DryadLINQ Programming Models for Scientific Analysis

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

School of Informatics and Computing, Pervasive Technology Institute

Indiana University Bloomington

Abstract

The programming model is a critical factor in determining runtimes. One reason why Cloud is more popular than Grid is that Cloud has several practical programming models. 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. We have conducted extensive experiments on DryadLINQ [1, 2], DryadLINQ CTP [3, 4], and LINQ to HPC [7] in order to identify the classes of scientific applications that fit well. This paper summarized our experiences and abstracts four DryadLINQ programming models that are applicable to a large class of scientific applications, which include pleasingly parallel, hybrid parallel, distributed grouped aggregation, and iterative MapReduce programming models.

Keyswords: Dryad, DryadLINQ, MapReduce, Programming Models, Multi-Core, Aggregation, Iterative MapReduce

1. Introduction

We are in the era of Big Data. 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 an 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 simplify the usage by leaving the complexity of parallel programming such as scheduling, fault tolerance, load balancing and communications to underlying runtime systems. 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, MapReduce has several limitations. First, 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 its Join step spawns a 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. Second, the original MapReduce does not support the iterative applications properly. Simply repetitive application of MapReduce for iterative applications will lead to significant overhead in terms of creating Map and Reduce tasks and reading/writing input/output data. For example, MPI PageRank outperforms Hadoop PageRank by a factor of 10, as Hadoop need materializes intermediate data to local file system and output data to HDFS in each iteration.

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, and is able to address some of the limitations that exist in the MapReduce systems. DryadLINQ is the high-level programming language and compiler for Dryad. The DryadLINQ compiler can automatically translate the Language-Integrated Query (LINQ) programs written by .NET language into distributed, optimized computation steps that run on top of the Dryad cluster. Thus, developers do not need to know much about Dryad or even about parallel and distributed computing when writing DryadLINQ programs. 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 four important DryadLINQ programming models that can be applied to a large class of scientific applications. These models are: pleasingly parallel, hybrid parallel, distributed grouped aggregation, and the iterative MapReduce programming model. It is important to note that in this paper, “DryadLINQ CTP” refers to the DryadLINQ community technical preview released in December 2010; "LINQ to HPC" refers to the newly released LINQ to HPC Beta 2 released in July 2011; “Dryad/DryadLINQ (2009)” refers to the version released in November 2009; and “Dryad/DryadLINQ” refers to all Dryad/DryadLINQ versions.

1. Related Work

Hadoop’s Pig Latin [35] is a natural analogue to DryadLINQ. DryadLINQ performs better than Pig when processing relational queries and iterative MapReduce tasks. One main reason for this performance difference is the underlying DAG execution model, which is more expressive and flexible than that of Hadoop. Another reason for the performance difference is that Pig translates relational queries into a set of MapReduce circles for execution without much exploring in regard to the optimization of a queries execution plan. YSmart [36] is another SQL-to-MapReduce translator that outperforms Pig by exploring query correlations.

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 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.

MapReduce and SQL for databases are two programming models used to perform distributed grouped aggregation. Some systems, such as Hadoop and Oracle, can support the pre-aggregation optimization for user-defined aggregation functions. However, their interfaces for implementing pre-aggregation are not as convenient as those in DryadLINQ [7]. In addition, a full-featured SQL database has extra overhead and constraints that prevent it from performing distributed grouped aggregation for large-scale input data.

Open source Java Twister [44, 45], Twister4Azure [46, 47] and Haloop are Iterative MapReduce frameworks. Twister interpolates between MPI and MapReduce and, when suitably configured, can mimic their characteristics. It can be positioned as a programming model that has the performance of MPI and the fault tolerance and dynamic flexibility of the original MapReduce. Haloop is a modified version of the Hadoop MapReduce framework and dramatically improves the iterative MapReduce by making the tasks scheduler loop-aware and adding various caching mechanisms.

