2013

Performance Evaluation of LINQ to HPC and Hadoop for Big Data

Ravishankar Sivasubramaniam
University of North Florida

PERFORMANCE EVALUATION OF LINQ TO HPC AND HADOOP FOR BIG DATA

by

Ravishankar Sivasubramaniam

A thesis submitted to the
School of Computing
in partial fulfillment of the requirement for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

May 2013

The thesis "Performance Evaluation of LINQ to HPC and HADOOP for Big Data" submitted by Ravishankar Sivasubramaniam in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

5/29/2013

Dr. Sherif Elfayoumy
Thesis Advisor and Committee Chairperson

Signature Deleted

5/29/2013

Dr. Roger Eggen
Committee Member

Signature Deleted

5/29/2013

Dr. Sanjay P. Ahuja
Committee Member

Accepted for the School of Computing:

Signature Deleted

5.29.'13

Dr. Asai Asaithambi
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Signature Deleted

7/2/13

Dr. Mark A. Tumeo
Dean of the College

Accepted for the University:

Signature Deleted

8/16/13

Dr. Len Roberson
Dean of the Graduate School

ACKNOWLEDGEMENT

I would like to express my gratitude to my thesis advisor Dr. Sherif Elfayoumy for the useful comments, remarks and engagement through the learning process of this master thesis. Furthermore I would like to thank Larry Snedden for the all the technical help and support. Also, I would like to thank Dr. Sanjay Ahuja, Dr.Roger Eggen and Dr. Asai Asaithambi for providing valuable suggestions.

I would like to thank my family and friends, who have supported me throughout entire process.

CONTENTS

FIGURES

TABLES

ABSTRACT

There is currently considerable enthusiasm around the MapReduce paradigm, and the distributed computing paradigm for analysis of large volumes of data. The Apache Hadoop is the most popular open source implementation of MapReduce model and LINQ to HPC is Microsoft's alternative to open source Hadoop. In this thesis, the performance of LINQ to HPC and Hadoop are compared using different benchmarks.

To this end, we identified four benchmarks (Grep, Word Count, Read and Write) that we have run on LINQ to HPC as well as on Hadoop. For each benchmark, we measured each system’s performance metrics (Execution Time, Average CPU utilization and Average Memory utilization) for various degrees of parallelism on clusters of different sizes. Results revealed some interesting trade-offs. For example, LINQ to HPC performed better on three out of the four benchmarks (Grep, Read and Write), whereas Hadoop performed better on the Word Count benchmark. While more research that is extensive has focused on Hadoop, there are not many references to similar research on the LINQ to HPC platform, which is slowly evolving during the writing of this thesis.

Chapter 1

INTRODUCTION

This thesis focuses on evaluating and comparing the performance of LINQ to HPC and Hadoop for unstructured data processing [LINQTOHPC12, HADOOP12]. With the growing volume of data captured, there is a huge interest for processing large sets of unstructured and structured data by organizations and scientific communities. Most Big Data processing systems take advantage of parallel and distributed computing architectures. Generally, the factors that are critical for processing large volumes of data are performance, cost, scalability and flexibility. Google's MapReduce programming model generated huge interest in parallel and distributed computing using commodity clusters [Dean08]. The MapReduce programming model greatly inspired Hadoop and LINQ to HPC implementations. Both platforms, Hadoop and LINQ to HPC, allow for processing unstructured and structured data on a cluster by distributing and managing the processing tasks.

In case of large data volumes, it is much more efficient for applications to execute computations near the data it operates on rather than moving the data where applications are running. This model increases the overall throughput and minimizes network congestion by reducing the time taken to move the data [HADOOP12]. This is one of the fundamental concepts behind LINQ to HPC and HADOOP.

Hadoop is a successful implementation of Google's MapReduce programming model and is now an Apache Foundation open source project. It enables the processing of large volumes of structured and unstructured data using cluster of commodity hardware in a simple, scalable, economical and reliable way. Hadoop is primarily installed on Linux clusters even though it could be installed on Windows platforms using emulators like Cygwin. Hadoop provides the Hadoop distributed file system, which can store and replicate data over a cluster using the MapReduce.

Cloudera CDH is an open source Apache Hadoop distribution coupled with Cloudera Manager to provide enterprise level support for advanced operations [CLOUDERA12]. Cloudera Manager provides graphical management capabilities to administer the Hadoop platform. CDH provides a streamlined path for implementing Hadoop platform and solutions. It delivers the core elements of Hadoop as well as the enterprise capabilities such as high availability, simple manageability, security, and integration with industry standard hardware and software solutions.

LINQ to HPC is a Microsoft research project formerly named DRYAD, which allows for distributed computing on the Windows Platform [LINQTOHPC12]. It was developed as the Hadoop alternative for Windows clusters. LINQ to HPC allows developers to process large volumes of unstructured data on a Windows cluster of commodity hardware. DSC (Distributed Storage Catalog) is a distributed file system to enable the storage and

replication of large data volume on clusters.

This thesis focuses on comparing the performance of both the Hadoop and the LINQ to HPC platforms through different experiments and using standard benchmarks on unstructured datasets. The motivation for this work comes from the increasing popularity of both platforms within organizations with Big Data processing needs. The results of these experiments should provide guidelines to practitioners on when to use each platform to achieve the best performance.

1.1 Overview of MapReduce

MapReduce is a programming model for processing large volumes of unstructured and structured data. It was originally developed by Google for processing Big Data to enhance search and Web Indexing [Dean08]. The MapReduce model is considered an efficient, scalable, and flexible distributed computing model for data intensive applications. The processing can take place on databases (structured) or file systems (unstructured). MapReduce takes advantage of computing near the data by decreasing data transfer latencies.

The MapReduce model partitions input data (key-value-pairs) and distributes tasks across the computing nodes of an underlying cluster. Key-value-pair is an abstract data type where key is a unique identifier for some item of data and value. The Map task process

the input key-value-pairs, the resultant intermediate from the Map tasks are then processed by the Reduce task to generate the output key-value-pairs.

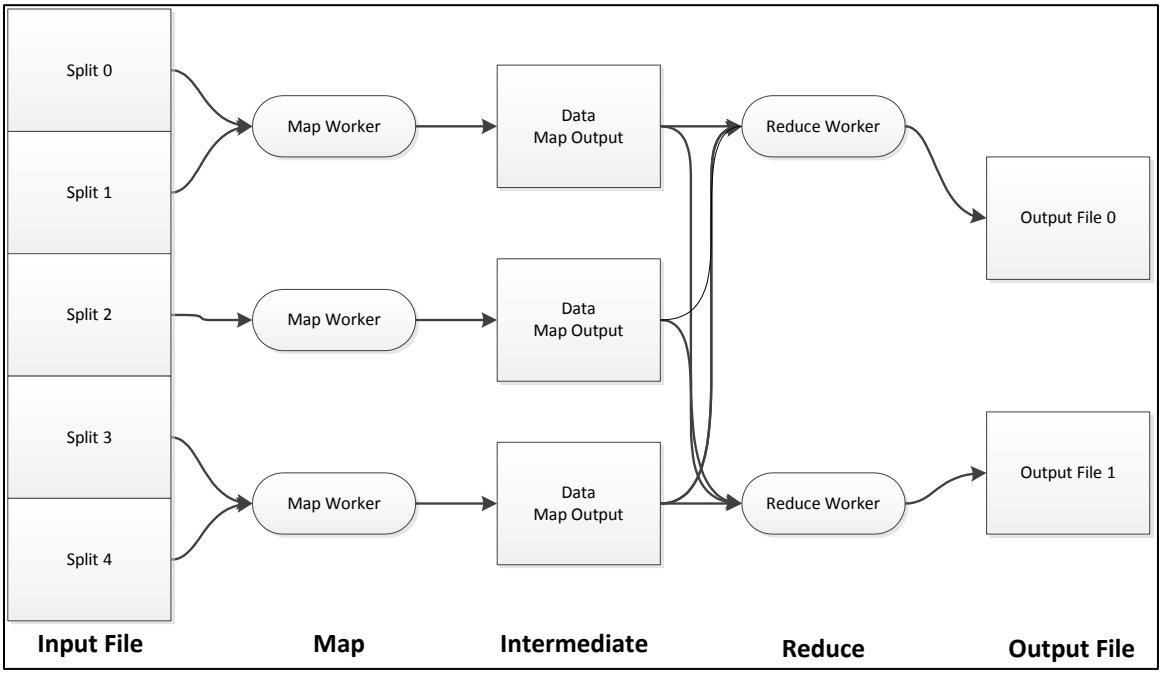

Figure 1: MapReduce Model

As illustrated in Figure 1, the Map function splits the input data into smaller problems and distributes them to Map Workers. Input splits are processed in parallel by Map Workers using different compute nodes. A Map Worker processes a smaller problem and passes its results back to the Master node. A Master/Head node is the primary node in cluster environments that consist of group of compute or process nodes. The MapReduce programming model collects the intermediate outputs and groups them together. The Reduce function is applied to each intermediate output in parallel, which produces the final output by combining the output from the map function. Figure 1 provides a depiction of the flow of actions that take place when a program calls the MapReduce

function.

1.2 Overview of Hadoop

Hadoop is a successful open source implementation of the MapReduce model. Hadoop includes a distributed file system called Hadoop Distributed File System (HDFS), which can store large sets of data on low-cost commodity hardware, and a MapReduce engine to process the data in a distributed environment [HDFS12]. Hadoop is reliable, scalable, cost effective, and efficient [HDFS12]. The Performance can be scaled linearly by adding more hardware resources to the cluster [HDFS12]. Hadoop has been successfully implemented in commercial environments with thousands of nodes processing petabytes of data [HDFS12]. Large corporations like Facebook, Yahoo, Amazon, LinkedIn, Visa and others have successful Hadoop implementations [HADOOP12A].

Hadoop is written in Java without specific hardware requirements. Hadoop supports a variety of operating systems including Linux, FreeBSD, Solaris, MAC OS/X and Windows.

1.2.1 Hadoop MapReduce Engine

The Hadoop MapReduce works similar to Google's MapReduce model that was discussed earlier. Hadoop MapReduce allows the processing of Big Data using

commodity hardware in a reliable, scalable and efficient manner. The Hadoop MapReduce engine provides features to enable scheduling, prioritizing, monitoring and failover of tasks [HDFS12].

The Hadoop MapReduce engine and Hadoop Distributed File System typically run on the same set of nodes in a cluster [HDFS12]. This allows the MapReduce engine to efficiently schedule the tasks where data resides. It also re-executes failed tasks. A task represents the execution of a single process or multiple processes on a compute node. A collection of tasks that is used to perform a computation is known as a job. A standard Hadoop cluster usually has a single master server and multiple worker or slave nodes. A worker is also called a compute node when it has a task tracker and called data node when it has data node. A master server consists of a name node, data node, job tracker and a task tracker. A worker node consists of a data node and task tracker. It is possible to have compute only nodes and data only nodes. The job tracker is responsible for scheduling and monitoring the task. The task tracker executes tasks as instructed by the job tracker. Figure 2 provides an architectural overview of a Hadoop system.

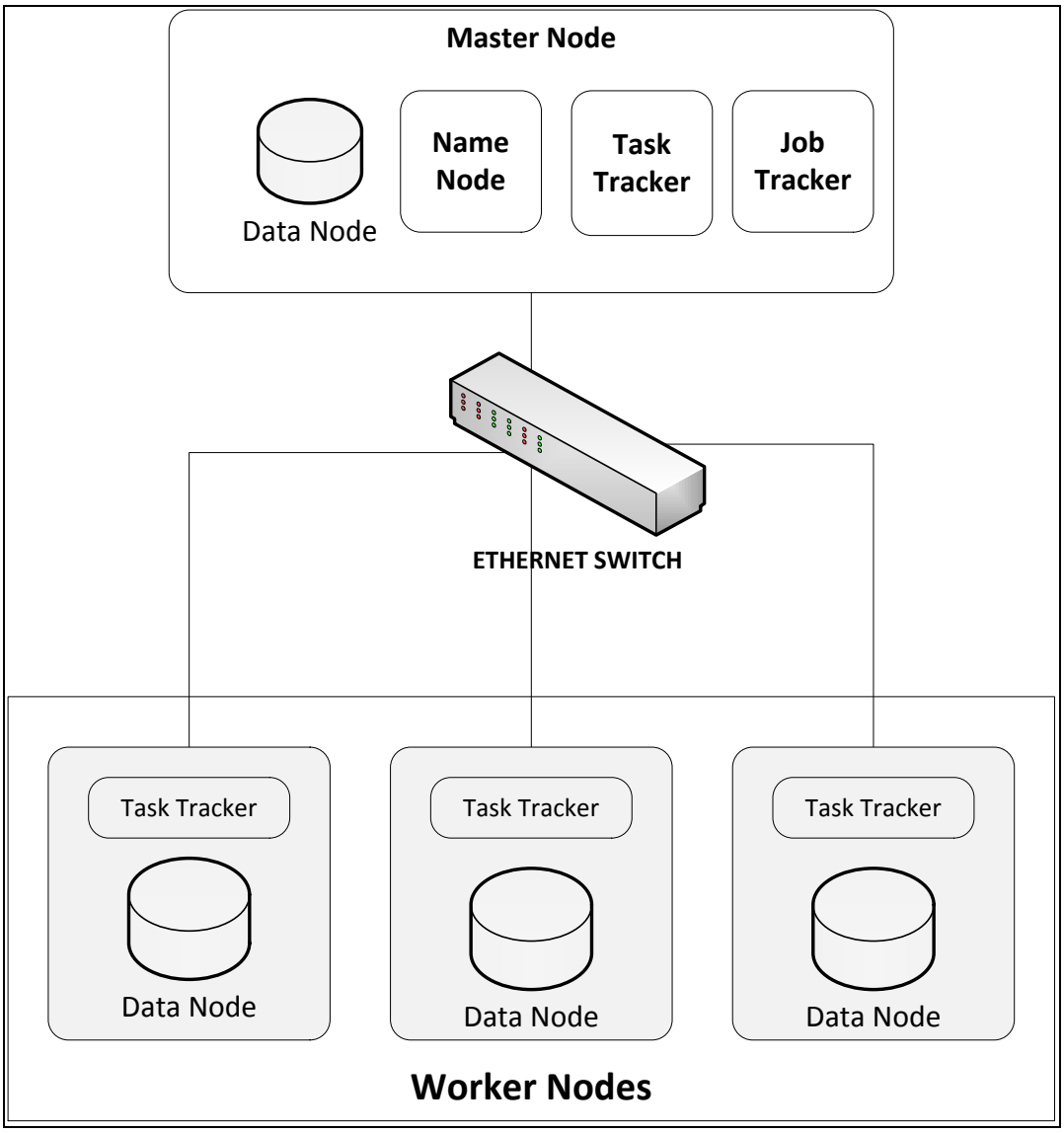

Figure 2: Hadoop Architecture

Typically, a Job configuration contains Input, Output, Map and Reduce functions along with other job parameters. The job tracker processes a task based on the Job configuration. The job tracker works along with the task trackers to process the job by distributing tasks to compute nodes in an efficient manner. However, Hadoop

MapReduce model is implemented in Java, MapReduce applications can be developed using any programming language.

