CSC 591 Data Intensive Computing

Final Project Report of

Comparative Study of HPCC and HADOOP based on PUMA Benchmarks

Submitted on: 1 May 2015

Submitted By:
Group #2

Team members:
Haiyu Yao (hyao4)
Nakul Shukla (nshukla)
Vishal Mishra (vmishra)

Abstract**:** *PUMA is a benchmark suite for Hadoop. We proposed to write its test apps in Enterprise Control Language (ECL) for HPCC and run a comparison study with Hadoop. MapReduce is a well-known programming model, developed within Google, for processing large amounts of raw data for example, crawled documents or web request logs. HPCC Systems from LexisNexis Risk Solutions offers a data-intensive supercomputing platform designed for the enterprise to process and deliver Big Data analytical solutions. HPCC Systems offers a consistent data-centric programming language, two processing platforms and a single architecture for efficient processing.*

Introduction

Data-intensive is used to describe applications that are I/O bound or with a need to process large volumes of data (Gokhale, Cohen, Yoo, & Miller, 2008; Gorton et al., 2008; Johnston, 1998). Such applications devote most of their processing time to I/O and movement of data.The fundamental challenges of data-intensive computing are managing and processing exponentially growing data volumes, significantly reducing associated data analysis cycles to support practical, timely applications, and developing new algorithms which can scale to search and process massive amounts of data. This data is usually so large that it must be distributed across thousands of machines in order to be processed in a reasonable time.

PUMA benchmark suite represents a broad range of MapReduce applications exhibiting application characteristics with high/low computation and high/low shuffle volumes. There are a total of 13 benchmarks, out of which *Tera-Sort*, *Word-Count*, and *Grep* are from Hadoop distribution.The rest of the benchmarks were developed in-house at Purdue University and are currently not part of the Hadoop distribution. The three benchmarks from Hadoop distribution are also slightly modified to take number of reduce tasks as input from the user and generate final time completion statistics of jobs.The ease of programmability, automatic data management and transparent fault tolerance has made MapReduce a favorable choice for large-scale data centers batch processing.

Related Work

On August 8, 2009 a Terabyte Sort benchmark test was conducted on a development configuration located at LexisNexis Risk Solutions offices in Boca Raton, FL in conjunction with and verified by Lawrence Livermore National Labs (LLNL). The test cluster included 400 processing nodes each with two local 300MB SCSI disk drives, Intel Xeon single core processors running at 3.00 GHz, 4GB memory per node, all connected to a single Gigabit ethernet switch with 1.4 Terabytes/sec throughput. Hadoop Release 0.19 was deployed to the

cluster and the standard Terasort benchmark written in Java included with the release was used for the benchmark. Hadoop required 6 minutes 45 seconds to create the test data, and the Terasort benchmark required a total of 25 minutes 28 seconds to complete the sorting test. The HPCC system software deployed to the same platform and using standard ECL required 2 minutes and 35 seconds to create the test data, and a total of 6 minutes and 27 seconds to complete the sorting test[11].

Hadoop Programming Model (MapReduce)

Hadoop is a framework for distributed storage and processing of large data sets on clusters.The storage part is the Hadoop Distributed File System and MapReduce performs the requisite processing. A MapReduce splits the input data sets into smaller independent chunks which are processed in a parallel configuration. The framework sorts the output of the maps which is then fed as input to the reduce tasks.

The map function is applied to the local data and the worker node writes that data into temporary storage. Only one out of the redundant copies of data is processed further. In the shuffle part, data belonging to a particular key is located on the same worker node.Data pertaining to each key is then processed in parallel in the reduce step. The input to a Map job is viewed as a set of <key,value> pairs and the output is presented as a list of <key,value> pairs in different domains. The input to a Reduce job is viewed as a set of <key, list<value>> pairs and the output is presented as a list of <key,value> pairs in different domains. The key and value here can be different thing and in different types among input and output.

In Java programming, the Map and Reduce are defined as two interfaces, so the basic work of coding a Hadoop application is just to implement these two interfaces. That is to say, a Hadoop application developer has to think all problems in the MapReduce paradigm. On the one hand, once a developer understand this paradigm, he does not need to learn anything else and can leverage all his Java knowledge to begin coding. On the other hand, not all problems can be adapted to MapReduce paradigm, especially for those tasks involving lots of dependencies between lines.