1. DryadLINQ Programming Models
   1. DryadLINQ

Dryad, DryadLINQ and DSC [6] are a set of technologies that support the processing of data intensive applications in the Windows platform. 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 [5] is the high-level, declarative, data flow programming language. The DryadLINQ programming model is based upon the Language-Integrated Query (LINQ) programming model and it takes advantage of the standard query operators as 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. Thus, the DryadLINQ programming model is more expressive and flexible than the MapReduce programming model when processing many semi-structure and un-structure data applications. In this paper, we will investigate the usability and performance of the four DryadLINQ programming models illustrated in Figure 1: a) pleasingly parallel, b) hybrid parallel, c) distributed grouped aggregation, and d) iterative MapReduce.

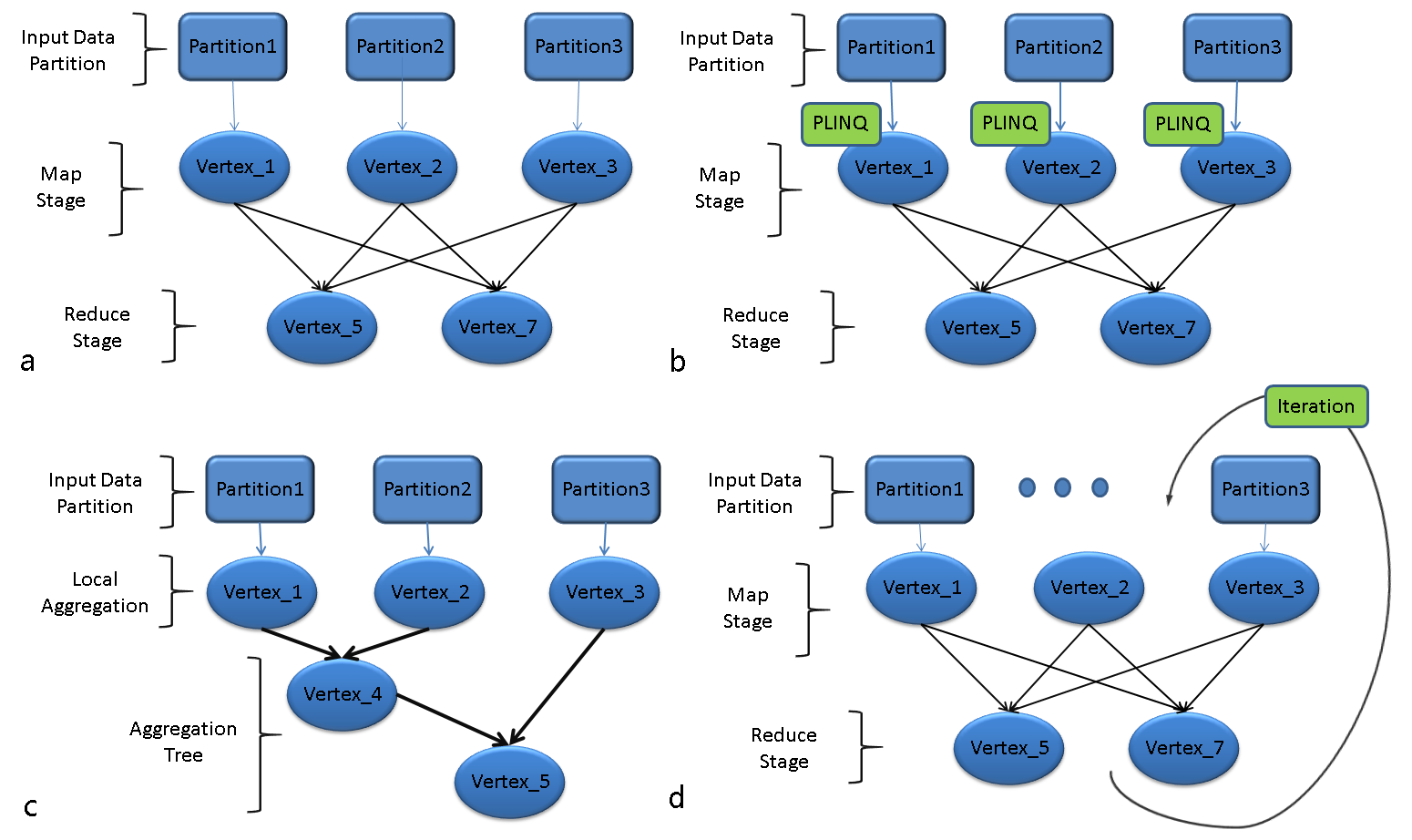


Figure 1: Four programming models for scientific applications in DryadLINQ CTP:a) pleasingly parallel, b) hybrid parallel, c) distributed grouped aggregation and d) iterative MapReduce.

* 1. DryadLINQ programming models
     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 clusters. These DryadLINQ data objects are represented as DistributedQuery<T> or DistributedData<T> objects to which the LINQ operators can apply. DryadLINQ applications can create DistributedData<T> objects from existing data stored on the DSC or convert it from the IEnumerable<T> objects using the 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

We’ve implemented a wide range of pleasingly parallel applications 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 alignment using the SWG application.

* + 1. 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 the 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. Another 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. The AsParallel() operator within the code is inherited from LINQ query, which indicates the runtime to run the SPMD program in parallel in each node.

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 [8], MDS interpolation [8] and Matrix-Matrix Multiplication [6].

* + 1. Distributed Grouped Aggregation Programming Model

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 aggregates 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. One direct approach is to use the hash partition operator in order to redistribute the records to the compute nodes so that identical records are stored on the same node. This approach then aggregates the records of each group on each node.

