# Design Pattern for Typical Scientific Applications in DryadLINQ CTP

School of Informatics and Computing, Pervasive Technology Institute

Indiana University Bloomington

*Abstract*-The design and implementation of higher level language interfaces are becoming increasing important for data intensive computation runtimes. DryadLINQ is a system and a set of language extensions that enable the programmers develop application that process large scale distributed data, and it has been successfully used to implement wide range of applications. Recently, Microsoft released the DryadLINQ CTP (LINQ to HPC) which contains new features and interfaces to achieve better performances for applications and usability for developers. In this paper, we present three design patterns in DryadLINQ CTP which can be used to accelerate a large class of scientific applications, exemplified by SW-G, Matrix-Matrix Multiplication, and PageRank applications with large scale real data.

# I. Introduction

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

However, its fixed and flat processing paradigm does not directly support relational operations that have multiple related inhomogeneous input data stream. This limitation causes the difficulties and inefficiency when using Map-Reduce to simulate relation operations like Join which is very common in database. For example, the classic implementation of PageRank with MapReduce is just found to be very in-efficient because the simulating of Join with MapReduce costs lots of network traffic during the computation. And the effort of optimizing MapReduce PageRank requires developers have sophisticated knowledge on web graph structure.

Dryad [2] is a distributed runtime designed to execute parallel applications on Windows clusters. Dryad lies between MapReduce runtimes and database, and it can address some of limitations of MapReduce systems. DryadLINQ [3] is a library that translates LINQ programs written by .NET language into distributed computations run on top of Dryad system. The developers can apply LINQ relational operators to DryadLINQ collections. Besides, Dryad/DryadLINQ can optimize the execution of those operations without any interruption from developers. For example, implementing PageRank with GroupAndAggregate can achieve better performance than that implementing with GroupBy as it leverages the partial aggregation optimization strategy. Developers can make use of this strategy by only making few changes to the GroupBy version.

In this paper, we explore the design patterns for three typical scientific applications in the DryadLINQ CTP released in December 2010. The contributions of this page are following:

1. We study the task granularity that improve LINQ’s support for coarse-grain parallelization with DryadLINQ CTP data model and interface
2. We demonstrated a hybrid parallel programming model not only utilizes parallelism in multiple nodes but also in multiple cores within one node.
3. We evaluated different distributed aggregation strategies in DryadLINQ CTP and compare their performance with other runtimes such as MPI, Hadoop, and Twister.

The structure of this paper is as follows: section 2 illustrates the basic programming model and several issues in DryadLINQ. Section 3 describes implementation of three typical scientific applications for those programming model in DryadLINQ which include SW-G, Matrix-Matrix Multiplication and PageRank. We will study the performance of the three applications with large scale real data, and study several optimization strategies for those applications. Section 4 is the conclusion and discussion.

## II. DryadLINQ Programming Model

Dryad, DSC [5] and DryadLINQ are set of technologies support data-intensive computing applications that can run on Windows cluster. Dryad is the general-purpose distributed execution engine for data-parallel applications. DSC is the Distributed Storage Catalog that works with NTFS to provide the data management functionality for Dryad and DryadLINQ. DryadLINQ is the query-based programming model for Dryad and DSC. The software stack of Dryad is shown in Fig 1.

 

Fig 1. Software Stack for DryadLINQ CTP

### 2.1 Pleasingly Parallel Programming Model:

DryadLINQ supports a unified data model and programming language for relational queries and user-defined functions [3]. The input data collections are represented as DistributedQuery<T> and DistributedData<T> objects. Then, these objects can be split into several partitions with interface like AsDistributed(), AsDistributedFromPartitions(), and RangeParititon(). Then, these partitions are distributed to remote compute nodes and get executed there by invoking the user defined function within ApplyPerPartition() or Select() operators. For example, the pseudo code for SW-G application in section 3.1 is as follows:

DistributedQuery<OutputInfo>outputResults = inputDataSet.AsDistributedFromPartitions()

