggregation tree approach by invoking the *GroupAndAggregate()* operator in DryadLINQ CTP [10].

3.3.2. *Performance Analysis*

We evaluated the performance of the three approaches by running PageRank jobs using various sizes of input data on 17 compute nodes on TEMPEST. Figure 12 shows that the aggregation tree and hierarchical aggregation approaches outperformed the hash partition approach. In the ClueWeb dataset, the URLs are stored in alphabetical order and the web pages that belong to the same domain are likely to be saved in one AM file. Thus, the intermediate data transfer in the hash partition stage can be greatly reduced by applying the pre-aggregation to each AM file. The hierarchical aggregation approach outperforms the aggregation tree approach because it has a coarser granularity processing unit. In addition, our experiment environment for the TEMPEST cluster has a homogeneous network and CPU capability.

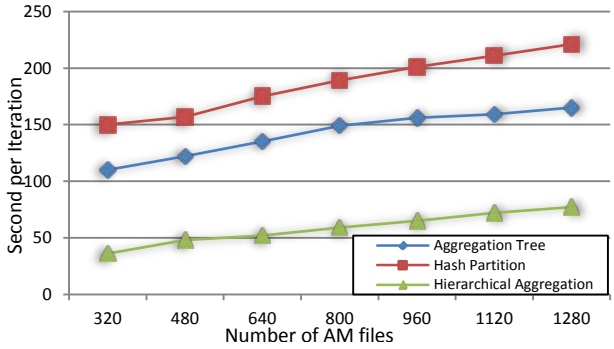

Fig. 12: PageRank Execution Time per Iteration with Three Aggregation Approaches on 17 Nodes

In general, the pre-aggregation approaches work well only when the number of output tuples is much smaller than the input tuples. The hash partition works well only when the number of output tuples is larger than the input tuples. We designed a mathematics model in order to theoretically analyze how the ratio between the

input and output tuples affected the performance of the aggregation approaches. First, we defined the data reduction proportion (DRP) [18] in order to describe the ratio as follows:

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

Table 4: Data Reduction Ratios for Different PageRank Approaches with the Clueweb09 Dataset

Input Size	Hash Aggregation	Pre-aggregation	Hierarchical Aggregation
320 files 2.3 GB	1:306	1:6.6:306	1:6.6:2.1:306
640 files 5.1 GB	1:389	1:7.9:389	1:7.9:2.3:389
1,280 files 9.7G	1:587	1:11.8:587	1:11.8:3.7:587

Further, we defined a mathematic model to describe how the DRP will affect the efficiency of different aggregation approaches. First, we assumed that the average number of tuples for each group is M (M=1/DRP) and that there are N compute nodes. Then, we assumed that the M tuples of each group are evenly distributed on the N nodes. In the hash partition approach, the M tuples with the same key are hashed into the same group on one node, which require M aggregation operations. In the pre-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 for each record of the two approaches is as follows:

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

Usually, DRP is much smaller than the number of compute nodes. Taking word count as an example, documents with millions of words may have several thousands common words. As the web graph structure obeys zipf's law, the DRP of the PageRank input data was not as small as the DRP in regard to word count. Thus, the pre-aggregation approach in PageRank may not deliver performance as well as word count [10].

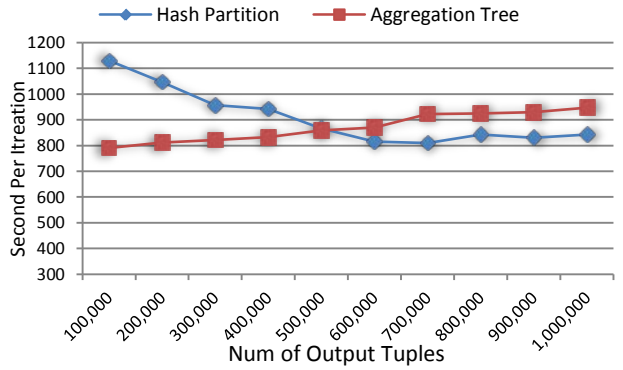

Fig. 13: Execution Time for Two Aggregation Approaches with Different DRP Values.