The implementation of the hash partition is simple, but caused a large increase in network traffic when the number of input records is large. A common way to optimize this approach is to apply pre-aggregation, which aggregates some of the subsets of the input records and then hash partitions the aggregated partial results across a cluster based on their key. This approach is better than the direct hash partition approach because the number of records transferred across the cluster becomes much smaller after the pre-aggregation operation. Two additional ways exist by which to implement the pre-aggregation: 1) a 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 used to guide a job manager in regard to performing asynchronous pre-aggregation for many of the input records subsets. DryadLINQ developers can implement pre-aggregation via a special GroupAndAggregate operator.

Var inputTuples = wordList.Select(word => new Tuple<string, int>(word, 1));

//Construct input tuples using Select

Var wordCount = inputTuples.GroupAndAggregate(t => t.Item1, g => new Tuple<string, int>

(g.Key, g.Sum(x => x.Item2)));

//Perform group and aggregate with optimized pre-aggregation execution plan

* + 1. Iterative MapReduce Programming Model

Intel’s RMS (recognition, mining, and synthesis) taxonomy [43] offers a way by which to describe a class of emerging applications. The technology underlying these applications is likely to have broad applicability, ranging across computer vision, rendering, physical simulation, (financial) analysis, and data mining. Common computing kernels exist at the core of these applications, which require iterative solvers and basic matrix primitives. These observations suggest that Iterative MapReduce will become an important runtime to a spectrum of scientific, industrial, and societal applications and the kernel framework for large-scale data processing.

Dryad/DryadLINQ can pipeline the execution of iterative MapReduce jobs by using deferred evaluation, and asynchronous vertices processing technologies. In addition, Dryad is able to maintain the data locality for processing vertices during different computation stages of DryadLINQ jobs [4]. In addition, it is able to reuse the static data stored in its memory for multiple iterations [4]. Below is the pseudo code for the iterative Kmeans computation using DryadLINQ. The output centers in each iteration are used as input centers in subsequent iterations. The static data, vectors, are never re-transferred across the network or re-loaded from the disk to memory after the first iteration.