ECL and ECL Programming Model

ECL (Enterprise Control Language) is the programing language used in HPCC Thor and HPCC Roxie nodes. Before ECL program is executed, it is compiled into C++ code and then is compiled into optimized native machine code. All the benchmark scripts we run on HPCC

cluster are implemented in ECL, so it is necessary to explain some about ECL programming model.

ECL is a declarative language. Programmers only use the language to tell computer what needs to be done rather than how to do that task. There are two types of ECL code: Definition and executable Actions. According to <HPCC ECL Language Reference>, the declarative characteristic of ECL are reflected by one of the two types: Definition. Definition is the code to only specify what need to do and they do not actually execute until it is called by Actions. Generally speaking, most of ECL code is definition code. In our benchmark scripts, we often only have 1 or 2 line of action code out of around 100 lines.

Moreover, the order of definition does not matter during the running phase. The definitions are compiled and optimized when an action using it executes, so it means the execute order is not determined until this point. This concept is called as "order-less execution" in <HPCC ECL Language Reference>.

Although the declarative characteristic and the concept of "orderless execution" requires us to think in a different mindset, they did not bring us too much trouble to switch to ECL programming model. If we think of Action as a main function and Definition as other functions invoked by the main function, what we knew still works in ECL programming.

However, the most challenging change was to switch from a traditional programming mindset to a dataset manipulation mindset. In ECL, everything we code works on a dataset as a whole. The most frequently used keywords are PROJECT and TRANSFORM. We need to think almost all programing as doing some data transformation. To some extent, it is very similar to the way you think when you use SQL.

Environment Preparation

To reduce the impact of environment on our testing result, we use the same cluster as the testbed. The cluster consists of 9 virtual machine nodes (1 master + 9 slaves). The configuration of the cluster is listed below:

- Main Memory: 2 GB per node (usually 1 GB free)
- CPU Cores: 2 per node (Intel(R) Xeon(R) E5645 @ 2.40GHz)
- Storage: 30 GB per node
- Operating System: Ubuntu Linux 14.04 Base

Hadoop supports three types of installations: local (Standalone) mode, pseudo-distributed mode, and fully-distributed mode. Fully-distributed mode is the standard mode used in the production

environment. It involves all components of Hadoop: MapReduce, HDFS and Yarn. Since we want to make the result as close as possible to actual production scenarios, we selected this mode for our Hadoop testbed. The major configuration of our Hadoop cluster is listed below:

- dfs.blocksize: 128 MB (64 MB for Adjacency-List)
- dfs.replication: 1
- mapreduce.map.java.opts: 512 MB
- mapreduce.reduce.java.opts: 768 MB
- mapreduce.task.io.sort.mb: 256 MB

The difference of dfs.blocksize between Adjacency-List's and other scripts is because Adjacency-List runs into memory shortage if we set dfs.blocksize to 128 MB. Even using 64 MB did not prevent memory shortage issue for 30 GB data testing. We tested dfs.replication = 1 and dfs.replication = 2 on our cluster using word-count scripts, and we found dfs.replication = 1 gave us better performance. The object of the project is to compare the best performance between Hadoop and HPCC, so we set dfs.replication to 1. The settings of mapreduce.map.java.opts and mapreduce.reduce.java.opts are determined via results from multiple experiments. When we used bigger value, the cluster ran into physical memory shortage. When we use smaller values, the cluster reports insufficient heap size error.

HPCC also supports similar modes: single-node mode and multi-node mode. Due to similar reasons as Hadoop's, we used multi-node mode as the testbed. To set up HPCC running in single-node mode, the only thing needed was to install a deb/rpm package on Linux. Additional work for setting up a HPCC cluster in multi-node mode is to set the node number and the responsibility of those nodes on a GUI provided by HPCC configuration manager. The major configuration of our Hadoop cluster is listed below:

- 2 thor slave processes per node
- no roxie
- replication factor: 2

We set 2 thor slave processes per node to fully use the computing resource. We did not use roxie, because the Thor of HPCC is a counterpart to Hadoop. Introducing roxie into HPCC would not be a fair comparison to Hadoop. The default value of replication factor in HPCC is 2. We did not find a way to change it, so we went ahead with the default value.