In order to quantitatively analysis how the DRP affected the aggregation performance, we compared the two aggregation approaches using a set of web graphs with different DRPs by fixing the number of output tuples and changing the number of input tuples. Figure 13 shows the time per iteration of the PageRank jobs for serial datasets whose output tuples ranged from 100,000 to 1000,000 while input tuples were fixed at 4.3 billion. As shown in Figure 13, different grouped aggregation approaches fit well with different DRP range of input data.

3.3.3. *Compare with Other Runtimes*

We compared the performance of the distributed grouped aggregation of DryadLINQ with OpenMPI [19], Twister [20], Hadoop, and Haloop [21]. We implemented PageRank using these five runtimes for the ClueWeb09 dataset with the Power method [22]. The DryadLINQ experiments were run with 24 cores per node on 16 nodes in TEMPEST. The MPI, Twister, Hadoop, and Haloop experiments were run with 8 cores per node on 16 nodes in Quarry.

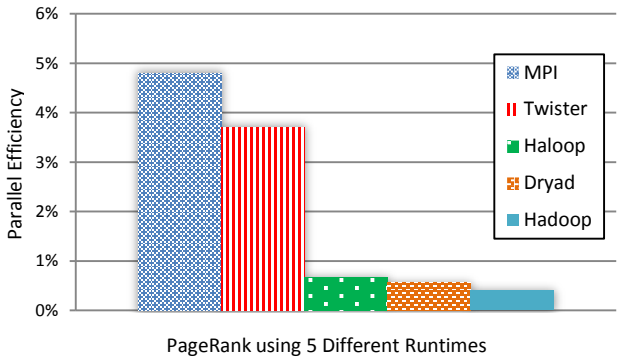

Fig 14 Parallel Efficiency of Five PageRank Implementation

As shown in Figure 14, the parallel efficiency of the PageRank jobs was noticeably smaller than 1%. The first reason is that PageRank is a communication intensive application, and the computation does not use a large proportion of the overall PageRank job turnaround time. Second, using multi-core technology does not help to increase parallel efficiency; instead it decreases overall parallel efficiency. The MPI, Twister and Haloop implementations outperformed the DryadLINQ implementations, because they could cache loop-invariable data or static data in the memory in multiple iterations. Dryad and Hadoop were slower than the other approaches, as their intermediate results were transferred via distributed file systems.

4. RELATED WORK

4.1. *Pleasingly Parallel Application*

We have shown that the DryadLINQ developers could easily tune task granularity in order to solve the workload balance issue. In the batch job scheduling systems, such as PBS, the programmers have to manually group/un-group or split/combine input data in order to control the task granularity. Hadoop provides an interface that allows developers to control task granularity by defining the size of the input records in the HDFS. This approach is an improvement, but still requires developers to understand the logic format of the input record in HDFS. DryadLINQ provides a simplified data model and interface for this issue based on the existing .NET platform.

4.2. *Hybrid Parallel Programming*

The hybrid parallel programming combines the inter node distributed memory parallelization with the intra node shared memory parallelization. MPI/OpenMP/Threading is the hybrid programming model that is utilized in high performance

computing. Paper [23] discusses the hybrid parallel programming paradigm using MPI.NET, TPL and CCR (Concurrency and Coordination Runtime) on a Windows HPC server. The results of the experiments show that the efficiency of the hybrid parallel programming model has to do with the task granularity, while the parallel overhead is mainly caused by synchronization and communication.

Twister and Hadoop can also make use of multiple core systems by launching multiple task daemons on each compute node. In general, the number of task daemons is equal to that of the cores on each compute node. The advantage of these systems is the unified programming and scheduling model can be used to leverage multi-core parallelism.

4.3. Distributed Grouped Aggreagtion

MapReduce and SQL database are two programming models that can perform grouped aggregation. MapReduce has been used to process a wide range of flat distributed data. However, MapReduce is not efficient when processing relational operations, such as Join, which have multiple inhomogeneous input data streams. The SQL queries are able to process the relational operations of multiple inhomogeneous input data streams; however, operations in full-feature SQL database have big overhead that prevents the application from processing large scale input data.

DryadLINQ lies between SQL and MapReduce, and addresses some of the limitations found in SQL and MapReduce. DryadLINQ provides developers with SQL-like queries by which to process efficient aggregation for single input data streams and multiple inhomogeneous input streams, but has reduced its overhead to less than SQL by eliminating some of the functionality of the database (transactions, data lockers, etc.). Further, Dryad can build an aggregation tree (some databases also provide this type of optimization) so as to decrease the data transformation in the hash partitioning stage.