IQueryable<Vector> KMeansStep(IQueryable<Vector> vectors, IQueryable<Vector> centers){

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

***centers*** = vectors.GroupBy(vector => NearestCenter(vector, ***centers***))

.Select(g => g.Aggregate((x,y) =>x+y) / g.Count());

}

1. Scientific Applications

We implemented SWG, Matrix-Matrix Multiplication, PageRank, and Kmeans using the DryadLINQ CTP and LINQ to HPC. We evaluated their performances on two Windows HPC clusters. The hardware resources used in this paper are as follows

Table 1. 32 nodes homogeneous HPC cluster TEMPEST

|  |  |  |
| --- | --- | --- |
| Compute Node | TMPEST | TEMPEST-CNXX |
| CPU | Intel E7450 | Intel E7450 |
| Cores | 24 | 24 |
| Memory | 24 GB | 50 GB |
| Memory/Core | 1 GB | 2 GB |

Table 2. 7 nodes inhomogeneous HPC cluster STORM

|  |  |  |  |
| --- | --- | --- | --- |
| Compute Node | 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

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

* 1. SWG Pleasingly Parallel Application

The Alu clustering problem [11] [12] is one of the most challenging problems faced when sequencing the 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 the Alu repeats can be deduced solely from the one million human Alu elements. Notably, Alu clustering can be viewed as a classic 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).

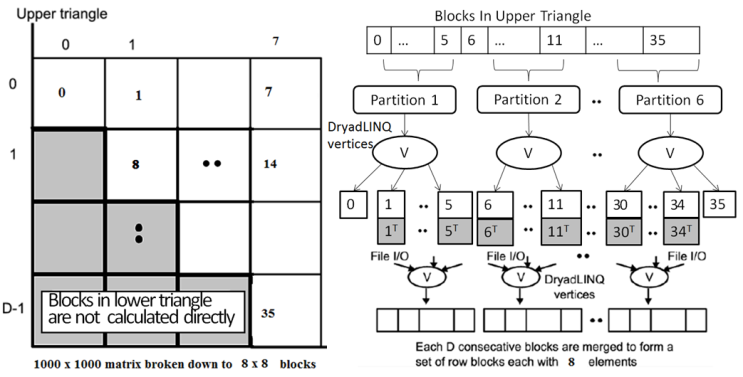


Figure 2: DryadLINQ implementation of SWG application

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 2. 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 schemes will cause different task granularities. The DryadLINQ developers can control task granularity by simply specifying the number of partitions using the RangePartition() operator.

* + 1. Workload Balance for Inhomogeneous Tasks

Low system utilization caused by workload imbalance issue is one of main sources of performance degradation. Workload imbalance issue occurs 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 use when tuning task granularity. The following experiments will study the workload balance issue in the DryadLINQ SWG application.

The SWG application 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 by which to solve 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 and 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.

* + 1. 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 in order to know the resource requirements of each job and the availability of the hardware resources. Another solution is to split the job into many finer granularity tasks and dispatch available tasks to idle computational resources.

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). The first SW-G 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 the finer partition granularity can deliver a better load balance on the inhomogeneous computational nodes. However, it also shows that the task scheduling cost increased as the number of partitions increased.

Figure 3: Performance Comparison for Skewed Figure 4: Relative Parallel Efficiency of Hadoop and

Distributed Data with Different Task Granularity. DryadLINQ with Different Task Granularity

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

* + 1. Compare with Hadoop

As shown in Figures 3 and 4, task granularity is important in regard to the workload balance issue in DryadLINQ. Further, we compared the task granularity issue in DryadLINQ with that in Hadoop. The DryadLINQ/PLINQ SW-G experiments were run using 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 4, 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.

* 1. Hybrid Parallel Programming Model

We explored the hybrid parallel programming model in DryadLINQ CTP by running the Matrix Multiplication with three algorithms and three multi-core technologies. The analysis and discussion were presented in detail in paper [7]. This paper has a more sophisticated performance analysis of the Fox algorithm using LINQ to HPC and Thread Pool on tree network Windows HPC cluster. The LINQ to HPC invokes inter-node parallelism while Thread Pool supports inner-node parallelism. Our experiments show that it is imperative to evaluate hybrid parallel programming paradigms that may potentially scale up to tens or hundreds of multicore processors.