1.2.2 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a distributed file system component of the Apache Hadoop platform [HDFS12, HDFS12A]. HDFS has many similarities to other existing distributed file systems. However, some advantages of HDFS over the existing distributed file systems include high fault-tolerance, scalability, ability to deploy on commodity hardware, and being open source. HDFS provides the interface for applications to move computation closer to data.

A typical HDFS cluster consists of a single name node and a number of data nodes. The NameNode is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself. Usually there are one data nodes per compute node in the cluster to manage the distributed storage.

HDFS stores files as a sequence of blocks across machines in the cluster. The Block replication provides data reliability and fault tolerance. Data files are divided into blocks and replicated to three data nodes by default. The replication parameter and block size are configurable per file and can be changed at any time. Moreover, applications can specify a different number of replicas.

The name node performs the file system namespace operations, such as opening, closing, and renaming files and directories. Data nodes serve the read and write requests from file system's clients. They also perform block creation, deletion, and replication upon instruction from the name node. The name node determines the mapping of blocks to data nodes and manages block replications based on heartbeat and block reports it receives periodically from the data node. Receipt of a heartbeat implies that the data node is functioning properly. A block report contains a list of all blocks on a data node. HDFS is highly fault-tolerant, and it can detect faults and recover lost and corrupt data automatically since data blocks are replicated.

1.3 Overview of LINQ to HPC

LINQ to HPC is a Microsoft product that provides a platform for creating and running applications which can process Big Data (structured and unstructured) on a cluster of commodity machines [LINQTOHPC12, Chappell11]. LINQ to HPC is built for Windows HPC Servers. It has three major components, namely, LINQ to HPC client, LINQ to HPC graph manager, and the Distributed Storage Catalog (DSC) [LINQTOHPC12, DSC12]. TLINQ to HPC provides a simple, scalable, reliable and cost effective platform for processing Big Data [Chappell11]. Figure 3 provides an architectural overview of a LINQ to HPC system.

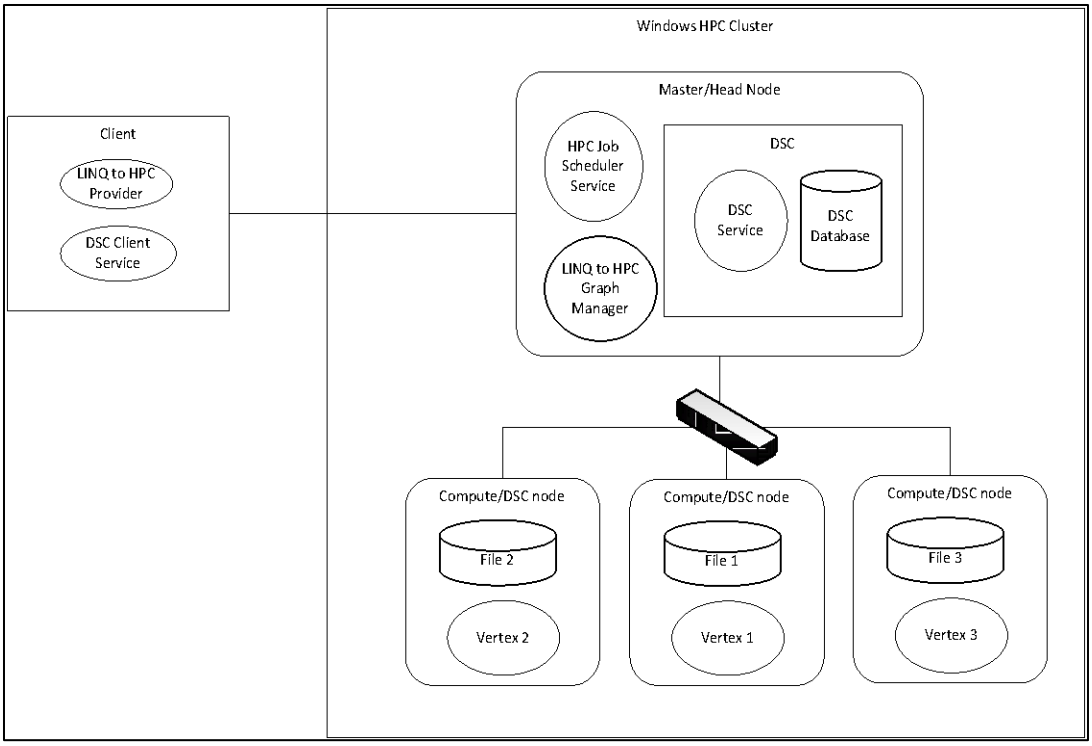

Figure 3: LINQ to HPC Architecture

LINQ to HPC uses the Dryad technology created by Microsoft Research [Israd09]. Dryad is similar to the MapReduce engine in Hadoop. Microsoft's search engine Bing, and Kinect are examples of the applications powered by Dryad. LINQ to HPC application run code on the cluster by creating LINQ to HPC queries that are executed in the runtime. LINQ to HPC application uses a graph model called Directed Acyclic Graph (DAG). The Directed Acyclic Graph defines the control flow and data flow dependencies among the individual tasks that are associated in executing a distributed query. Each node of the graph represents a unit of work called vertices, which will be performed by a single DSC node of the cluster using specific inputs and produce specific outputs. Dryad allows graph

vertices to use any number of input and output sets, whereas MapReduce restricts tasks to use one input and output set.

LINQ to HPC differentiates itself from Hadoop by providing an easy to use query based programming model. The programming model is based on Microsoft's Language Integrated Query (LINQ). Unlike the MapReduce model, the query based programming model is easy to comprehend, more expressive, and flexible [Chappell11].

LINQ to HPC creates an optimized execution plan for the query based on several factors, including the topology of the cluster. The query plan decomposes the grouped aggregation into efficient, distributed computations using the expression trees. The partial aggregation used by the plan greatly reduces the amount of network data transfer.

1.3.1 LINQ to HPC Client

The LINQ to HPC client contains two components, namely, the LINQ to HPC provider and DSC client services. LINQ to HPC provider resides on the client machine and analyzes application queries and creates an optimized execution plan to execute the LINQ to HPC queries. The provider communicates to the Windows HPC scheduler to initiate a LINQ to HPC job. The DSC client service manages data used by LINQ to HPC queries. The DSC client talks to the DSC service that runs on the Windows HPC cluster.

1.3.2 LINQ to HPC Graph Manager

LINQ to HPC graph manager is responsible for executing individual tasks that make up a LINQ to HPC job. An instance of LINQ to HPC graph manager is created for each LINQ to HPC job that is initiated by the job manager. The graph manager distributes computations across DSC nodes based on the execution plan created by the LINQ to HPC provider. The LINQ to HPC graph manager starts and stops vertices on the DSC node as needed. Additionally, it manages failures and assignment of tasks. The LINQ to HPC graph manager talks to the DSC service on the master node to assign a vertex to execute to a DSC node based on the execution plan.

1.3.3 Distributed Storage Catalog

The Distributed Storage Catalog (DSC) is a distributed file system that provides the ability to store large volumes of data across the cluster in a reliable, cost effective, fault-tolerant, and secure way [LINQTOHPC12, DSC12]. The DSC has a service that manages the data used by LINQ to HPC and a database that holds the catalog of the DSC file and file sets. The database also holds the metadata for the cluster including the location of files in the DSC node, file to file set mapping, properties of the file and file sets. The DSC service runs on the master node and the compute nodes can be configured to be controlled by DSC service and these nodes are called DSC nodes. DSC service allows for the creation of DSC files sets, which are logical groupings of DSC files.

The DSC nodes perform tasks assigned to them by the DSC service. Tasks may include file validation, file replication, reclaiming temporary storage, and performing the computations of each vertex. File replication provides data reliability and fault tolerance. Files are replicated to three DSC nodes by default. The replication is configurable and can be changed any time.

DSC file set is a collection of DSC files that is created and finalized and cannot be modified. LINQ to HPC provides a command line utility to perform basic file operations like adding a DSC file, managing permission, and deleting a file. LINQ to HPC queries can be used to interact directly with the data even though the DSC file set contains distributed data.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 1 provides a background into the MapReduce model, the architecture of Hadoop, and Microsoft's LINQ to HPC; Chapter 2 provides a literature review; Chapter 3 explains the research approach, the experimentation model, and provides a detailed description of evaluation metrics; Chapter 4 discusses the research methodology; Chapter 5 presents and discusses the experimentation results; Chapter 6 presents the conclusion and directions for future research.

Chapter 2

LITERATURE REVIEW

The MapReduce model, developed by Dean and Ghemawat, introduced a programming model and the associated implementation for distributed processing of large volumes of unstructured data using commodity hardware [Dean08]. The MapReduce model, implemented on Google's cluster by Dean and Ghemawat, had demonstrated good performance for sorting, and pattern searching (Grep) on unstructured data. Dean and Ghemawat had suggested a particular implementation of Grep that we have adopted in parts to carry out the benchmarking aspects of the experiments.

The Apache Hadoop website provides ample information on the implementation, sample code, and quick start guides to implementing Hadoop [HADOOP12]. The information provided on the Hadoop website was used for understanding the architecture and implementing Hadoop clusters.

MapReduce uses a stricter pipeline expression of distributed computations as compared to Dryad's expressive directed acyclic graphs (DAG) [Israd07]. DryadLINQ is an implementation of LINQ, a high level SQL query based language, for Dryad clusters [Israd09]. Compared to MapReduce, DryadLINQ offers an extended set of data operations to simplify writing complex algorithms [Dean08].

In 2009, Dinh et al. conducted a performance study of Hadoop Distributed File system for reading and writing data [Dinh09]. They used the standard benchmark program TestDFSIO.java that is available with the Hadoop distribution. Their study discussed the implementation, design, and analysis of reading and writing performance. In the experimentation part of this research, we adopted similar read and write benchmarks to the one discussed by the authors. We used the native Read and Write commands available in Hadoop and LINQ to HPC for the benchmarking purposes. We did not use TestDFSIO.java since a similar benchmark was not available for LINQ to HPC.

In 2009, Pavlo et al. discussed an approach to comparing MapReduce model to Parallel DBMS [Pavlo09]. As part of their experiments, they compared Hadoop, Vertica, and DBMS-X. The authors used benchmarks consisting of a collection of tasks that were run on the three platforms. For each task, they measured each system's performance for various degrees of parallelism on a cluster of 100 nodes. They used Grep, Aggregate, Join, and Selection tasks. In this research, we used the Grep and Aggregate benchmarks for our experiments. Join and Selection benchmarks can be used in the future to extend this research. The rest of the benchmarks and metrics used in this thesis are discussed in details in Chapter 3.

Ekanayake et al. discussed the use of DryadLINQ for scientific data analysis and compared the performance with Hadoop for the same application [Ekanayake09]. A scientific, proprietary, application was programmed by the authors as a benchmark for comparing the two systems. Our approach in this thesis focuses primarily on generic

benchmarks with limited modification to the sample programs provided by Hadoop and LINQ to HPC.

In 2010, Jiang et al. conducted a performance study of the Apache Hadoop on a 100-node Amazon EC2 cluster [Jian10]. They provided a detailed discussion of the design factors and performance tuning of the Apache Hadoop environment. They used Grep, Aggregate and Join benchmarks. Great parts of their approach were adopted in designing experiments for this research. We used similar benchmarks (Grep and Aggregate) and metrics in addition to few more benchmarks and metrics as discussed in Chapter 3.

Gonz´alez-V´elez and Leyton's research focused on evaluating the performance of Hadoop running in a virtualized environment [Gonzalez11]. They used a cloud running VMware with 1+16 nodes to evaluate the performance. The experiments were designed to use the Hadoop Random Writer and Sort algorithms to determine whether significant reductions in the execution time of computations were observed. The only metrics used in that research were execution time and CPU usage. For the purpose of our experimentation, we adopted a similar design approach using a virtualized environment, and used similar benchmarks and metrics as discussed in Chapter 3.

In 2011, Fadika et al. presented a performance evaluation study to compare MapReduce platforms under a wide range of use cases [Fadika11]. They compared the performance of MapReduce, Apache Hadoop, Twister, and LEMO. The authors designed the performance design test under the following seven categories: data intensive, CPU

intensive, memory intensive, load-balancing, iterative application, fault-tolerance and cluster heterogeneity. That study shed some light on the available design decisions, which can be used for future studies.

Chappell gave an introductory overview to the LINQ to HPC in his paper sponsored by Microsoft [Chappell11]. Also, The LINQ to HPC programming guide provides details on creating applications using LINQ to HPC [LINQTOHPC12A]. This guide was used in understanding and implementing our experiments. The LINQ to HPC SDK sample code provides a set of sample codes and programs [LINQTOHPC12B].

The Cloudera website provides information pertaining to the Cloudera distribution of Hadoop (CDH) [CLOUDERA12]. The Cloudera Installation Guide provides detailed systematic instruction on setting up CDH version 4 on Linux cluster [CLODERA12A]. The Cloudera Quick Start Guide was used to set up Cloudera and perform administrative tasks [CLODERA12B].

Forrester research rates Cloudera as a leader in Enterprise Hadoop Solutions market [FORRESTER12]. Cloudera, Amazon Web Services, EMC Greenplum, Horton Works, IBM, MapR, Outerthought, DataMeer, DataStax, Zettaset are some of the well established enterprise Hadoop-based solutions. All of these vendors offer MapReduce but not everyone offers HDFS. Amazon is the most prominent provider in Enterprise Hadoop market, but it does not offer a Hadoop hardware appliance. IBM and EMC are more

oriented towards the enterprise data warehouse market. Cloudera is a Hadoop vendor with, inarguably, the most adoption in enterprise.

The growing volume of unstructured and structured data has created huge opportunities for Big Data analysis. Hadoop has gained a lot of initial momentum with support from technology companies like Yahoo, Facebook, Amazon and others. There is currently no competitor to Hadoop in this space and the only product that stands a chance to compete with Hadoop is Microsoft's LINQ to HPC. In addition, Hadoop is an open source system and LINQ to HPC is a proprietary system, which makes the comparison even more interesting for many organizations and researchers.

Chapter 3

EVALUATION APPROACH

In this chapter, we discuss the approach followed in the design and implementation of experiments to compare the performance of Hadoop and LINQ to HPC platforms. The discussion will provide detailed information on the performance parameters, performance metrics, benchmarks, configurations, and datasets.