.ApplyPerPartition(inputs =>

PerformAlignments(inputs));

### 2.2 Hybrid Parallel Programming Model

Dryad is supposed to process coarse-granularity tasks for large scale distributed data. To leverage the computation power of multi-core system, one approach is to perform the computation to PLINQ objects. The DryadLINQ provider can automatically transfer PLINQ query to parallel computation. The other approach is to manually port multi-core technologies in .NET like TPL, thread pool into DryadLINQ task. In above hybrid parallel programming model, the Dryad runtime handle the parallelism in node level and the PLINQ, TPL, and thread pool technologies deal with the parallelism in multi-core level.

The performance of application implemented in hybrid model is affected not only by parallel algorithm and programming model in node level, but also by factors in core level like cache and memory bandwidth [16]. In the paper, we verified the hybrid parallel programming model for matrix multiplication with different combination matrix multiplication algorithms and multi-core technologies.

### 2.3 Partial Aggregation:

The GroupBy-Aggregate is a fundamental operation in many data mining applications such as web graph traversal. The MapReduce and SQL in database are two approaches to perform grouped aggregation. However, MapReduce is not efficient to process aggregation operations that have multiple inhomogeneous input data stream. For example, MapReduce PageRank is found in-efficient due to significant network traffic during the computation. Relational Databases like Oracle provides optimized mechanisms to integrate user defined aggregation function to SQL queries. But, the operations in full-feature SQL database has lots of extra overhead which prevents application from processing large scale input data.

Dryad programming model lies between SQL and MapReduce, and it can address some limitations of SQL and MapReduce. DryadLINQ CTP providers the partial aggregation strategy which aims to optimize the performance of some communication intensive applications. It builds the aggregation tree to perform partial aggregation for many sub data sets before hash partitioning them to make the global aggregation computation.

## III. Implementations

We implemented SW-G, Matrix-Matrix Multiplication, and PageRank with DryadLINQ CTP on Windows HPC R2 Clusters. The hardware resources we use in this paper are as follow:

Table 1. 32 nodes HPC cluster TEMPEST

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

Table 2. 7 nodes HPC cluster STORM

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

### 3.1 SW-G Application

The Alu clustering problem [6][7] is one of the most challenging problems for sequencing clustering because Alus represent the largest repeat families in human genome. There are about 1 million copies of Alu sequences in human genome, in which most insertions can be found in other primates and only a small fraction (~ 7000) are human-specific. This indicates that the classification of Alu repeats can be deduced solely from the 1 million human Alu elements. Notable, Alu clustering can be viewed as a classical case study for the capacity of computational infrastructures because it is not only of great intrinsic biological interests, but also a problem of a scale that will remain as the upper limit of many other clustering problem in bioinformatics for the next few years, e.g. the automated protein family classification for a few millions of proteins predicted from large meta-genomics projects.



Fig 2. Program Flow for SW-G Application

We implemented the DryadLINQ application to calculate the pairwise SW-G distances in **parallel** for a given set of gene sequences. To clarify our algorithm, let’s consider an example with 10,000 gene sequences, which produces a pairwise distance matrix of size 10,000 × 10,000. We decompose the overall computation into a block matrix D of size 8 × 8, each block contains 1250 × 1250 sequences in this case. Due to the symmetry of the distances D(i,j) and D(j,i), we only calculate the distances in the 36 blocks of the upper triangle of the block matrix as shown in Fig 2. Assuming there are 6 compute nodes, and we split the 36 blocks into 6 partitions each of which contains 6 blocks. Each Dryad tasks invokes the user defined function PerformAlignments() via ApplyPerPartition to apply Alu clustering computation to the 6 blocks that dispatched to them. The main component of DryadLINQ SW-G code is as follows:

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

#### Scheduling for inhomogeneous tasks

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

One solution to solve or alleviate this issue is to construct the SW-G blocks input data by randomly selecting sequences. To verify the second strategy, we manually generate a set of gene sequences with a given mean sequence length (400) with varying standard deviations following a normal distribution of the sequence lengths. We constructed the SW-G blocks input data by randomly selecting sequences from above data set as well as by selecting in a sorted order based on the sequence length. As it shown in Fig 3, the randomly distributed can deliver a better performance than skew distributed input data [1].