* + 1. Fox Matrix-Matrix Multiplication Algorithms

The matrix-matrix multiplication is a fundamental kernel [12] that can be used to achieve high efficiency in both theory and practice. The computation can be partitioned into subtasks, which makes it an ideal candidate application in hybrid parallel programming studies using Dryad/DryadLINQ. However, there is no one optimal solution fits all scenarios. Different trade-offs of partition granularity largely correspond to computation and communication costs and are affected by memory/cache usage and network bandwidth/latency. We investigated the performance of the Fox/Thread Pool implementation using different numbers of nodes and cores per node. The purpose is to create a timing model for the Fox algorithm implemented with LINQ to HPC on a tree network Windows HPC cluster. The difficulty exists when attempting to model the communication/synchronization overhead when using LINQ to HPC on Windows HPC cluster.

(Eq. 1)

The matrix-matrix multiplication is defined as A \* B = C (Eq. 1). Figure 6 shows the work flow of the Fox algorithm on a mesh of 2X2 compute nodes. Matrices A and B are split along both rows and columns to construct a matching 2X2 block data mesh. During each step of computation, each process holds a current block of Matrix A by broadcasting and a current block of Matrix B by shifting upwards and then computing a block of Matrix C. We assume that the input sub-matrices A and B already reside in compute nodes that have been decomposed in the two-dimensional fashion, as shown in Figure 6. In addition, we also assume that output sub-matrices C will end up decomposed in the same way. Thus, our timings did not consider overhead of loading of sub-matrices A, B and C.



Figure 6: Work Flow of the Fox Algorithm on a Mesh of 2\*2 Nodes

* + 1. Timing model for the Fox algorithm with LINQ to HPC on a tree network Cluster

This section creates a theoretical analysis of the Fox algorithm with LINQ to HPC. Assume the M\*M matrix multiplication jobs are partitioned and run on a mesh of √N\*√N nodes. The size of the sub-matrices in each node is m\*m, where m=M/√N. The “broadcast-multiply-roll” cycle of the algorithm is repeated √N times. For each such cycle, the time taken to broadcast one sub-matrix A is:

We assume that the broadcast is done as a linear pipeline in our LINQ to HPC implementation. Tstartup is the start-up time of the pipeline per step of the pipe. (Tio+Tcomm) is the cost to transfer one matrix element over cluster. We take Tio into account when modelling communication overhead is because Dryad usually uses NTFS to transfer intermediate data over HPC cluster. Our related experiments results show that the IO overhead makes up 40% of the overall data transportation overhead. Therefore, we must consider the disk IO when modelling the communication overhead. As the process to “roll” sub-matrix B can be done in parallel in a tree network cluster as long as the aggregated requirement of network bandwidth is satisfied by the switch, its overhead is:

Finally, the time taken to compute the sub-matrix product (including the multiplication and addition) is:

2\*

The total computation time of the Fox algorithm is:

(Eq. 2)

(Eq. 3)

We define parallel efficiency as the speedup per processing node, shown in equation 4. The measured average is less than 0.5 second. As shown in equation 3, parallel efficiency can be deduced as equation 4 for large matrices sizes.

(Eq. 4)

* + 1. Performance analysis on Tempest

Figure 7: parallel overhead vs. 1/Sqrt(n) on 4x4 Figure 8: parallel overhead vs. 1/Sqrt(n) on 3x3,4x4,5x5 nodes with 1 core per node. (n is grain size, number nodes with1 core per node. showing universal behaviour

of matrix elements per node)

Figure 9: the same as Fig 8, but with timings using Figure 10: Perpormance of Intel MKL using 1,8,16,24

OpenMPI on Linux cluster. cores per node. Showing parallel overhead

Figure 11: the same as Fig 2, but with timings using Figure 12: parallel overhead vs. 1/Sqrt(n) using 16nodes

different number of cores per node 1core per node and 16 cores on 1 node. Compare distributed memory parallel with shared memory parallel in terms of parallel overhead.