The goal of this research is to conduct a comprehensive comparison between Hadoop and LINQ to HPC with special emphasis on performance and resource utilization aspects. The fact that one of the systems, Hadoop, is Open Source and the other, LINQ to HPC, is commercial triggers a lot of interest in the results of this study. In order to, fairly and effectively, compare the two systems, the Cloudera Hadoop and LINQ to HPC were setup on clusters with the same configuration (Processors, RAMs and hard disks) on a virtualized environment with a total of eight nodes. One of the nodes was designated to play dual roles (both master and worker) and the remaining seven nodes were setup as worker nodes. Virtualization provided for the flexibility to vary the workloads and available resources to perform the experiments. The benchmarks (Grep, Word Count, Read and Write) programs were run on both Hadoop and LINQ to HPC, and the results of the performance metrics and resource utilization with varying load, and varying dataset sizes were recorded. Figure 4 provides an architectural overview of the experiments setup.

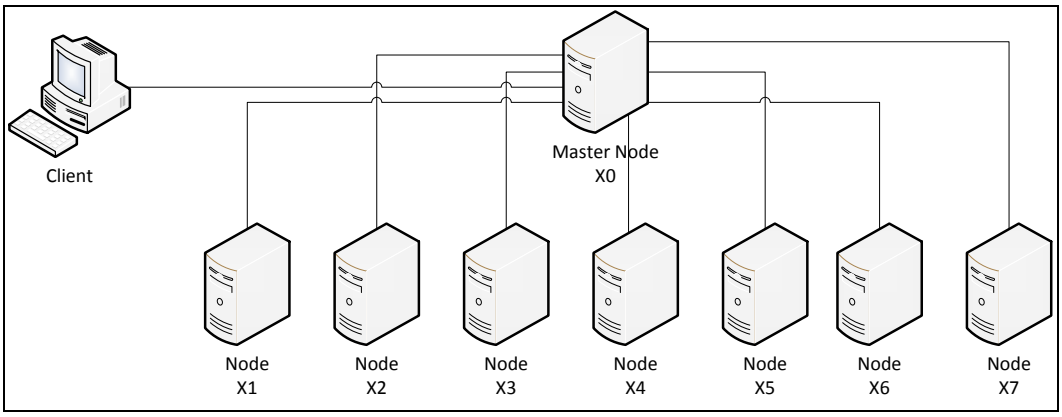

Figure 4: Experiment Architecture

3.1 Experimentation Overview

In this section, we present our experimental setup. The test bed was setup with the same configurations for Hadoop and LINQ to HPC clusters. Cluster size, dataset size, and benchmarks are the three independent experiment parameters. In total, we used six cluster configurations and two data sets along with four benchmarks to conduct a total of 48 experiments for each of Hadoop and LINQ to HPC Clusters. Average CPU usage, Average Memory Usage, and Execution Time were used as the performance metrics, or dependent parameters. Each experiment was repeated three times under the same conditions to reduce the impact of system fluctuation errors. In all, 96 experiments (288 runs) were conducted to capture the performance metrics.

3.1.1 Cluster Configuration Characteristic

The experiments used six cluster configurations, C3, C4, C5, C6, C7, C8 and C9, with varying number of nodes. Cluster configurations used varying number of nodes to study the scalability of the two platforms. For example, Cluster configuration C3 has one Master/Worker node and two Worker nodes, whereas Cluster configuration C8 has one Master/Worker node and seven Worker nodes. Each experiment was repeated three times as mentioned earlier. Table 1 provides the details of the cluster configurations.

Cluster Config. No.	C3	C4	C5	C6	C7	C8
Master Node	1	1	1	1	1	1
No. of Worker Nodes	2	3	4	5	6	7

Table 1: Cluster Configurations

3.1.2 Dataset Description

In this section, we discuss the details of the datasets used for the experiments. We used two datasets, D1 and D2, as shown in Table 2. The datasets were obtained from the Google Ngram dataset repository that is publicly available for download. Sizes of the dataset used were about 6GB and 18GB. These sizes were carefully chosen given the available cluster sizes and their configurations. Considering the hardware configuration used for the experimentation, size of the data is big.

We chose to use the Google Ngram data repository because of its size and public availability. Google's Ngram datasets are published by Google to provide the Books Ngram Viewer service. According to official Google research blog, these datasets were generated in 2009. Google specialists scanned over 5.2 million books, processed 1,024,908,267,229 words of running text, and published the counts for all 1,176,470,663 five-word sequences that appear at least 40 times in books. There are 13,588,391 unique words, after discarding words that appear less than 200 times. Data formatted as Tab-Delimited data. The format of the file is as follows:

Ngram TAB year TAB match_count TAB page_count TAB volume_count NEWLINE

A couple of examples using 1-grams are below:

```
circumvallate   1978   313    215   85
circumvallate   1979   183    147   77
```

The Google's Ngram repository has hundreds of files with each file around 1.2 GB in size. This provided us with flexibility in designing the experiments and allows for future extensions. The D1 dataset has four tab delimited files of 1.56 GB each, and data set D2 has 12 tab delimited files of 1.56 GB each. Both datasets were used to conduct the experiments and record the results. Appendix A provides the details of each data set.

3.1.3 Performance Metrics and Benchmarks

This section gives an overview of the benchmarks and metrics used in the experiments. The four benchmark tasks (Grep, Word Count, Read and Write), were used to evaluate and compare the performance of Hadoop and LINQ to HPC. The benchmarks were chosen based on the literary review conducted [Dean08, Dinh09, Pavlo09, Jian10, Gonzalez11]. The Read and Write Benchmarks were used to evaluate the performance of the distributed file system of Hadoop (HDFS) and LINQ to HPC (DFS). The Grep and Word Count benchmarks were used to evaluate the performance of the data processing engine of Hadoop (MapReduce) and LINQ to HPC (Dryad).

Read Benchmark involves loading the benchmark data set from local file system to the Distributed File system. Write Benchmark involves downloading the benchmark data set from Distributed file system to the Local File system. Grep Benchmark extracts matching strings from text files and counts how many times they occurred. Word Count Benchmark reads text files and counts how often words occur. The input is text files and the output is text files, each line of which contains a word and the count of how often it occurred, separated by a tab.

The three metrics were execution time, average CPU utilization, and average memory utilization. These metrics were selected based on the literature reviewed [Gonzalez11, Pavlo09]. Execution time and CPU utilization are commonly used metrics and many of the studies use these metrics to evaluate platforms performance. The three metrics were recorded and reported for the four benchmarks. Though the clusters were dedicated for

the experiments, i.e. no other programs were running, we decided to run each experiment for three times in order to eliminate any potential overhead introduced by routine housekeeping operations that might be coincidentally performed during experiment execution.

Based on the literature reviewed [Gonzalez11, Pavlo09] the average CPU utilization and average memory utilization were measured as percentages of the overall CPU time and available memory, respectively, while the execution time was measured in seconds. Average CPU usage was calculated by recording detailed CPU utilization during execution of each benchmark task for all the active nodes at a sampling rate of one second. The detailed CPU utilization was then aggregated by averaging the value across the nodes and time during the execution of each benchmark task. Average Memory usage was calculated by recording the detail memory utilization during execution of each benchmark task for all the active nodes at a sampling rate of one second. The detail memory utilization was then aggregated by averaging the value across the nodes and time during the execution of each benchmark task.

3.2 Architecture Overview

LINQ to HPC was installed on an eight node Windows HPC cluster with one master node and eight computing nodes where the master node acted as a computing node as well. Similarly, the Cloudera Hadoop was installed on an eight node Linux cluster with one master node and eight computing node where the master node, also, acted as a computing

node. Both clusters were configured similarly with70GB of hard drive space, 4GB RAM on the master node, and 2 GB RAM on each of the seven computing nodes.

3.2.1 LINQ to HPC Architecture

LINQ to HPC was installed on eight nodes Windows HPC Cluster. The Windows HPC cluster was setup on virtual machines running Windows HPC server 2008 R2 and LINQ to HPC was installed on all of the nodes. Client components LINQ to HPC provider and HPC client were installed on the client machine running windows. Visual studio 2010 was used to compile and run the benchmark programs. Figure 5 provides the architectural overview of the LINQ to HPC setup used in this research.

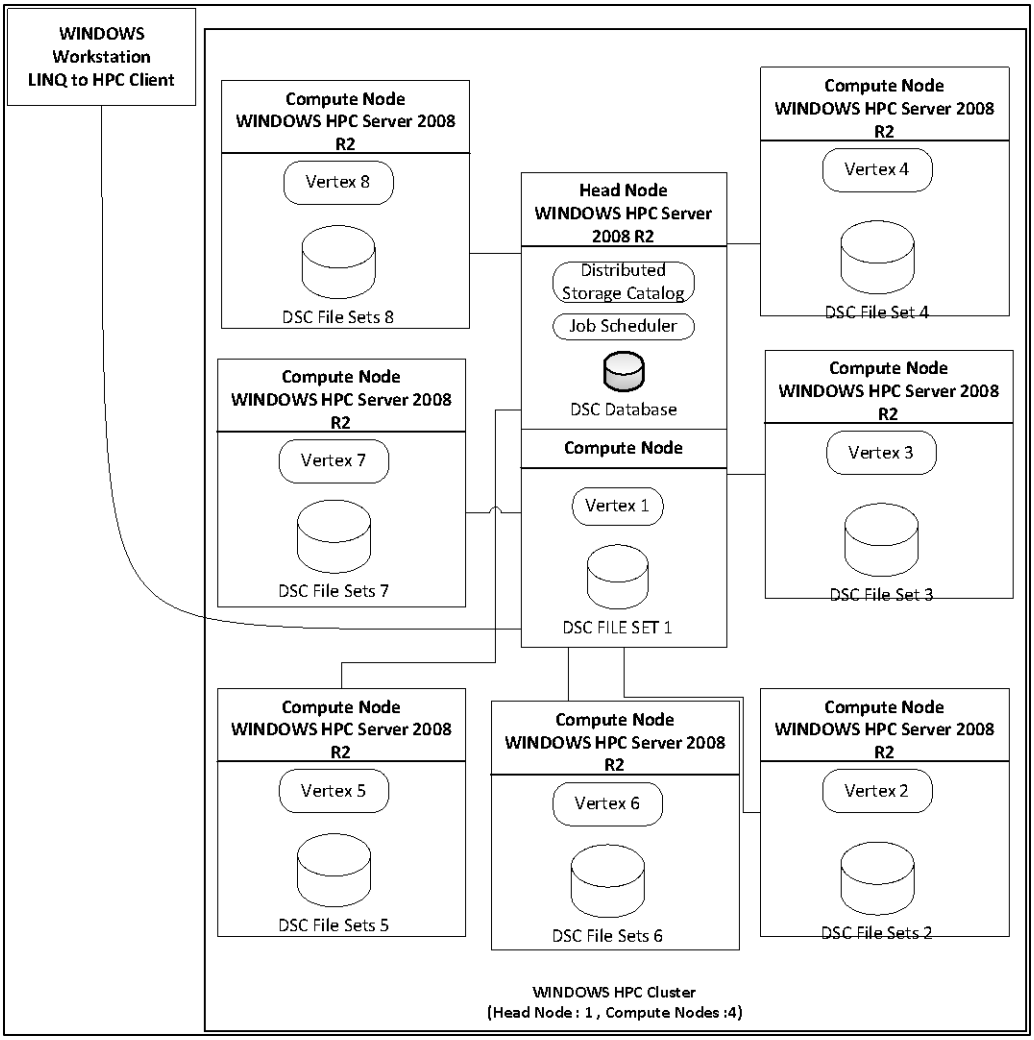

Figure 5: LINQ to HPC Experiments Architecture

3.2.2 Hadoop Architecture

The Cloudera Hadoop was installed on an eight nodes Linux Cluster with one master node and eight computing node. The master node acts as a computing node, as well. The Linux cluster was setup on virtual machines running CentOs Linux and Cloudera Hadoop. Client workstations ran CentOs Linux.

The Cloudera Installation was completed based on the systematic instruction available on Cloudera's installation guide [CLODERA12A]. Figure 6 provides an architectural overview of the Hadoop setup used in our experiments.

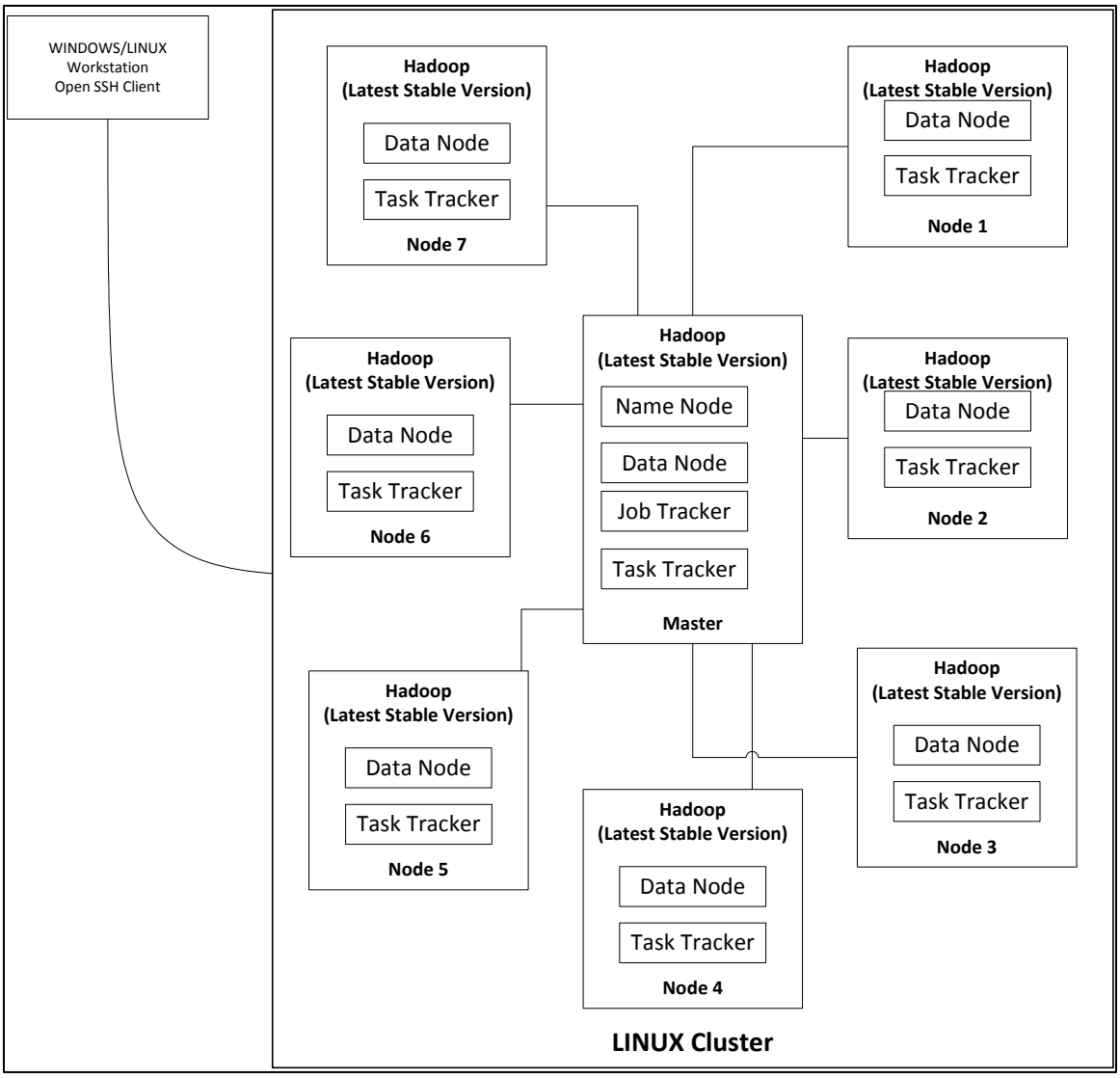

Figure 6: Hadoop Experiments Architecture

3.3 Hardware and Software Considerations

In this section, we discuss the hardware and software specifications used for The LINQ to HPC and Hadoop environments. The LINQ to HPC and Hadoop experiments were performed on hardware of identical specifications.