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

#### Scheduling for inhomogeneous cluster

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

 

Fig 4. CPU Time for SW-G Tasks with Various Task Granularities.

We verify the second approach by executing the 4096 sequences SW-G jobs on the inhomogeneous HPC STORM (Table 2) with different task granularity. Fig 4 shows the CPU time and task scheduling time for three SW-G jobs with different number of partitions: 6, 24, and 192. In the first SW-G job, the entire job is split into 6 partitions. The difference in CPU time for each task is caused by the difference in computational capability among nodes. It is clearly illustrated that finer granularity tasks can deliver a better work load balance on inhomogeneous computational nodes. However, it also shows that the task scheduling cost increase as the number of partitions increases.

## 3.2 Hybrid Parallel Programming

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

In this section, we will port the multi-core technologies to different algorithms and study their performance. By default the matrix in this section are square matrix, whose element is double number. The basic equation to calculate matrix-matrix multiplication is: 

### Matrix-matrix multiplication algorithms

The row split algorithm split the A matrix by rows, the program scatters the rows blocks of A matrix onto compute nodes. The whole B matrix is copied to every compute node. Each Dryad task multiplies some row blocks of A Matrix by entire B matrix, and retrieves the output results to main program to combine into C matrix.

The row/column split algorithm [18] splits the A matrix by rows and split the B matrix by columns. The whole computation consists of several iterations whose number is equal to the number of rows partition of A matrix. One Dryad task within each iteration multiply the row block of A matrix by the column block of B matrix. The output of Dryad tasks within the same iteration will be retrieved to the main program to aggregate one row block of C matrix. The main program collects results in multiple iterations to generate final output C matrix.

The two dimensional block decomposition in Fox algorithm split the A matrix and B matrix into square blocks matrix. And it make uses of square process mesh. For example, let us assume run this algorithm on a 2X2 processes mesh. Accordingly, the A matrix and the B matrix are split by both rows and columns and construct a 2X2 block mesh respectively. In each computation step, every process holds a block of matrix A and a block of matrix B and computes a block of matrix C. The algorithm is shown in Fig 5.

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

DistributedQuery<object> inputData = inputObjects.AsDistributed();

inputData = inputData.Select(data=>update(data));



Fig 5. Program Flow for DryadLINQ Matrix-Matrix Multiplication in Fox Algorithm

We evaluate the three algorithms by running matrix-matrix multiplication jobs with various input data from 2400 to 14400 with one core per node on 16 compute nodes (4X4 mesh). Fig 6 shows that the Fox algorithm can achieve the best performance among others for large input data. Comparing with other algorithm, the Fox has finer granularity task as it only calculate one block of a matrix and b matrix. This will cause the high cache hitting rate during the computation. The row/column algorithms perform the worst due to cost to launch Dryad vertex in each iteration.

Fig 6. Three Algorithms with Various Size of Data

## Parallelism in core level

We evaluated the multi-core technologies in .NET 4 by running matrix-matrix multiplication jobs with various size of input data from 2400 \* 2400 to 12000 \* 12000 on a 24-core Windows server. Fig 8 shows the performance results of matrix-matrix multiplication jobs for four different multi-core technologies. As illustrated in Fig 7, the PLINQ has the best performance among all the four algorithms.