Approach of Running Benchmarks

To ensure our testing result as trustworthy as possible, we used the following approaches:

- We ensured Hadoop and HPCC have same input and generate same output so that neither of them will do more work than the other does.
- Hadoop and HPCC shared same cluster environment and only either of them would be started up during benchmark test, so the resource used by them should be similar.
- We consecutively ran same benchmark task for Hadoop and HPCC to reduce the fluctuation of resource due to the nature of virtual machine cluster (other users' activities might affect the resource we can use).
- We executed each benchmark task at least twice so that we can detect abnormal poor performance due to the nature of virtual machine cluster
- We tested Hadoop and HPCC on different size of data so that we can check their scalability.

Studied PUMA Benchmark in MapReduce and in ECL

This section intends to describe the test applications of PUMA benchmark we have studied:

Word-Count

It is a test application counting the occurrence of each distinct word in all input files. The input files can be text files in any format. The script splits each line into words using some kinds of delimiters such as space(\s) and tab(\t) and then count them. After counting, the script generates an output file in which each line is in the format of "<word> <count>".

In Hadoop, when a job is submitted, all files in the input directory will be fed to Map by the Hadoop framework. And then, Map splits lines into words and output a tuple <word, 1> for each word. Reduce take tuples <word, list<Integer> (a bunch of "1")> as input, and sums all those 1's to emit the result tuple <word, count>.

In ECL, the first thing to do is let ECL read all input files. We did not find some concept like directory in HPCC DFS. Although we can code all file names explicitly, this is impractical for real development. Fortunately, HPCC offers a concept called SuperFile. In HPCC DFS, all files in the same format can be bundled as a SuperFile which can be seen as a logical combination of all those concrete files. ECL code only needs to read that SuperFile to access the content of all input files as a "Dataset". The rest of the work is just to define a TRANSFORM "attribute" (a ECL term which be thought of as a user-defined function) to split lines into words and then make use of TABLE function to count the occurrence of each word.

Inverted-Index

According to Ahmad said, this test application "takes a list of documents as input and generates word-to-document indexing". The output are <word, file name> tuples after removing duplicates."

In Hadoop, Map emits <word, file name> tuples with each word emitted once per docId. Reduce combines all tuples on the key <word> and emits <word,file name> tuples after removing duplicates.

In ECL, we also create a SuperFile as what we did in Word-Count, but it is not used for access the content. We used the standard library function "STD.File.SuperFileContents" with "NOTHOR" function to get the names of all the files included in the SuperFile. And then, use the filenames to access all individual file to create inverted index in a similar way as Hadoop's.

Self-Join

Self Join generates associations among k+1 fields given the set of k-field associations. In the existing Hadoop implementation, Map receives candidate lists of the form {element1, element2, ..., elementk}, each list in alphanumerically sorted order. Map breaks these lists into <{element(1), element(2), ...,element(k-1)}, {element(k)}> tuples.

Reduce prepares a sorted list of all the Map values for a given key by building <{element1, element2, ..., elementk-1}, {val1,val2, ..., valj}> tuples. From these tuples, (k+1)-sized candidates can be obtained by appending consecutive pairs of map values vali, vali+1 to the (k-1)-sized key. By avoiding repetition of (k-1)-sized keys for every pair of values in the list, tuples are a compact representation of the (k+1)-sized candidates set.

In ECL, we still create a SuperFile to include all files need to process. we use "DEDUP" with "SORT" to remove duplicate lines in the superfile firstly. We split each of the deduped lines into two parts: the last word (say n_{th} word) and the rest (say $n\text{-}1_{th}$ words). And then we use "AGGREGATE" function to merge the n_{th} word of those lines having same $n\text{-}1_{th}$ words. At last step, we eliminate lines if there is only one line having the $n\text{-}1_{th}$ words.

Adjacency List

It generates adjacency and reverse adjacency lists of nodes of a graph for use by PageRank-like algorithms. Map receives as inputs graph edges <p, q> of a directed graph that follows the power

law of the World-wide Web. The Hadoop implementation assumes that the probability a node has an out-degree of i, is proportional to $1/i^{skew}$ with an average out-degree of 7.2.

Map emits tuples of the form <q, from_list{p}:to_list{}> and <p, from_list{}:to_list{q}>. For a given key, Reduce generates unions of the respective lists in the from_list and to_list fields, sorts the items within the union lists, and emits <p(and q), from_list{sorted union of all individual from_list}:to_list{sorted union of all individual to_list}> tuples.