3.3.1 Hardware

The LINQ to HPC and Hadoop were setup on eight 64-bit virtual machines. Each machine used Ext3 file systems with a virtual hard drive of 70 GB and 4 GB RAM on the master node and 2 GB of RAM on slave nodes, and each with a single virtual dual core processor Xeon 5150 2.66 GHZ. The hypervisor was Microsoft HyperV 6.1.

3.3.2 Software

The LINQ to HPC Beta 2 was installed on a Windows HPC cluster running WINDOWS 2008 R2 server edition. Visual Studio 2010 was used to compile the benchmarks.

Cloudera Distribution of Hadoop CDH 4.0.1 (Apache Hadoop 2.0, Cloudera Manager 4.0) was installed on a Linux cluster running CentOS 6.2.

Chapter 4

RESEARCH METHODOLOGY

This section provides an overview of the processes and methodologies adopted for modeling the experiments, configuring and executing the benchmarks, and collecting and processing of the results.

4.1 LINQ to HPC Cluster

LINQ to HPC applications setup uses a High Performance Computing (HPC) cluster to process a large volume of data. The LINQ to HPC and the Distributed Storage Catalog (DSC) contained services that run on a HPC cluster, as well as client-side components that are invoked by applications. LINQ to HPC setup involves the installation of Windows HPC Cluster, LINQ to HPC on all the nodes in cluster, LINQ to HPC on client machine and Configuring LINQ to HPC.

4.1.1 Configuring Windows HPC Cluster

The LINQ to HPC was setup on a Windows HPC cluster consisting of eight Windows HPC Server 2008 R2, 64 bit virtual machines. The virtual machines were created using Microsoft HyperV hypervisor. Each virtual machine was configured with a single dual core processor, ext3 file systems with a virtual hard drive of 70 GB, and 2 GB of RAM, except the master node was assigned 4 GB RAM.

4.1.1.1 Windows HPC Cluster Setup

The installation of LINQ to HPC was performed with accordance to the procedure described in HPC documentation [HPC12]. WINDOWS HPC Server 2008 R2 was installed on eight virtual machines. The master node and compute nodes in the HPC cluster were added as members of an Active Directory domain. The HPC Cluster was setup by execution HPC Pack 2008 Express R2. The configuration of the master node was completed first, and was followed by the configuration of the compute nodes.

4.1.1.2 LINQ to HPC Setup

After completing the Windows HPC Cluster setup LINQ to HPC and the DSC were setup on the cluster. The following steps were followed to install LINQ to HPC on each of the cluster's eight nodes. LINQ to HPC Beta 2 was installed on each of the nodes. During the installation process installation, type (master or compute) was set based on the type of node.

4.1.1.3 LINQ to HPC Client Setup

The client machine has to have HPC Cluster Manager client version before the installation of LINQ to HPC client components. This procedure installs the HPC Cluster

Manager, the HPC Job Manager, and HPC PowerShell on the client machine. The following steps were performed to install the software on the client. Install the HPC Pack 2008 R2 Express by following the installation wizard. On the Select Installation Type page, select Install only the client utilities and follow the wizard. The next step is to proceed with LINQ to HPC Client installation. Open the LINQ to HPC Beta 2 download and execute LINQ to HPCSetup.exe. The Microsoft LINQ to HPC Beta 2 Installation Wizard appears and follow the wizard's instruction. On the Select Installation Type page, select Install LINQ to HPC on a client and follow the wizard.

4.1.1.4 LINQ to HPC Configuration

The configuration of LINQ to HPC involves defining a node group, adding users to the cluster, adding nodes to the DSC and configuring a replication factor.

A new node group, LinqToHpcNodes, was added to the groups by using the HPC Cluster Manager in the client machine. Users must be members of the HPC Users group on the cluster to use the DSC and submit LINQ to HPC jobs. Using the HPC Cluster Manager Utility in the client machine, a user was added to the new node group.

Each node was added to the DSC service using the DSC NODE ADD command. On the client machine using the HPC power shell client, the below command was used to add master node DRYAD1 node to the DSC.

```
DSC NODE ADD DRYAD1 /TEMPPATH:c:\L2H\HpcTemp /DATAPATH:c:\L2H\HpcData
/SERVICE:DRYAD1
```

The replication factor for LINQ to HPC was set to three using the DSC PARAM SET command. On the client machine using the HPC power shell client, the following command was executed to set the replication factor.

```
DSC PARAMS SET ReplicationFactor 3
```

4.2 Executing LINQ to HPC Benchmarks

The four benchmarks (Read, Write, Grep and Word Count) were run from the LINQ to HPC client running on the client machine. The Read and Write benchmarks were run using the standard command used to put files in DSC and get files from DSC. The Grep and Word Count used the sample program that is available in Microsoft Software Developers Network (MSDN) [LINQTOHPC12B].

4.2.1 Write Benchmark

The Write benchmark uses the DSC command FILESET ADD to load files from the client machine to the DSC cluster. The ADD command creates a new file set with the name specified. It uploads files from source directory to the DSC. The NTFS permissions on the file sets are based on the User group and privileges. Owner and administrators of the file set have full control permissions, whereas users in the Authenticated Users group

have read permissions. The /service option specifies the name of the cluster's master node.

Syntax

```
DSC FILESET ADD sourceDirectory targetFileSetName [/service:headnode]
[/public]
```

The following command was used to create a new fileset, NGRAM, and copy files from local folder D1 to file set NGRAM in DSC.

```
DSC FILESET ADD \\THOTH\Share\DATASET\D1 NGRAM /service:dryad
```

4.2.2 Read Benchmark

The Read benchmark uses the DSC command FILESET Read command to get files from DSC to the local client machine. The Read command downloads files from the file set that is specified as target FileSet name, to the local client directory specified as the target directory. The /service option specifies the name of the cluster's master node.

Syntax

```
DSC FILESET Read targetFileSetName targetDirectory [/service:headnode]
```

The following commands copies file set D1 from DSC to local folder RC3D11.

```
DSC FILESET read D1 \\THOTH\Share\DATASET\RC3D11  /service:DRYAD1
```

4.2.3 Grep Benchmark

The Grep benchmark on LINQ to HPC was run using the “FGrep” sample program, which is a sample implementation of the UNIX Grep command. The FGrep sample uses a single LINQ to HPC query to return the matching lines. The following sample code provides the LINQ to HPC query used for pattern search.

```
//FGrep SAMPLE

int count = 0;
foreach (LineRecord line in context.FromDsc<LineRecord>(fileSetName)
    .Where(r => regex.IsMatch(r.Line)))
{
    Console.WriteLine(line);
    count++;
}
Console.WriteLine("\nFound {0} matching lines.", count);
```

From the command line in the directory containing the FGrep binary, we ran FGrep and passed in three arguments. The first argument was the name of the Input DSC file set, the second was the name of the output file set, and the third was the regular expression to search for in each line.

Syntax

```
FGrep <INPUT FILE SET <OUTPUT FILE SET> <SEARCH STRING>
```

For example, the following command searches the file set named “NGRAM” for all lines that contain the word “and” and output the result to DSC file set “OUPUT”. The below example uses the word “and” as the search string.

```
FGrep “NGRAM” “OUTPUT” and
```

Below is a sample output produced by the Grep benchmark for the above “and” search string where the second token represents the search string frequency.

```
and 8159675
```

4.2.4 Word Count Benchmark

The Word Count benchmark uses the sample program “MapReduce”. The MapReduce sample program counts the occurrences of words in a DSC file set. The following sample code explains the main aspects of the Word Count benchmark.

```
// Define a map expression:

Expression<Func<LineRecord, IEnumerable<string>>> mapper = (line) =>
    line.Line.Split(new[] { ' ', '\t' },
StringSplitOptions.RemoveEmptyEntries);

// Define a key selector:

Expression<Func<string, string>> selector = (word) => word;

// Define a reducer (LINQ to HPC is able to infer the
// Decomposable nature of this expression):

Expression<Func<string, IEnumerable<string>, Pair>> reducer =
```

```
    (key, words) => new Pair(key, words.Count());

// Map-reduce query with ordered results and take top 200.

IQueryable<Pair> results =
context.FromDsc<LineRecord>(inputFileSetName)
    .MapReduce(mapper, selector, reducer)
    .OrderByDescending(pair => pair.Count);
```

A new Fileset was created using the DSC FILESET ADD command to load the dataset onto the cluster. From the command line in the directory containing the MapReduce binary, we ran MapReduce and passed two arguments. The first argument was the name of the Input DSC file set, and the second was name of the output file set.

Syntax

```
MapReduce <INPUT FILE SET>  <OUTPUT FILE SET>
```

For example, the following command uses the file set named “NGRAM” as input and counts the occurrences of each word in the input file set. The result is stored in the output file set “OUPUT”.

```
MapReduce “NGRAM” “OUTPUT”
```

Below is a sample output produced by the MapReduce benchmark. The first token in each line represents a word from the fileset and the second token in each line represents the frequency.

```
and  89
are  24
get 41
is 76
```

4.3 Collecting Metrics and Processing Results for LINQ to HPC

This section discusses in detail the methodology used to collect and process the metrics data. It is worth mentioning that the experiments were conducted in a sequential fashion using PowerShell scripts and the metrics data were collected using Windows HPC cmdlets utility [HPCCMDLET12]. The metrics data were redirected to flat files, as opposed to noting them down from the screen. The logs and metrics data were imported to Oracle database tables and were aggregated, as explained below, for analysis.

4.3.1 Metrics Collection

Windows HPC Server 2008 R2 provides a cmdlet utility that can be used to get information about jobs, nodes, and metrics for building custom reports. We used the Cmdlet “Get-HpcMetricValueHistory” to collect the values of the specified metric based on a specified time period.

As discussed in the earlier section, we collected three metrics for each experiment. These metrics were run time, average CPU usage, and average memory usage. The run time was obtained by running the “Get-Date” cmdlet in HPC PowerShell before and after the execution of benchmarks and redirecting the results to the log. The start and end times are important not only to measure the run time but also to extract the average CPU and average memory. The Get-HpcMetricValueHistory cmdlet was used to query the HPC database to obtain the average CPU usage and average memory usage metrics between the start and end times.

The results of cmdlet Get-HpcMetricValueHistory be filtered using metric names, node names as well as counter parameters. Except StartDate and EndDate parameters, all the other parameters are optional. When used without the optional parameters, cmdlet retrieves the values of all the counters, for all of the metrics, and on all of the nodes of the HPC cluster.

Syntax

```
Get-HpcMetricValueHistory [-StartDate] <DateTime> [-EndDate] <DateTime> [-Counter <String> ] [-MetricName <String> ] [-NodeName <String> ] [-Scheduler <String> ] [ <CommonParameters>]
```

cmdlet was used without the optional parameters and using only the StartDate and EndDate to collect the metrics values. The results were pipelined and exported to a flat file using the cmdlet utility Export-CSV. Below is a sample command.

```
Get-HpcMetricValueHistory -StartDate "Monday, April 02, 2012 2:34:00 PM" -EndDate "Monday, April 02, 2012 2:36:33 PM" | Export-Csv c:\dryad\log.csv
```

We collected HPCCpuUsage and HPCPhysicalMem metrics. The HPCCpuUsage collects percentage CPU usage for all processors on the compute node, and HPCPhysicalMem collects the available physical memory on the compute node in megabytes. The command creates a flat file with the following details: Node name, metric name, time and value. The output data was rolled-up to the minutes with a sampling rate of one second. Below is a sample output produced by the Get-HpcMetricValueHistory command.

```
DRYAD1,HPCCpuUsage,_Total,"4/9/2012 11:40:00 AM",3.369397
DRYAD1,HPCCpuUsage,_Total,"4/9/2012 11:41:00 AM",46.00943
DRYAD1,HPCCpuUsage,_Total,"4/9/2012 11:42:00 AM",3.574733
DRYAD1,HPCPhysicalMem,,"4/9/2012 11:40:00 AM",1721.61
DRYAD1,HPCPhysicalMem,,"4/9/2012 11:41:00 AM",1439.224
```

4.3.2 Aggregating the Results

The data collected in logs were imported to relational database tables. The data in the tables were then queried for aggregation and production of summarized reports. For the LINQ to HPC experiments we used two tables. One table stores the start and end times of the experiments with one record per experiment. The data extracted from the metrics were stored in a results table with the time, node name, metric name and metric value. The experiments produced 200,000 records in the results table. These two tables were joined and grouped to form an aggregate view of the result details, execution time in seconds, percent average CPU usage, and percent average memory usage. This consolidated view was used to prepare the charts and analyze the results.

4.4 Cloudera Hadoop Cluster

The Cloudera Hadoop cluster was created using an eight-node Linux cluster. The setup and configuration involved the installation of the Cloudera manager and CDH on the cluster followed by the setup of client and configuration of the cluster.

4.4.1 Cloudera Hadoop Cluster Configuration

The cluster, created using Microsoft HyperV, consists of eight 64-bit virtual machines running CentOS 6.2. Each machine uses a single dual core processor, Ext3 file systems with a virtual hard drive of 70 GB, and 4 GB RAM on the master node and two GB of RAM on compute nodes. The nodes had DNS entries and reserved IP addresses. Nodes also had IPtables enabled, and SELinux disabled. The Cloudera distribution of Hadoop (CDH 4.0) was installed without any custom tuning from the default installation scripts. All nodes had pre shared SSH in order to communicate properly.