Fig 7. Speed up for Different Method of Multi-core Parallelism on 24 cores Compute Node

### Port multi-core tech into Dryad task

We port the above multi-core technologies into the three matrix-matrix multiplication algorithms [10]. Fig 8 shows the relative speed up for three algorithms with different multi-core technology for 10000 square matrix multiplication jobs with 24 cores per node on 16 compute nodes. Fig 9 shows that the Fox algorithm has the worst performance results which are reversed from the results showed in Fig 7. In matrix-matrix multiplication, the computation cost O(n^3) increase faster than the communication cost O(n^2). Thus one of the main reasons is that after porting multi-core into Dryad task, the task granularity for the row split and row/column split algorithm becomes finer as well, which alleviates the low cache hit rate issue for coarse-granularity task.

Fig 8. Relative Speed up for Different Combination of Algorithms and Multi-core Technologies

As it illustrates in Fig 9, the communication cost of Fox algorithm is much larger than the other two algorithms. While its CPU utilization is less than the other two approaches. (Todo need replace with context switch profiling data in HPC cluster manager)



Fig 9. CPU & Network Utilization for Different Algorithms

## 3.3 Distributed Aggregation

We evaluated the distributed aggregation [11] in DryadLINQ by running PageRank applications with large real data on HPC cluster. We evaluated the interface and implementation of three different of distributed aggregation strategies in DryadLINQ which include: Group Aggregation, Partial Aggregation, and Hierarchical Aggregation. Besides, we compare the performance of DryadLINQ PageRank with the versions implemented with other runtimes MPI, Hadoop, Haloop [12], and Twister [13][14].

The PageRank is already well-studied web graph ranking algorithm. It calculates numerical value to each element of a hyperlinked set of web pages, which reflect the probability that the random surfer access those pages. The process of PageRank can be understood as a Markov Chain which needs recursive calculation to converge. An iteration of the algorithm calculates the new access probability for each web page based on values calculated in the previous computation. The iterating will not stop until the Euclidian distance between two subsequent rank value vectors less than a predefined threshold. In this paper, we implemented the DryadLINQ PageRank with the ClueWeb09 data set [15] which contains 50 million web pages.

We split the entire ClueWeb graph into 1280 partitions, each of which is saved as adjacency matrix (AM) file. The characteristics of input data we use in this paper is as follows: (B stands for Billions)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No of AM files | File Size | No of web pages | No of links | Ave out-degree |
| 1280  | 10GB | 49.5M | 1.40B | 29.3 |

#### Group aggregation

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

for (int i = 0; i < \_iteration; i++)

 {

 newVertices = pages.Join(vertices, page => page.source, vertex => vertex.source,

 (page, vertex) => page.links.Select(dest => new Vertex(dest, vertex.value / (double)page.numLinks)))

 .SelectMany(list => list).GroupBy(vertex => vertex.source).Select(group => new Vertex(group.Key, group.Select(vertex => vertex.value).Sum() \* 0.85 + 0.15 / (double)\_numUrls));

 vertices = newVertices;

 }

The **Page** objects are used to store the structure of web graph. Each element **Page** in collection pages contains a unique identifier number page.source and a list of identifiers specifying all the pages in the web graph that **page** links to. We construct the DistributedQuery<Page> **pages** objects from the AM files with function PagesFromAMFile(). The **rank** object is a pair specifying the identifier number of a page and its current estimated rank. In each iteration, the program first joins the **pages** with **ranks** and calculates the partial rank values. Then, the calculated partial rank values are flattened and hashed across the cluster with SelectMany(). Then, the dispersed partial rank values are collected into some groups with GroupBy() where each group represents a set of partial ranks with the same source page that point to them. The grouped partial rank values is accumulated with Sum(), and generate a new estimated rank value table.