5. DISCUSSION AND CONCLUSION

In this paper, we discussed the three design patterns in the DryadLINQ CTP to be used in scientific applications. The Smith Waterman – Gotoh algorithm (SWG) is a pleasingly parallel application which consists of Map and Reduce steps. We implement it using the ApplyPerPartition operator, which can be considered as distributed version of "Apply" in SQL. In the Matrix Multiplication, we explored a hybrid parallel programming model that combines inter-node distributed memory with intra node shared memory parallelization. The hybrid model is implemented by porting multicore technologies such as PLINQ and TPL into user-defined functions within the DryadLINQ queries. PageRank is a communication intensive application that requires joining two input data streams and then performing the grouped aggregation over partial results. We implemented PageRank with the three distributed grouped aggregation approaches. To our knowledge, these patterns have covered a wide range of distributed scientific applications.

Further, we discussed the issues that affected the performance of the applications implemented within these DryadLINQ programming models. By studying the experiments results, the following results were evident: 1) DryadLINQ CTP provides a unified data model and flexible programming interface for developers, which can be used to solve the workload balance issue for pleasingly parallel applications; 2) porting multi-core technologies, such as PLINQ and TPL to DryadLINQ tasks can increase the system utilization for large input datasets; and 3) the choice of distributed grouped aggregation approaches with DryadLINQ CTP has a substantial impact on the performance of data aggregation/reduction applications.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Jaliya Ekanayake, Thilina Gunarathne, et al. (2010). *Applicability of DryadLINQ to Scientific Applications*, Community Grids Laboratory, Indiana University.

[2] Dean, J. and S. Ghemawat (2004). "MapReduce: Simplified Data Processing on Large Clusters." Sixth Symposium on Operating Systems Design and Implementation: 137-150.

[3] Apache (2010). "Hadoop MapReduce." Retrieved November 6, 2010, from http://hadoop.apache.org/mapreduce/docs/current/index.html.

[4] Isard, M., M. Budiu, et al. (2007). Dryad: distributed data-parallel programs from sequential building blocks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. Lisbon, Portugal, ACM: 59-72.

[5] Yu, Y., M. Isard, et al. (2008). DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. Symposium on Operating System Design and Implementation (OSDI). San Diego, CA.

[6] Introduction to Dryad, DSC and DryadLINQ. (2010). http://connect.micorosft.com/HPC

[7] Ekanayake, J., A. S. Balkir, et al. (2009). DryadLINQ for Scientific Analyses. Fifth IEEE International Conference on eScience: 2009. Oxford, IEEE.

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

[9] Malewicz, G., M. H. Austern, et al. (2010). Pregel: A System for Large-Scale Graph Processing. Proceedings of the 2010 international conference on Management of data, Indianapolis, Indiana.

[10] Yu, Y., P. K. Gunda, et al. (2009). Distributed aggregation for data-parallel computing: interfaces and implementations. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. Big Sky, Montana, USA, ACM: 247-260.

[11] Moretti, C., H. Bui, et al. (2009). "All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids." IEEE Transactions on Parallel and Distributed Systems **21**: 21-36.

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

[13] Li, H., Y. Huashan, et al. (2008). A lightweight execution framework for massive independent tasks. Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Austin, Texas.

[14] G. Fox, A. Hey, and Otto, S (1987). Matrix Algorithms on the Hypercube I: Matrix Multiplication, Parallel Computing, **4**:17-31

[15] Jaliya Ekanayake (2009). Architecture and Performance of Runtime Environments for Data Intensive Scalable Computing. Supercomputing 2009 (SC09). D. Showcase. Portland, Oregon.

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

[17] 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.

[18] S. Helmer, T. Neumann, G. Moerkotte (2003). Estimating the Output Cardinality of partial Preaggregation with a Measure of

Clusteredness. Proceeding of the 29th VLDB Conference. Berlin, Germany.

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

[20] J.Ekanayake, H.Li, et al. (2010). Twister: A Runtime for iterative MapReduce. Proceedings of the First International Workshop on MapReduce and its Applications of ACM HPDC 2010 conference June 20-25, 2010. Chicago, Illinois, ACM.

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

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

[23] Judy Qiu, Scott Beason, et al. (2010). Performance of Windows Multicore Systems on Threading and MPI. Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, IEEE Computer Society**:** 814-819.