4.4.1.1 Installation of Cloudera Manager and CDH

The Cloudera Manager and CDH setup was performed using the automated installation script provided by Cloudera. The Cloudera Manager was installed by executing the Cloudera Manager installer with the default settings. The CDH was installed using the Cloudera Manager Admin console. The Cloudera manager was used to install the CDH

on the client, and the client configuration file generated by the Cloudera Manager for the cluster was download and deployed manually.

4.5 Executing Hadoop Benchmarks

The four benchmarks (Read, Write, Grep and Word count) were run from the Linux client running on the client machine. The Read and Write benchmarks were run using the standard Hadoop HDFS File system (FS) shell command. FS shell commands were used to put files in HDFS and get files from HDFS. The Grep and Word Count use the sample program provided as part of the standard Hadoop examples.

The File System (FS) shell was invoked using “bin/hadoop fs <args>” [HDFS12B]. The commands in FS shell behave similar to the corresponding UNIX commands. All FS shell commands take path URIs (scheme://authority/path) as arguments. For HDFS, the scheme is hdfs, and for the local file-system the scheme is file. Scheme and authority are optional, and if not specified the default scheme as specified in the configuration file is used.

4.5.1 Write Benchmark

The Write benchmark uses the HDFS command “fs –put”, as shown below, to load files from the client machine to the HDFS. The put command copies files from source

directory from local client machine to the HDFS file system. In addition, it reads input from stdin device and writes to the destination file system.

Syntax

```
hadoop fs -put <localsrc> ... <dst>
```

The - -config option was used to specify explicitly the Hadoop configuration file in the client machine.

```
hadoop --config /usr/conf  fs -put /usr/dataset/d2 /d2;
```

4.5.2 Read Benchmark

The Read benchmark uses the HDFS command “fs –get” to copy/read files from HDFS to the local client machine, as shown below. The get command copies files to the local client directory from the HDFS file system.

Syntax

```
hadoop fs -get [-ignorecrc] [-crc] <src> <localdst>
```

Here too, the - -config option was used to specify explicitly the Hadoop configuration file in the client machine.

```
hadoop --config /usr/conf  fs -get /d2 /usr/dataset/output/C5D21;
```

4.5.3 Grep Benchmark

The Grep benchmark uses the “grep” sample program provided as part of the hadoop-examples-0.20.2-cdh3u4.jar package. The grep is a map-reduce implementation of the UNIX Grep command. This map-reduce program counts the matches of a regular expression in the input files. First, we used the fs -put command to load the dataset onto the HDFS cluster. Next, we executed the grep program from the client by and passing three arguments. The first argument is the name of the input HDFS file location, the second is the location in HDFS where results have to be stored, and the third is the regular expression to search for in each line.

Syntax

```
hadoop /usr/lib/hadoop/hadoop-examples-0.20.2-cdh3u4.jar grep <Input hdfs file location> <output file location> <Regular expression>
```

In the below example, the command searches the file set d2 for all lines that contain the search string “and” and results are stored under /C7D2R1. The experiment was repeated three times for datasets D1 and D2 and for the six cluster configurations.

```
hadoop --config /usr/conf jar  /usr/lib/hadoop/hadoop-examples-0.20.2-cdh3u4.jar grep /d2 /C7D2GR1 'and';
```

Below is a sample output produced by the Grep benchmark. The first token represents the frequency and the second token represents the value of the search string.

```
8159675 and
```

4.5.4 Word Count Benchmark

For the Word Count benchmark, we adopted the “wordcount” program that was provided as part of the hadoop-examples-0.20.2-cdh3u4.jar package. The word count is a map-reduce implementation to count the occurrences of each string token in the input file set. First, we used the fs -put command to load the dataset onto the HDFS cluster. Next, we executed the wordcount program from the client by passing two arguments. The first argument is the name of the HDFS file set for which the word count has to be performed, and the second argument is the name of the output directory under HDFS where the output will be stored.

Syntax

```
hadoop /usr/lib/hadoop/hadoop-examples-0.20.2-cdh3u4.jar wordcount
<Input hdfs file location>  <output file location>
```

In the below example, wordcount counts the occurrence of each word for the files under directory /d2 in HDFS and aggregates the counts. The results of the wordcount program are stored in directory /C7d2WC2.

```
hadoop  --config /usr/conf jar  /usr/lib/hadoop/hadoop-examples-0.20.2-cdh3u4.jar wordcount /d2 /C7D2WC2;
```

Below is a sample output produced by the Word Count benchmark. The first token in each line represents a word from the file set and the second token represents the aggregated frequency of that word.

```
and 89
are 24
get 41
is 76
```

4.6 Collecting Metrics and Processing Results for Hadoop

This section provides a detailed discussion on the methodology used to collect and process the metrics data. It is worth mentioning that the experiments were conducted in a sequential fashion using Linux shell scripts and the metrics data were collected using the SAR command. The metrics data were redirected to flat log files, as opposed to noting them down from the screen. The logs and metrics data were imported to database tables and were aggregated, as explained below, for analysis.

4.6.1 Metrics Collection

As discussed in the earlier section, we collected three metrics for the experiment. These metrics were: run time, average CPU usage, and average memory usage. Timestamps were obtained by running the command “date” in Linux shell before and after the execution of benchmarks and redirecting the results to log files. SAR command was run in the background in each node to capture the CPU and memory activity with a sampling rate of 1 second. The output from SAR was redirected to a log file. The flat file logs were imported to Oracle database tables for analysis. The Linux utility SAR reports the measures of selected cumulative activity counters in the operating system.

Syntax

```
sar [ -A ] [ -b ] [ -B ] [ -C ] [ -d ] [ -h ] [ -i interval ] [ -m ] [
-p ] [ -q ] [ -r ] [ -R ] [ -S ] [ -t ] [ -u [ ALL ] ] [ -v ] [ -V ] [
-w ] [ -W ] [ -y ] [ -n { keyword [,...] | ALL } ] [ -I { int [,...] |
SUM | ALL | XALL } ] [ -P { cpu [,...] | ALL } ] [ -o [ filename ] | -f
[ filename ] ] [ -s [ hh:mm:ss ] ] [ -e [ hh:mm:ss ] ] [ interval [
count ] ]
```

For the experiments, two SAR commands were run one for capturing CPU utilization and another for capturing memory utilization. The commands were run on each node, as background processes and the resultant files were transferred and consolidated at the client machine.

The following command captured the cpu activity and redirected the result to sar.cpu.log at a sampling rate of one second. The command runs as a background process.

```
Sar –u 1 > sar.cpu.log &
```

The sample output shown below is the result of running the SAR command with –u option.

```
Linux 2.6.32-220.17.1.el6.x86_64 (CISHADOOP1.ccec.unf.edu)  07/21/2012
       _x86_64_      (1 CPU)
03:45:53 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
03:45:54 PM     all     10.10      0.00     10.10      0.00      0.00     79.80
03:45:55 PM     all      7.22      0.00      8.25      0.00      0.00     84.54
03:45:56 PM     all      2.97      0.00      8.91      0.00      0.00     88.12
03:45:57 PM     all      2.00      0.00      8.00      0.00      0.00     90.00
03:45:58 PM     all      2.00      0.00     10.00      0.00      0.00     88.00
```

The following command captured the memory activity and redirected the result to sar.memory.log at a sampling rate of one second. The command runs as a background process.

```
Sar –r 1 > sar.memory.log &
```

The sample output shown below is the result of running the SAR command with –r option.

```
Linux 2.6.32-220.17.1.el6.x86_64 (CISHADOOP1.ccec.unf.edu)  07/21/2012     _x86_64_
       (1 CPU)

03:45:53 PM kbmemfree kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit
03:45:54 PM   2676704   1239084     31.64     30656    187400   1010124     10.02
03:45:55 PM   2676704   1239084     31.64     30656    187416   1010444     10.03
```

4.6.2 Aggregating the results

The data collected in log files were imported to relational database tables. Data in the tables were then processed for aggregation and provide summarized reports. For the Hadoop experiments, we used three tables. One table stored the start and end times of the experiments with one record per experiment. The metrics data extracted from the SAR command were stored in two result tables one for CPU and another for memory utilization. The experiments produced approximately 5 million records in the results tables. These tables where joined and grouped to form an aggregate view of experiment details, execution time in seconds, percent average CPU usage and percent average memory usage. This consolidated view was used to prepare the charts and analyze the results.

Chapter 5

ANALYSIS OF RESULTS

The discussion and analysis of the results for both Hadoop and LINQ to HPC are organized by the benchmarks, i.e., Grep, Word Count, Read, and Write, and the results of each of the benchmarks are summarized by the different cluster configurations, dataset and metric.

5.1 Grep Benchmark Results

Grep benchmark results for the Hadoop and LINQ to HPC are summarized in Table 2 for the different cluster configurations (C3, C4, C5, C6, C7, and C8) and datasets (D1 and D2) for average execution time, percent CPU utilization, and average percent memory utilization. The values presented in the table represent the average value of three different runs.

CONFIGURATION	DATA SET	Execution Time (S)		Average CPU (%)		Average Memory (%)	
		HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC
C3	D1	365.67	143	87	20	94	30
	D2	991.67	353	92	23	95	29
C4	D1	260.67	126	84	13	93	34
	D2	740.67	265	89	22	94	33
C5	D1	238.33	124.33	81	11	94	34
	D2	648.33	224.67	88	19	94	34
C6	D1	241.67	103.33	75	10	94	34
	D2	561.67	208	87	17	94	34
C7	D1	208	103.33	74	9	94	35
	D2	512	208	84	16	94	35
C8	D1	191	98	67	8	93	38
	D2	455.67	191.67	77	14	93	3

Table 2: Grep Benchmark Results Summary

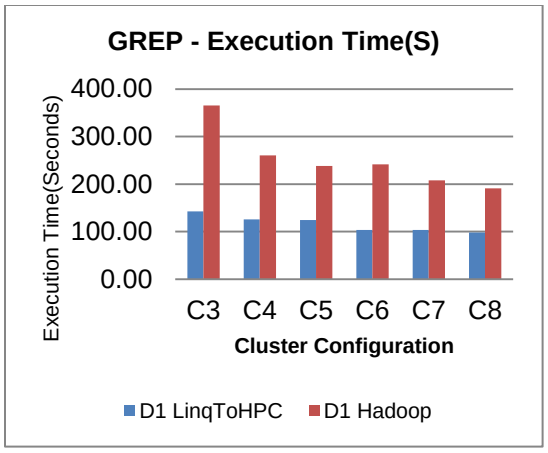

Figure 7: GREP Execution Time for D1

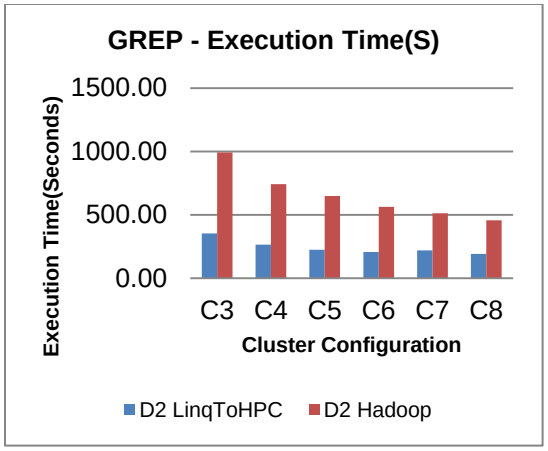

Figure 8: GREP Execution Time for D2

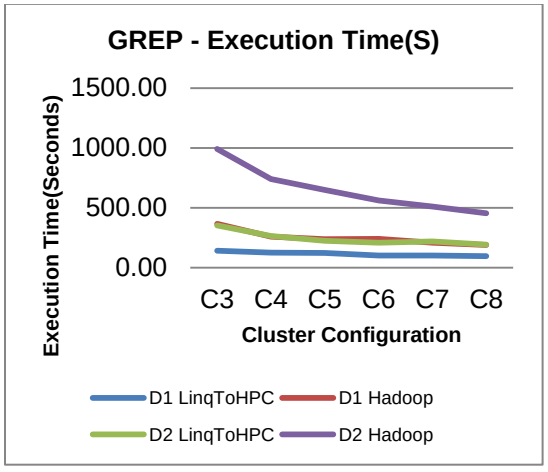

Figure 9: Grep Execution Time Line chart

Figures 7 and 8 provide a comparison of the execution times between Hadoop and LINQ to HPC for the Grep benchmark on the different clusters configuration for the two datasets. Results show that LINQ to HPC performed approximately two times better on the Grep benchmark on dataset D1 and 2.5 times better on dataset D2 on all cluster configurations. As the number of nodes in the cluster for Hadoop and LINQ to HPC increased, the run times were reduced for both datasets. The difference in average execution time for Hadoop and LINQ to HPC was statistically significant (p= 0.002) for both datasets.

Figure 9 suggests that as the cluster gets bigger the gap between LINQ to HPC and Hadoop is decreased from 60% to 50%. We expect Hadoop to catch up to LINQ to HPC's performance with larger clusters, as in more practically sized clusters.