#### Partial aggregation

The GroupBy PageRank is not efficiency due to excessive data movement in the hash partition stage. The partial aggregation can decrease the data transformation by applying the aggregation to many sub set of entire input data before the hash partition stage. DryadLINQ provide the GroupAndAggregate() operator that enable users to make use of partial aggregation strategy. We implemented PageRank with GroupAndAggregate() operator as follows:

for (int i = 0; i < \_iteration; i++)

 {

newVertices = pages.Join(vertices, page => page.source, vertex => vertex.source, (page, vertex) => page.links.Select(dest => new Vertex(dest, vertex.value / (double)page.numLinks))).SelectMany(list => list)

 .GroupAndAggregate(t => t.source, g => new Vertex(g.Key, g.Sum(x => x.value)\*0.85+0.15 / (double)\_numUrls));

vertices = newVertices;

 }

In ClueWeb data set, as the urls are stored in alphabet order, web pages belong to same domain are more likely saved in one AM file. Thus the intermediate data transformation in the hash partition stage can be greatly decreased by applying the partial aggregation to each AM file.

#### Hierarchical aggregation



Fig 10. Program Flow of PageRank with Hierarchical Aggregation

We also implemented DryadLINQ PageRank with hierarchical aggregation strategy via invoking user defined aggregation function in DryadLINQ task. Figure 10 shows program flow of implementation of hierarchical aggregation PageRank. There are three level aggregation stages: 1) the initial aggregation within user defined Map task. 2) the second level aggregation within each DryadLINQ partition. 3) the third level aggregation to calculate the global PageRank values.

Fig 11. Three Aggregation Strategies with Various Size of Input Data

Fig 11 shows the GroupAndAggregate out performs GroupBy by 30% for different size of data sets. The Hierarchical Aggregation out performs GroupBy when the input data are small. But, it does not perform as well as GroupBy when the input data get larger. One of the main reason is that computation cost in PageRank increase faster than that of communication cost. And partial aggregation strategy actually shift the communication workload to computation workload.

Fig 12. CPU and Network Utilization for Different Aggregation Strategies

As it is shown in Fig 12 clearly that the hierarchical aggregation has much less communication cost than other algorithms, while it requires much more computation cost. Basically, it just transforms the communication cost to the computation cost. As the communication cost does not increase as fast as computation cost, the hierarchical aggregation strategy does not perform as well as GroupBy for large input data.

Fig 13. PageRank Implemented with Five Runtimes

We also compare the performance of DryadLINQ PageRank with other runtimes: MPI, Twister, Hadoop, and Haloop. The Y axis in Fig 13 is the parallel efficiency which is defined as T(1)/n\*T(n), where n refers to number of cores used in experiments. The Dryad out performs Hadoop, and its performance is similar to Haloop, but much slower than MPI and Twister.

## IV. Discussion and Conclusion

We presented the design patterns for three typical scientific applications, and evaluate their performance. First we show that the developers can easily control the partition granularity with DryadLINQ interface to solve work load balance issue. In many job scheduling systems, the programmer has to manually group/un-group or split/combine input data to control task granularity. The Hadoop provide the interface that allows developers to control task granularity by reading input data from the HDFS. This is a much better improvement, but it still require developers know some detail about what’s the logic data format stored in HDFS. While DryadLINQ provider a simplified data model and interface for this issue.

Second, we investigated the hybrid parallel programming model with the matrix-matrix multiplication case. We show that porting multi-core technology can increase the overall performance significantly. And we show that different combination of algorithm in nodes level and multi-core technology in core level will affect overall performance of application.

At last, we studied the aggregation strategies in DryadLINQ CTP. We showed that the partial aggregation out performs GroupBy by 30% for PageRank application. We also show that the choice of DryadLINQ interface does not only affect the easiness of programming but also affects the performance of application.

# References:

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

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks”, European Conference on Computer Systems, March 2007.

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

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

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

[6] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, D. Thain, “All-Pairs: An Abstraction for Data Intensive Computing on Campus Grids”, IEEE Transactions on Parallel and Distributed System, 13, Mar. 2009.

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

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

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

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

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

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

[13] Twister, <http://www.iterativemapreduce.org/>

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

[15] ClueWeb09: <http://boston.lti.cs.cmu.edu/Data/clueweb09/>

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

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

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

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

[20] J. Ekanayake, A. Soner, T. Gunarathne, and G. Fox, C. Poulain, “DryadLINQ for Scientific Analysis”. 2009.