In order to measure the parallel overhead of our hybrid parallel programing model in LINQ to HPC, we performed a set of experiments using different numbers of compute nodes and cores per node for various matrices sizes. Figure 7 shows the parallel overhead of the Fox algorithm when using different numbers of nodes. The parallel overhead is larger when using more compute nodes due to the involved communication overhead. As shown in Figure 8, the parallel overhead is also larger when using more cores per node with a fixed number of nodes. This overhead is due to the synchronization costs and the memory bandwidth competition of the Threads Pool program. In addition, we implemented the blocked matrix multiply algorithm in order to avoid cache confliction issues for the large matrices. Figure 7 and figure 8 show the dramatic decrease in parallel overhead for the large matrices sizes that occurs due to the computation overhead that become dominant factor in the overall cost.

In order to verify that our theoretical analysis of the Fox algorithm with LINQ to HPC was sound, we ran matrix multiplication jobs in order to measure the parallel overhead using various matrices sizes and numbers of computer nodes. Figure 9 plots 1/e-1 versus 1/Sqrt(n) with e calculated with Equation 4 and a fitting function used to time the sequential program. “n” is the “grain size”, the number of matrix elements per node. The linear behaviour for the small 1/Sqrt(n) (large matrices) shows that the function form of Equation 4 is correct. In addition, it shows that the faster the Tflops (more cores used) is, the larger the linear rising term. Figure 10 also shows parallel overhead versus 1/Sqrt(n) but uses different numbers of compute nodes with 16 cores per node. The results indicate that the parallel overhead coefficient, (Tio+Tcomm)/Tflops, is independent of the number of compute nodes and dependent only upon the “grain size”. Figure 8 and figure 9 indicate that the high performance system that uses a faster Tflops and Tcomm+Tio is sensitive to the especially when processing small matrices sizes.

* 1. PageRank 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:

Table 4. Characteristics of ClueWeb09 input data

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

* + 1. PageRank using Three Distributed Grouped Aggregation Approaches

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

First, we implemented PageRank using the hash partition approach that contained three main functions [17]: Join(), GroupBy(), and a 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 represented a pair that contained the identifier number of a page and its current estimated rank value. Next, 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. Finally, 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 as it is inhomogeneous in regard to 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]

* + 1. 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 11 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.

Figure 13 provides CPU utilization (left) and network utilization (right) information of the three aggregation approaches obtained from the HPC cluster manager. It is apparent that hierarchical aggregation requires much less network bandwidth than the other two approaches. In fact, it pays off the less network utilization in terms of more CPU overhead that used to reduce the size of intermediate data.

Figure 11: time to compute PageRank per iteration Figure 12: time per iteration with two aggregation

with three aggregation approaches using ClueWeb09 approaches using different numbers of output tuples

data on 17 nodes in Tempest (from 100,000 to 1,000,000) while number of input

tuples fixed as 4.3 billion. Shows DRP effects.