Figures 10 and 11 provide a comparison of the average CPU usage between Hadoop and LINQ to HPC for the Grep benchmark on different cluster configurations on the two datasets. Hadoop's CPU usage was approximately three times higher when compared to LINQ to HPC for all cluster configurations and datasets. However, the average CPU usage decreased in Hadoop as more nodes were added. Interestingly, unlike Hadoop, LINQ to HPC's average CPU usage increased slightly when more nodes were added to the cluster as shown in Figure 10 below. The difference in average CPU usage for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

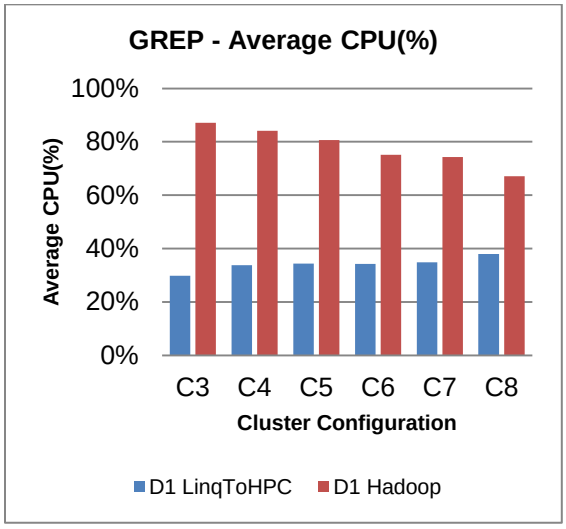

Figure 10: Grep Average CPU Usage for D1

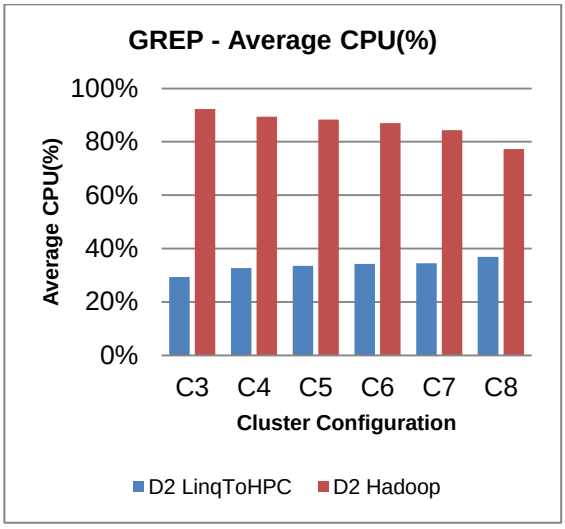

Figure 11: Grep Average CPU Usage for D2

Figures 12 and 13 provide a comparison of the percentage average memory usage between Hadoop and LINQ to HPC for the Grep benchmark on different cluster configurations for the two datasets. Results reveal that Hadoop used approximately five times more memory than LINQ to HPC for all cluster configurations and datasets. The average memory usage was found to be consistent in Hadoop and did not vary much with the increase in the number of nodes or data volume, which indicates Hadoop uses memory more effectively. On the other hand, the memory usage of LINQ to HPC decreased slightly as more nodes were added to the cluster. The difference in average memory usage for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

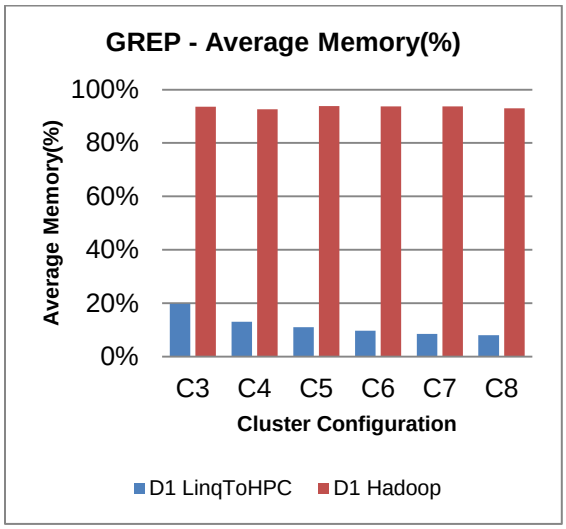

Figure 12: Grep Average Memory Usage for D1

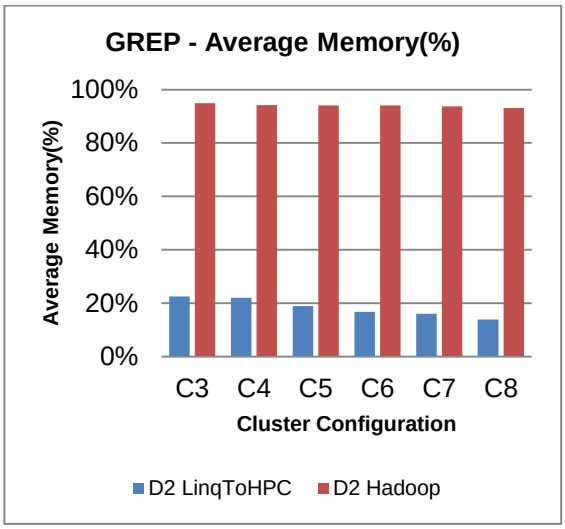

Figure 13: Grep Average Memory Usage for D2

5.2 Word Count Benchmark Results

The Word Count benchmark results are summarized in Table 3 for Hadoop and LINQ to HPC using the different cluster configurations (C3, C4, C5, C6, C7, and C8) and datasets (D1 and D2) for execution time, average CPU utilization, and average memory utilization. The values presented in the table represent the average value of three different experiment runs.

CONFIGURATION	DATASET	Execution Time (S)		Average CPU (%)		Average Memory (%)	
		HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC
C3	D1	1306	2156.67	96	33	95	26
	D2	3822.33	6479.33	98	31	95	28
C4	D1	991.33	1555.33	95	28	95	31
	D2	2836	4465	98	31	95	31
C5	D1	842.33	1298.33	95	29	95	32
	D2	2442.67	3377.67	97	32	95	31
C6	D1	725.67	1297	92	25	95	33
	D2	2018.33	3490.33	97	28	95	32
C7	D1	669.67	1349.67	89	21	94	33
	D2	1774.67	2889.67	96	30	95	32
C8	D1	621	1072.67	83	20	93	36
	D2	1630.67	2491	93	28	95	35

Table 3: Word Count Benchmark Results Summary

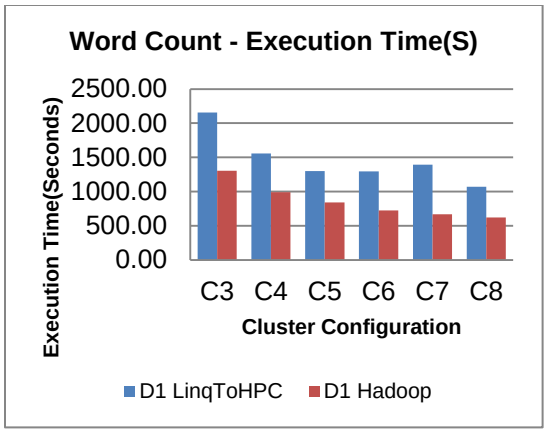

Figure 14: Word Count Execution Time for D1

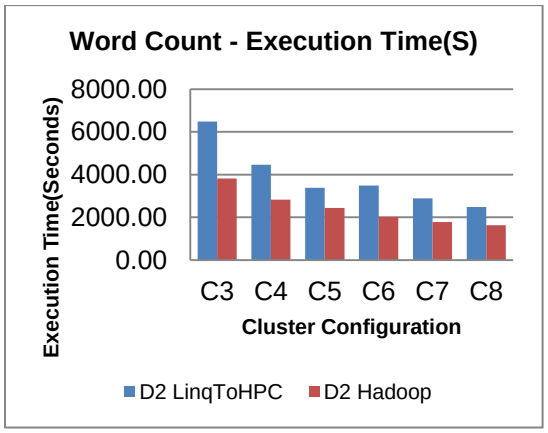

Figure 15: Word Count Execution Time for D2

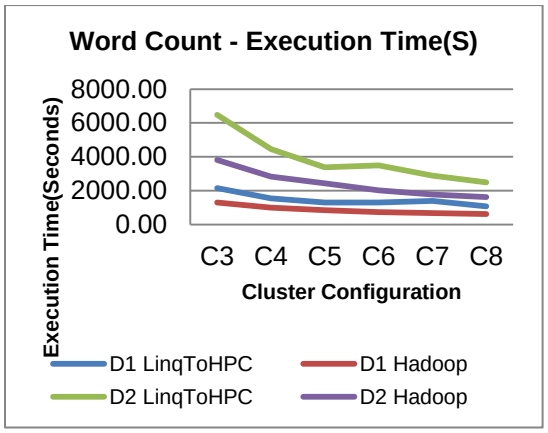

Figure 16: Word Count Execution Time Line Chart

Figures 14, 15, and 16 represent a comparison of execution times between Hadoop and LINQ to HPC for the Word Count benchmark. This benchmark is the most computational intensive task of the four benchmarks. Hadoop performed approximately 50% better on datasets D1 and D2 for all cluster configurations. For dataset D2 and on a three-node cluster the difference between Hadoop and LINQ to HPC execution times was greater than it was when the numbers of nodes were increased. Another interesting observation is that Hadoop splits the source files into smaller blocks when the data files were put into the Hadoop distributed file system. The data was distributed more evenly in Hadoop across the nodes compared to LINQ to HPC. The difference in the average execution time for Hadoop and LINQ to HPC was statistically significant ($p = 0.01$) for D1 and statistically insignificant ($p = 0.06$) for D2.

Figures 17 and 18 represent a comparison of the average CPU utilization between Hadoop and LINQ to HPC for the Word Count benchmark on different cluster configurations and datasets. The CPU usage of Hadoop was approximately three times more that of LINQ to HPC for all cluster configurations and datasets. The average CPU usage decreased in Hadoop as the number of nodes was increased. Unlike Hadoop, LINQ to HPC's average CPU usage increased slightly for bigger clusters. The difference in average CPU usage for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

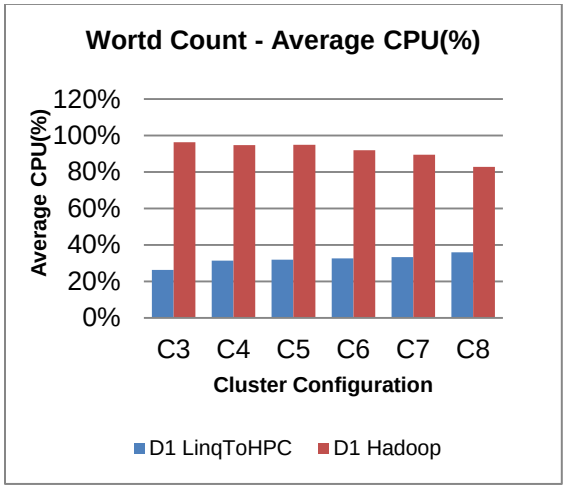

Figure 17: Word Count Average CPU Usage for D1

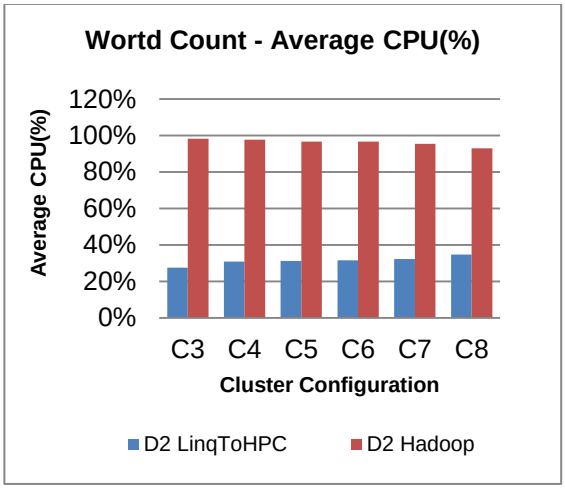

Figure 18: Word Count Average CPU usage for D2

Figures 19 and 20 represent a comparison of the average memory utilization between Hadoop and LINQ to HPC for the Word Count benchmark. Results show that Hadoop used, approximately, four times more memory than LINQ to HPC for all cluster configurations and dataset sizes. The average memory usage was found to be consistent in Hadoop and did not vary much in larger clusters or with increased data volumes. On the contrary, the memory usage of LINQ to HPC decreased with larger clusters. The difference in average memory usage for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

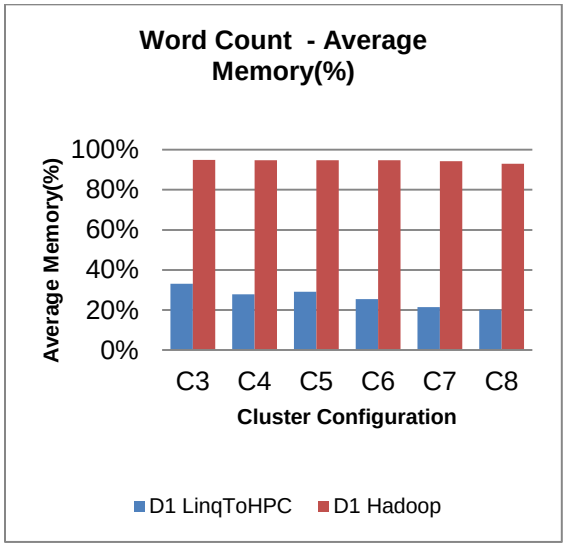

Figure 19: Word Count Average Memory Usage for D1

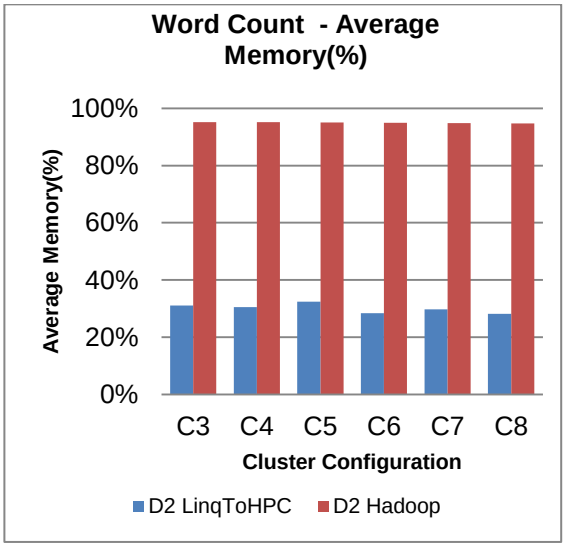

Figure 20: Word Count Average Memory Usage for D2

5.3 Read Benchmark Results

The Read benchmark results are summarized in the Table 4 for the Hadoop and LINQ to HPC for execution time, average CPU percentage utilization, and average memory percentage utilization. The results presented in the table represent the average value of three different runs.

CONFIGURATION	DATA SET	Execution Time (S)		Average CPU (%)		Average Memory (%)	
		HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC
C3	D1	5658	205.67	44	11	96	31
	D2	17019.67	619	44	12	97	30
C4	D1	5646	192.67	37	7	96	34
	D2	17039.33	598.33	36	13	97	33
C5	D1	5184.67	185.67	31	4	97	34
	D2	15388.67	610.67	31	5	97	35
C6	D1	5003	194.67	28	3	97	35
	D2	14480.67	604.33	27	5	97	35
C7	D1	4808.67	208.33	29	3	97	36
	D2	13785	596	25	4	96	36
C8	D1	4832	185	24	3	97	39
	D2	13724.67	639	23	3	97	39

Table 4: Read Benchmark Results Summary

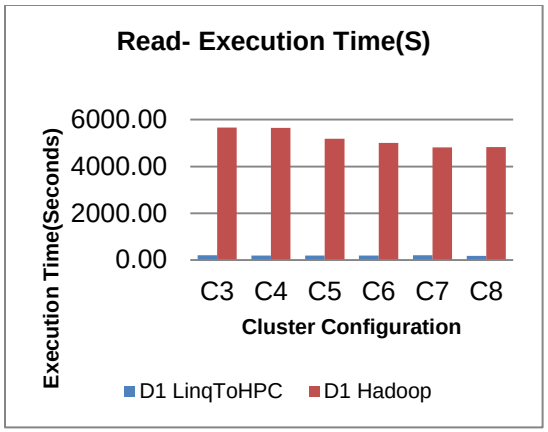

Figure 21: Read Execution Time for D1

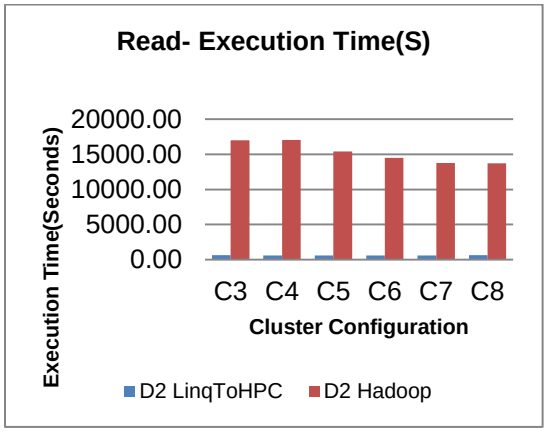

Figure 22: Read Execution Time for D2

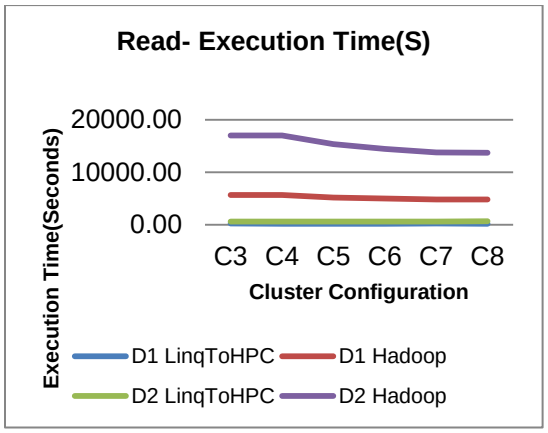

Figure 23: Read Execution Time Line chart

Figures 21, 22, and 23 represent a comparison of the execution time between Hadoop and LINQ to HPC for the Read benchmark on the different cluster configurations and datasets. Hadoop took approximately twenty five times more time to read the data from the distributed file system compared to LINQ to HPC.

The number of nodes in the cluster did not have an impact on the Read performance of LINQ to HPC. However, in Hadoop there was a slight improvement in the performance as the number of nodes was increased. The difference in average execution time for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

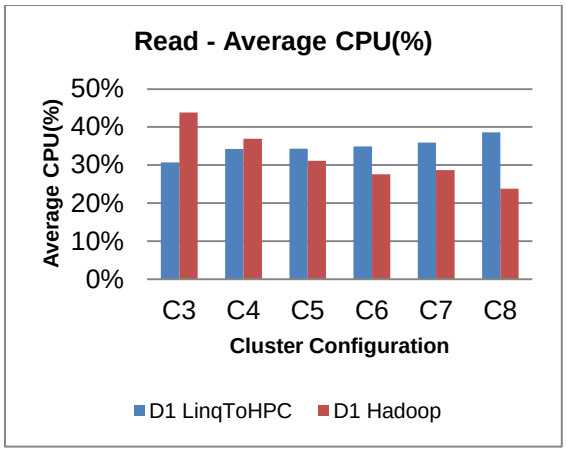

Figure 24: Read Average CPU Usage for D1

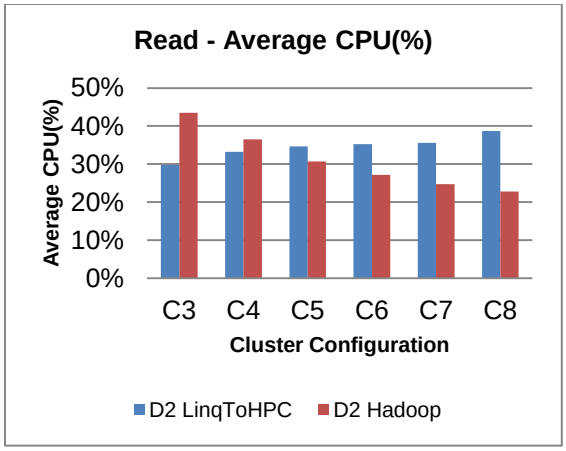

Figure 25: Read Average CPU Usage for D2

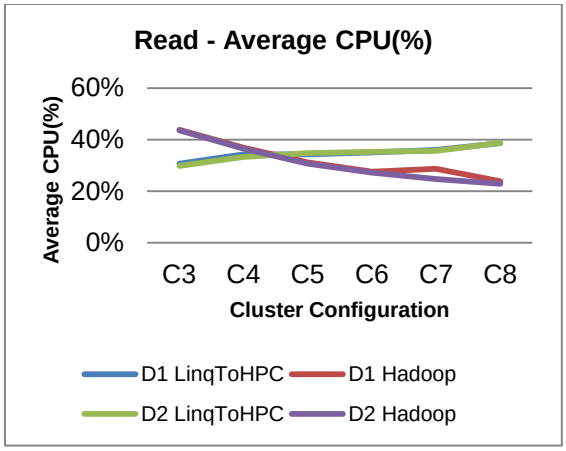

Figure 26: Read Average CPU Usage Line chart

Figures 24 and 25 represent a comparison of the average CPU usage between Hadoop and LINQ to HPC for the Read benchmark. Unlike Grep and Word Count benchmarks, the CPU usage for the Read benchmark followed an interesting pattern. The average CPU usage increased as the number of nodes increased in LINQ to HPC whereas it decreased as number of nodes increased in Hadoop.

There was not much insight on how CPU usage happens in LINQ to HPC through literature but we believe the observed CPU behavior in LINQ to HPC is attributed to the way data was distributed. The difference in the average CPU usage for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

Figures 27 and 28 represent a comparison of the average CPU usage between Hadoop and LINQ to HPC for the Read benchmark on different cluster configurations and different dataset sizes. Hadoop used approximately ten times more memory than LINQ to HPC for all cluster configurations. However, the Average memory usage was consistent in Hadoop and did not vary much with the increase in number of nodes or increase in data volume. On the other hand, LINQ to HPC memory usage decreased slightly as more nodes were added. The difference in the average memory usage for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

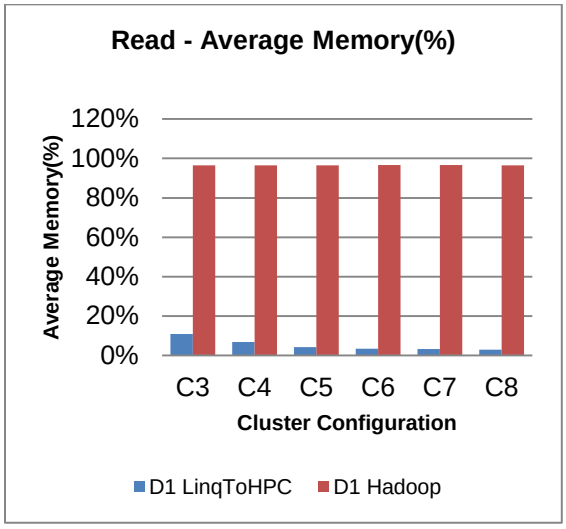

Figure 27: Read Average Memory Usage for D1

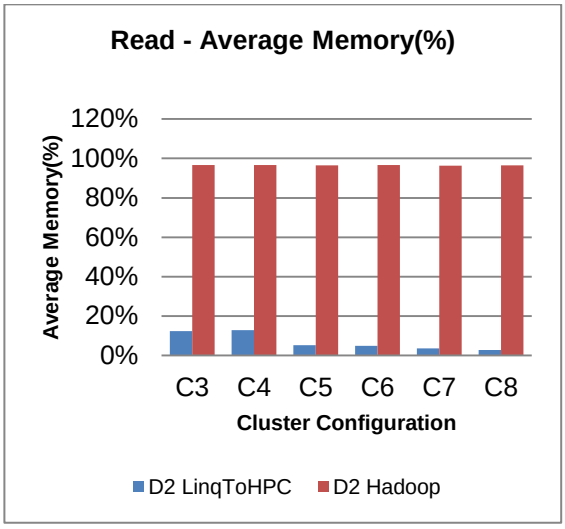

Figure 28: Read Average Memory Usage for D2

5.4 Write Benchmark Results

The Write benchmark results are summarized in Table 5 for the Hadoop and LINQ to HPC for the different cluster configurations (C3, C4, C5, C6, C7, and C8) and dataset sizes (D1 and D2) for execution time, average CPU percentage utilization, and average memory percentage utilization. The results presented in the table represent the average value of three different runs.

CONFIGURATION	DATA SET	Execution Time (S)		Average CPU (%)		Average Memory (%)	
		HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC	HADOOP	LINQ TO HPC
C3	D1	9931.67	407	78	18	86	30
	D2	29252	1269.33	79	18	94	30%
C4	D1	9450	305.67	64	17	85	32
	D2	29061	1057	64	18	92	32
C5	D1	8025.33	302.67	50	13	80	33
	D2	24701.67	1005.67	49	13	91	33
C6	D1	7446.33	320	42	11	77	34
	D2	22387	1030.67	42	12	90	34
C7	D1	7445.67	298	40	9	73	35
	D2	21553.33	1086.67	37	9	89	35
C8	D1	7326.33	279.67	35	8	71	38
	D2	21301.67	956	34	9	88	38

Table 5: Write Benchmark Result Summary

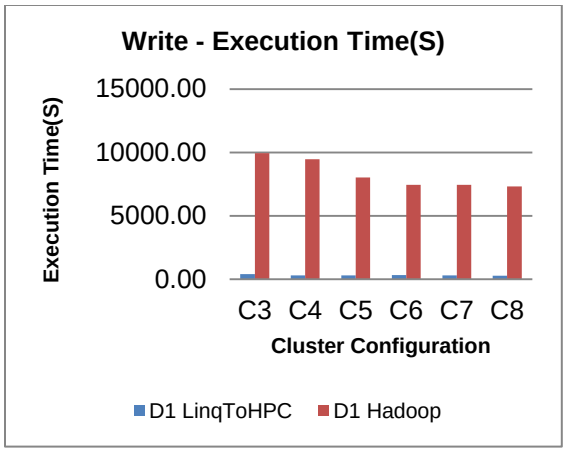

Figure 29: Write Execution Time for D1

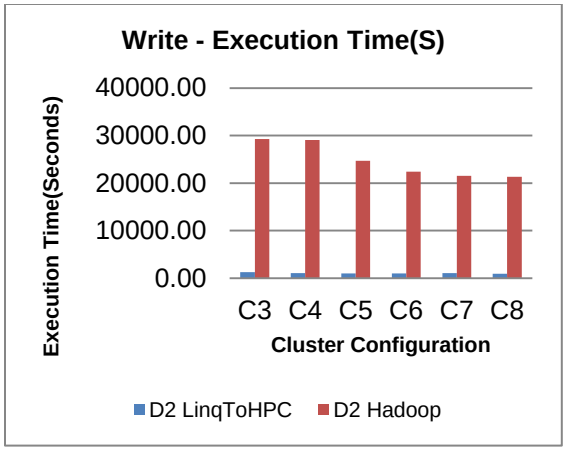

Figure 30: Write Execution Time for D2

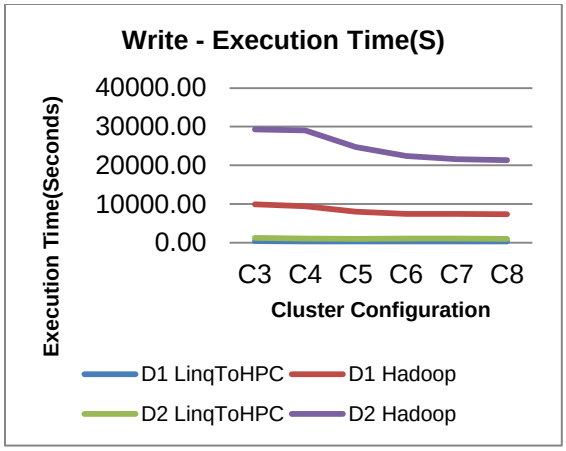

Figure 31: Write Execution Time Line Chart

Figures 29, 30, and 31 represent a comparison of the execution time between Hadoop and LINQ to HPC for the Write benchmark on different cluster configurations and datasets. Overall, LINQ to HPC performed better on the Write benchmark with datasets D1 and D2 for all cluster configurations.

Hadoop took approximately twenty five times more time to write the data from the local client to the distributed file system compared to LINQ to HPC. As the number of nodes increased, the Write benchmark performance improved on both LINQ to HPC and Hadoop. The difference in the average execution time for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

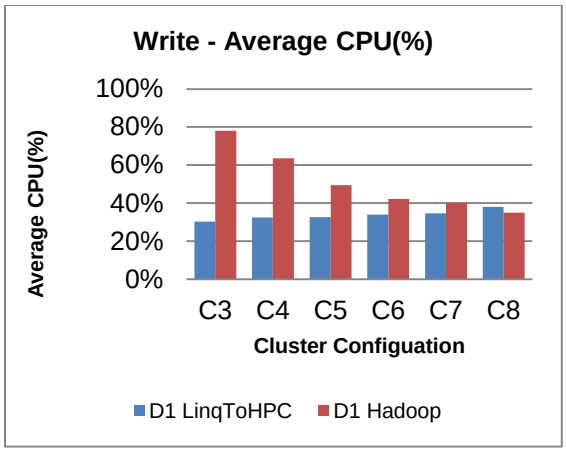

Figure 32: Write Average CPU for D1

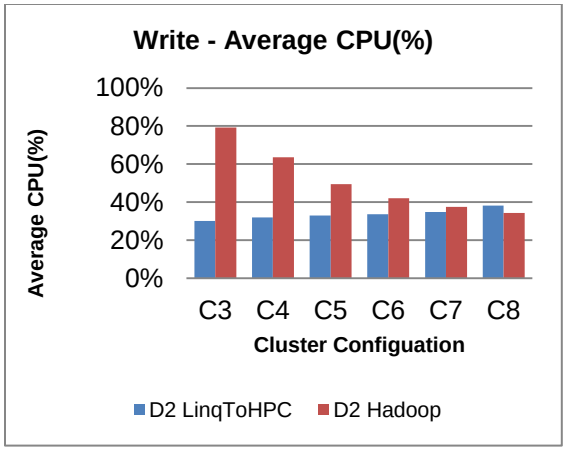

Figure 33: Write Average CPU for D2

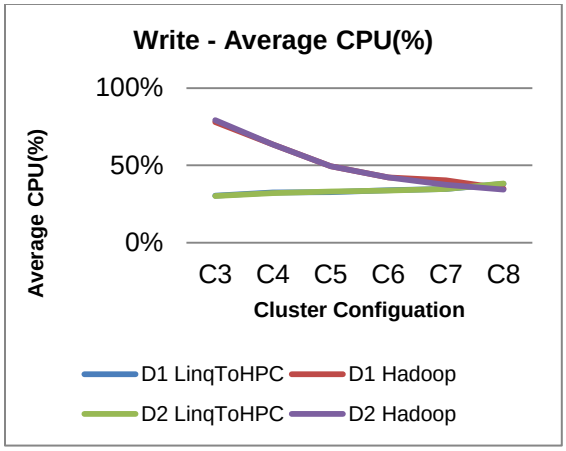

Figure 34: Write Average CPU Line Chart

Figures 32 and 33 represent a comparison of the average CPU usage between Hadoop and LINQ to HPC for the Write benchmark on the different cluster configurations and datasets. Unlike the Grep and Word Count benchmarks, the CPU usage for the Write benchmark follows an interesting pattern, similar to that of the Read benchmark.