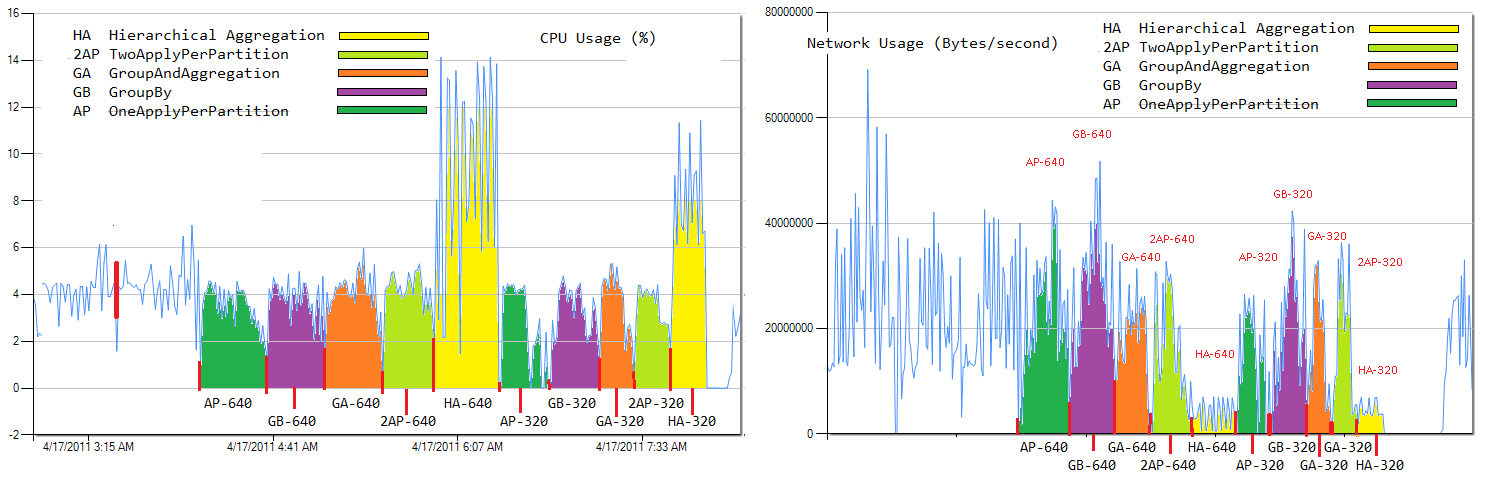


Figure 13: Trade-off between CPU (left) and network utilization (right) with different aggregation strategies

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:

(1)

Table 5. DRP of different number of AM files of three aggregation approaches

|  |  |  |  |
| --- | --- | --- | --- |
| Input Size | Naive 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 9.7 GB | 1:587 | 1:11.8:587 | 1:11.8:3.7:587 |

Further, we defined a mathematic model used 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 needs N more aggregation operations. Thus, the total number of aggregation operations for the M tuples is (M/N)\*N+N. The average number of aggregation operations for each record of the two approaches is as follows:

(2)

Usually, DRP is much smaller than the number of compute nodes. Using 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]

In order to verify the impact of DRP on the different aggregation approaches, we ran PageRank using web graphs of the different DRP values. As shown in Figure 12,when the number of out tuples is smaller than 500k (small DRP), the aggregation tree approach provides the better performance. When the number bigger than 500k (big DRP), the harsh partition provides the better performance.

* 1. Kmeans Iterative MapReduce

We studied the iterative MapReduce in DryadLINQ using the Kmeans clustering. It is a typical iterative algorithm that aims to partition N observations into K clusters where each observation belongs to the cluster with the nearest centroid. The process of computing the K centroids require that multiple iterations converge.

* + 1. Kmeans with iterative MapReduce

The Kmeans clustering has been implemented using DryadLINQ and compared its performance with other implementations in paper [7]. In this paper, we implemented Kmeans with LINQ to HPC, and make further experiments to study its performance for iterative computations. We sketch the pseudo code for Kmeans in section 3.2.4. The two input streams IQeruyable<Vector> vectors and IQueryable<Vector> centers were distributed over the cluster via DSC service in advance. We form one group per center that calculated by NearestCenter().The vectors within each group are summed up using Aggregate(). The aggregated values are divided by the size of that group to calculate new centers, which will be used as input centers in next iteration.