The average CPU usage increased as the number of nodes increased in LINQ to HPC, whereas it decreased as the number of nodes increased in Hadoop. Here too, we believe the observed CPU behavior in LINQ to HPC is due to the way data was distributed. The difference in the average CPU usage for Hadoop and LINQ to HPC was statistically significant ($p <= 0.002$) for both datasets.

Figures 35 and 36 represent a comparison of the average memory utilization between Hadoop and LINQ to HPC for the Write benchmark on different cluster configurations and dataset sizes. Hadoop used approximately seven times more memory than LINQ to HPC on all cluster configurations. Although the average memory usage was consistent in Hadoop, i.e., it did not vary much with the increase in cluster size or data volume, LIN Q to HPC memory usage decreased slightly for larger clusters. The difference in the average memory usage for Hadoop and LINQ to HPC was statistically significant ($p < 0.001$) for both datasets.

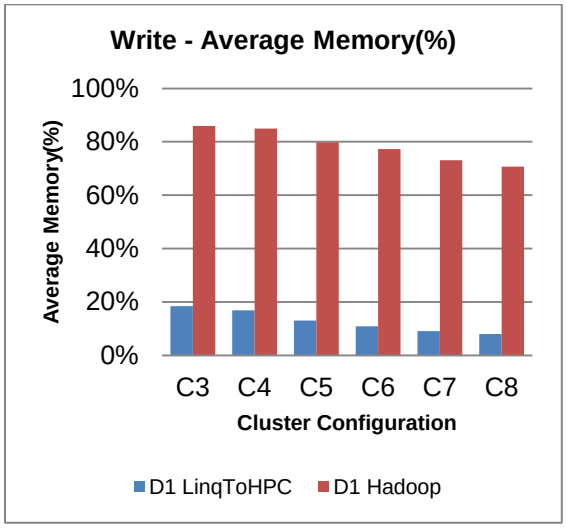

Figure 35: Write Average Memory for D1

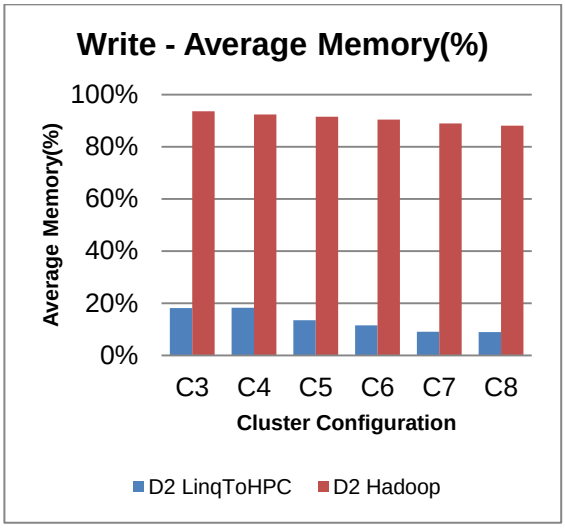

Figure 36: Write Average Memory for D2

5.5 Result Discussions Summary

This section provides a summary of the findings based on the detailed analysis performed in the previous sections. The result discussions are summarized in Table 6 for the Hadoop and LINQ to HPC.

BENCHAMRK	FIGURE	COMMENT
Grep	7, 8 and 9	LINQ to HPC performed approximately two times better than Hadoop for the Grep benchmark based on execution time. As the number of nodes increased, the gap between LINQ to HPC and Hadoop narrowed.
Grep	10 and 11	The average CPU usage was consistently about three times higher for Hadoop than LINQ to HPC
Grep	12 and 13	The average memory usage was also consistently higher for Hadoop compared to LINQ to HPC.
Word Count	14, 15 and 16	Hadoop performed 50% better than LINQ to HPC for the Word Count benchmark with respect to execution time. As the number of nodes was increased, the execution time reduced for both LINQ to HPC and Hadoop. When the number of nodes was increased the gap between LINQ to HPC and Hadoop narrowed.
Word Count	17 and 18	The average CPU usage was consistently three times higher for Hadoop compared to LINQ to HPC.
Word Count	19 and 20	The average memory usage was consistently higher for Hadoop.
Read	21, 22 and 23	LINQ to HPC performed better on the Read benchmark based on the execution time metric. Hadoop took, on average, twenty five times more time to read the data from the distributed file system compared to LINQ to HPC. . The number of cluster nodes did not have an impact on the Read performance of LINQ to HPC. In

		Hadoop, there was a slight performance improvement as the number of nodes was increased.
Read	24, 25 and 26	The average CPU usage increased as the number of nodes increased in LINQ to HPC whereas it decreased as the number of nodes increased in Hadoop.
Read	27 and 28	The average memory usage was consistently higher for Hadoop.
Write	29, 30 and 31	Hadoop took, on average, twenty-five times more time to write the data from the local client machine to the distributed file system compared to LINQ to HPC. As the number of nodes increased the Write benchmark, the execution time of LINQ to HPC and Hadoop improved.
Write	32, 33 and 34	The average CPU usage increased as the number of nodes increased in LINQ to HPC, whereas it decreased as the number of nodes increased in Hadoop.
Write	35 and 36	The average memory usage was consistent in Hadoop whereas LINQ to HPC memory usage decreased slightly for larger clusters.

Table 6: Result Discussions Summary

Chapter 6

CONCLUSION AND FUTURE WORK

Most organizations and enterprises are flooded with a deluge of data, typically referred to as Big Data. This data comes from traditional systems, sensors, mobile devices, cloud application and social media to name a few. IBM research claims that 2.5 quintillion bytes of data is created every day, so much that 90% of all data has been created in last two years only. IBM also claims that 80% of enterprise data is unstructured [BIGDATA12]. Traditionally, enterprises have been analyzing historical structured data only. With the availability of Big Data volumes, enterprises started to realize the significant opportunity and potential value of analyzing newer types of data to answer questions that were previously considered beyond their reach. Until recently, managing and analyzing Big Data were not practical because of the prohibitive cost, bad performance, and lack of tools and technical knowhow.

Hadoop is increasingly becoming the popular option to manage, process, and analyze huge volumes of unstructured data that comes from disparate data source. Hadoop has disrupted the enterprise data and analytics market with a scalable platform. Enterprises look at Hadoop as an extension to their existing IT environments to tackle the volume, velocity, and variety of Big Data. A number of companies like Cloudera, Horton Works, EMC, to name a few, are emerging to provide an enterprise grade Hadoop.

There are only few alternative platforms to Hadoop including Microsoft's LINQ to HPC, Lexis Nexis, IBM Pure Data (Netezza), Aster Data SQL-MR, and Green Plum Map Reduce. Microsoft's LINQ to HPC differentiates itself from the other platforms by enabling programmers to write high level queries based on Language Integrated Query (LINQ). The query-based programming model is simple, expressive and flexible than distributed computing frameworks, which require complex Map/Reduce pattern. Another key factor that differentiates LINQ to HPC from other platform is its ability to run on Windows HPC servers, which is widely used in enterprise environments.

Before this thesis, there was no performance analysis study to compare Hadoop and LINQ to HPC in an enterprise application environment. In addition, Hadoop is an open source system and LINQ to HPC is a proprietary system, which makes the comparison even more interesting for many organizations and researchers.

Experiments showed that LINQ to HPC performs better than Hadoop on three of the four-benchmark tasks (Grep, Read and Write) based on the execution time metric. Average Memory utilization of LINQ to HPC was better than Hadoop for all four benchmarks. The Average CPU utilization of LINQ to HPC was better than Hadoop for two of the four-benchmark tasks (Grep and Word Count). Hadoop was faster than LINQ to HPC on the Word Count benchmark, but the difference was not significant as the data size increased. On the I/O benchmarks (Read and Write) LINQ to HPC performed on an average three times better than Hadoop based on the execution time metric. On the Grep benchmark, LINQ to HPC performed, on an average, two times faster than Hadoop. As

the number of nodes were increased the gap between Hadoop and LINQ to HPC for the Grep and Word Count Benchmark results were getting closer, but the number of nodes did not significantly affect the I/O benchmark performance of Hadoop and LINQ to HPC. Hadoop processed data files to convert them into small blocks and distributed them effectively throughout the cluster, whereas LINQ to HPC stored them without processing and replicated them across the cluster. We believe this was the main reason why LINQ to HPC outperformed Hadoop in the Read and Write benchmark.

6.1 Future Work

Although Hadoop and its variants enjoy a much larger adoption in enterprise environments, our experiments indicate that LINQ to HPC performs better in most typical use scenarios, particularly for smaller implementations given the cluster sizes used in our experiments. Comparing LINQ to HPC and Hadoop in larger sized clusters and using terabytes of data will be of interest to larger organizations and scientific communities. In addition, in our experiments we used Windows HPC clusters for both Hadoop and LINQ to HPC for the obvious reason of neutralizing the effect of operating systems, yet most organizations that adopted Hadoop use Linux as the underlying operating system. It will be of interest to conduct further experiments using Hadoop on Linux clusters.

REFERENCES

Print Publications:

[Dean08]
Dean, Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” Communications of the ACM - 50th anniversary issue: 1958 - 2008, Volume 51 Issue 1, Pages 107-113, January 2008.

[Ekanayake09]
Ekanayake, Gunarathne, Fox, Balkir, Poulain, Araujo, Barga, “DryadLINQ for Scientific Analyses,” E-SCIENCE '09 Proceedings of the 2009 Fifth IEEE International Conference on e-Science, Conference Publications, Pages 329-336, 2009.

[Fadika11]
Pavlo, Dede, Govindaraju, Ramakrishnan, “Benchmarking MapReduce Implementations for Application Usage Scenarios,” GRID '11 Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing, Pages 90-97, 2011.

[Gonzalez11]
Gonzalez-Velez, Kontagora, “PERFORMANCE EVALUATION OF MAPREDUCE USING FULL VIRTUALISATION ON A DEPARTMENTAL CLOUD,” International Journal of Applied mathematics and Computer Science, Vol. 21, No. 2, 2011.

[Israd07]
Isard, Budiu, Yu, Birrell, Fetterly, "Distributed data-parallel computing using a high-level programming language", ACM SIGOPS Operating Systems Review - EuroSys'07 Conference Proceedings, EuroSys’07, Volume 41 Issue 3, Pages 59-72, June 2007.

[Israd09]
Isard, Yu, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language", SIGMOD '09 Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, Pages 987-994, 2009.

[Jiang10]
Jiang, Ooi, Shi, Wu, “The performance of MapReduce: an in-depth study,” Proceedings of the VLDB Endowment, Volume 3 Issue 1-2, Pages 472-483, September 2010.

[Pavlo09]
Pavlo, Paulson, Rasin, Abadi, Dewitt, Madon, Stonebraker, “A comparison of approaches to large-scale data analysis,” SIGMOD '09 Proceedings of the 35th SIGMOD international conference on Management of data, Pages 165-178, 2009.

Electronic Publications:

[BIGDATA12]
“Big Data”, http://www-01.ibm.com/software/data/bigdata/, last accessed September 7, 2012.

[Chappell11]
Chappell, “INTRODUCING LINQ TO HPC”, white paper, Microsoft Research, 2011.

[CLOUDERA12]
“Cloudera, CDH”, http://www.cloudera.com/hadoop/, last accessed September 7, 2012.

[CLODERA12A]
“Cloudera Installation Guide”, https://ccp.cloudera.com/display/CDHDOC/CDH3+Installation+Guide, last accessed September 7, 2012.

[CLODERA12B]
“Cloudera Quick Start Guide”, https://ccp.cloudera.com/display/CDH4DOC/CDH4+Quick+Start+Guide, last accessed September 7, 2012.

[DINH09]
“Hadoop Performance Evaluation, 2009". http://wr.informatik.uni-hamburg.de/_media/research/labs/2009/2009-12-tien_duc_dinh-evaluierung_von_hadoop-report.pdf, last accessed October 19, 2012.

[DSC12]
“Distributed Storage Catalog, DSC File Sets and Files”, http://msdn.microsoft.com/en-us/library/hh378120.aspx, last accessed September 7, 2012.

[FORRESTER12]
"The Forrester Wave: Enterprise Hadoop Solutions, Q1 2012". http://www.forrester.com/The+Forrester+Wave+Enterprise+Hadoop+Solutions+Q1+2012/fulltext/-/E-RES60755, last accessed October 19, 2012.

[HADOOP12]
“Hadoop”, http://hadoop.apache.org, last accessed September 7, 2012.

[HADOOP12A]
“Data Intensive Analytics with Hadoop: A Look Inside”,http://www-304.ibm.com/easyaccess/fileserve?contentid=217007, last accessed September 7, 2012.

[HDFS12]
“The Hadoop Distributed File System: Architecture and Design”, http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf, last accessed September 7, 2012.

[HDFS12A]
“Scalability of the Hadoop Distributed File System”, http://developer.yahoo.com/blogs/hadoop/posts/2010/05/scalability_of_the_hadoop_dist/, last accessed September 7, 2012.

[HDFS12B]
http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html, last accessed September 7, 2012.

[HPC12]
“Windows HPC Documentation”, http://technet.microsoft.com/en-us/library/cc972800(WS.10).aspx, last accessed September 7, 2012.

[HPCCMDLET12]
http://technet.microsoft.com/en-us/library/ff950195.aspx, last accessed September 7, 2012.

[LINQTOHPC12]
“Overview of LINQ to HPC and the Distributed Storage Catalog”, http://msdn.microsoft.com/en-us/library/hh378106.aspx, last accessed September 7, 2012.

[LINQTOHPC12A]
“LINQ TO HPC Programmer's Guide”, http://msdn.microsoft.com/en-us/library/hh378147, last accessed September 7, 2012.

[LINQTOHPC12B] LINQ to HPC SDK Sample Code, http://msdn.microsoft.com/en-us/library/hh696859, last accessed September 7, 2012.

Appendix A

Dataset Description

Table 7 provides the details regarding the data sets used for the experimentation.

Data Set No	Dataset Size (GB)	Dataset Files
D1	6.24	http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-0.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-1.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-2.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-3.csv.zip
D2	18.72	http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-0.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-1.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-2.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-3.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-4.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-5.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-6.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-7.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-8.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-9.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-10.csv.zip http://commondatastorage.googleapis.com/books/ngrams/books/googlebooks-eng-all-2gram-20090715-11.csv.zip

Table 7: Dataset Description