* + 1. Performance analysis

Figure 14 and figure 15 show the comparison of the performance of the different implementation of the Kmeans. Figure 15, it used a fixed number of compute nodes and input data points, while the number of iterations increased from 1 to 16. Both the implementation shows the linear performances, which indicate the constant overhead per iteration of the two implementations. Figure 15 is the relative speedup of Kmeans jobs using fixed iterations and input data points when the number of compute nodes increased from 1 to 16. The relative speedup of LINQ to HPC Kmeans was less than 4 when using 16 nodes, which was smaller than that of OpenMPI. DryadLINQ can chain the execution of multiply LINQ queries by using deferred evaluation technology. In addition, the input and output streams can be pipelined between vertices in consequent iterations. The low relative speedup of the LINQ to HPC Kmeans occurs because of the high overhead of the core LINQ to HPC operations when compared to OpenMPI.

Figure 14: time per iteration with different size of Figure 15: running time with different iterations input data with 8 nodes with 1 core per node with 16 nodes with 1 core per node

Figure 16: relative speed-up with different nodes Figure 17: running time with fixed workload

with 1 core per node for 10 iterations with different nodes with 1 core per node

1. Discussion and Conclusion

In this paper, we studied four typical programming models (pleasingly parallel, hybrid parallel, distributed grouped aggregation, and iterative MapReduce) for scientific applications using DryadLINQ, DryadLINQ CTP and LINQ to HPC. The Smith Waterman – Gotoh algorithm (SWG) is a pleasingly parallel application that consists of Map and Reduce steps. We implement this application using the ApplyPerPartition operator, which can be considered to be a distributed version of “Apply” in SQL. We studied the hybrid parallel programming model used in paper []. In this paper, we created a timing model for the LINQ to HPC Fox implementation and studied the main factors that effected the parallel overhead of our implementation. PageRank is a communication intensive application that requires joining two input data streams in order to perform the distributed grouped aggregation over partial results. We implemented PageRank using the three distributed grouped aggregation approaches with interfaces provided by DryadLINQ CTP. Finally, we investigated the iterative MapReduce using Kmeans cluster application. To our knowledge, these patterns 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 results of the experiments, we were able to determine that: 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.

The industry and community are constantly and significantly lowering the cost of high-end computation technologies that previously were only adopted by giant IT companies such as Microsoft, Google, and IBM. Thus, there is the crisis for closed, private computation technologies. In November 2011, Microsoft discontinued Dryad/DryadLINQ and they announced an end-to-end roadmap for Big Data that embraces Apache HadoopTM. However, as a LINQ provider, the DryadLINQ was the most successful distributed data parallel runtime in the Windows platform. Several features within the DryadLINQ have had a profound impact on the design and implementation of data flow programming languages in the Big Data area.

Acknowledgements

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

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. Hui Li, Yang Ruan, Yuduo Zhou, Judy Qiu and Geoffrey Fox, Design Patterns for Scientific Applications in DryadLINQ CTP, to appear in Proceedings of The Second International Workshop on Data Intensive Computing in the Clouds ( DataCloud-2) 2011, The International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), Seattle, WA, November 12-18, 2011
7. Introduction to Dryad, DSC and DryadLINQ. (2010). http://connect.micorosft.com/HPC
8. Ekanayake, J., A. S. Balkir, et al. (2009). DryadLINQ for Scientific Analyses. Fifth IEEE International Conference on eScience: 2009. Oxford, IEEE.
9. 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).
10. 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.
11. 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.
12. 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.
13. Batzer MA and Deininger PL (2002). "Alu repeats and human genomic diversity." Nature Reviews Genetics **3**(5): 370-379.
14. 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.
15. G. Fox, A. Hey, and Otto, S (1987). Matrix Algorithms on the Hypercube I: Matrix Multiplication, Parallel Computing, **4**:17-31
16. Jaliya Ekanayake (2009). Architecture and Performance of Runtime Environments for Data Intensive Scalable Computing. Supercomputing 2009 (SC09). D. Showcase. Portland, Oregon.
17. ClueWeb09: http://boston.lti.cs.cmu.edu/Data/clueweb09/
18. 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.
19. 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.
20. OpenMPI http://www.open-mpi.org/
21. 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.
22. 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.