In ECL, we still create a SuperFile to include all files need to process. We used "AGGREGATE" function twice to scan the superfile and generate <q, from_list{p}:to_list{}> and <p, from_list{}:to_list{q}>. And then we use "JOIN" function to join the two intermediate files based on the key and output <p(same as q), from_list{union of all individual from_list}:to_list{union of all individual to_list}>.

K-Means

K-means iterates to successively improve the clustering. It classifies movies based on their ratings using anonymized movies rating data which is of the form <movie_id: list{rater_id, rating}>. Random starting values are used for the cluster centroids.

Map computes the cosine-vector similarity of a given movie with the centroids, and determines the centroid to which the movie is closest (i.e., the cluster to which it belongs). Map emits <centroid_id, (similarity_value,movie_rating)> . Reduce determines the new centroids by computing the average of similarity of all the movies in a cluster. The movie closest to the average is the new centroid and Reduce emits movies data along with their current centroid as well as new centroids data (model file) for use in the next iteration. The algorithm iterates until the change in the centroids is below a threshold.

In ECL, we have to scan the data three times to transform <movie_id: list{rater_id, rating}> to <movie_id, rating1, rating2, … rating10>. The reason for do this costly transformation is that HPCC cannot handle variable field records (The element number in list{rater_id, rating} is variable.).

ECL contains a standard library to do K-means clustering (ECL-ML Machine Learning Module). We leverage the library and produce results that are similar to Hadoop results using Allegiance function from the library. As mentioned in previous scripts, we input the data from a "SuperFile" and apply TRANSFORM functions to get the data ready in the proper format for the application of K-Means on it.

For sake of simplicity and reducing the time it takes for scripts to run in ECL, we only used the first 10 ratings of each movie in both ECL and Hadoop implementations, though Hadoop was capable of handling more ratings in a reasonable amount of time.

Histogram - Movies

The script generates a histogram of input data. We use the movie rating data and the input is of the form <movie_id: list{rater_id, rating}>. Based on the average ratings of movies (ratings range from 1 to 5) we bin the movies into 8 bins each with a range of 0.5.

Map computes the average rating for a movie, determines the bin, and emits <bin_value, 1> tuples. Reduce collects all the tuples for a bin and outputs a <bin_value, n> tuple.

The data input and preparation in ECL for this script is similar to K-Means as the data input is the same. However, in this script we take all the ratings associated with a movie and not just the 10 we took in K-Means. We input data from a "SuperFile" and run TRANSFORM functions to get it in the proper format. Then we apply an ECL standard AVERAGE function on the data to obtain the appropriate bins.

Results

The table below shows a summary of the results of the benchmark test. Hadoop's Adjancecy-List script always failed to process 30 GB data, so its result displays as "N/A". The 30GB Histogram test was not run because of the excessive amount of time it would take as our past experiments predicted. 10 GB data a little over 3 hours to complete and the result already showed HPCC's deficiencies on the task. We estimate a time for 30GB based on linear scalability which we feel is a valid assumption based on running the script on smaller datasets and analyzing the times.

Unit: hh:mm:ss

	10 GB		20 GB		30 GB	
	Hadoop	HPCC	Hadoop	HPCC	Hadoop	HPCC
Word Count	7:42	2:03	21:25	4:05	29:07	29:53
Inverted Index	10:01	6:19	21:49	10:52	27:42	16:22
Adjacency-List	19:46	11:58	44:09	30:20	N/A	51:56
Self-Join	4:58	5:46	11:20	13:52	16:01	21:28

	1 GB		10 GB		30 GB	
	Hadoop	HPCC	Hadoop	HPCC	Hadoop	HPCC
K-Means	00:36	2:14	3:16	13:45	8:17	34:19
Histogram	00:36	19:56	2:59	3:02:57	7:27	9:08:51 (Est.)

The rest of the section will illustrate six histogram charts to compare each script's results between Hadoop and HPCC and one extra histogram chart to compare the results between K-Means and Histogram, because they used same input but showed very different results. In the histograms charts, the Y-axis represents the time taken for processing the data in seconds and the X-axis represents the size of the data that was executed. The comparison in the section will be objective and we will show our opinion of these results in the Concluding Remarks section.

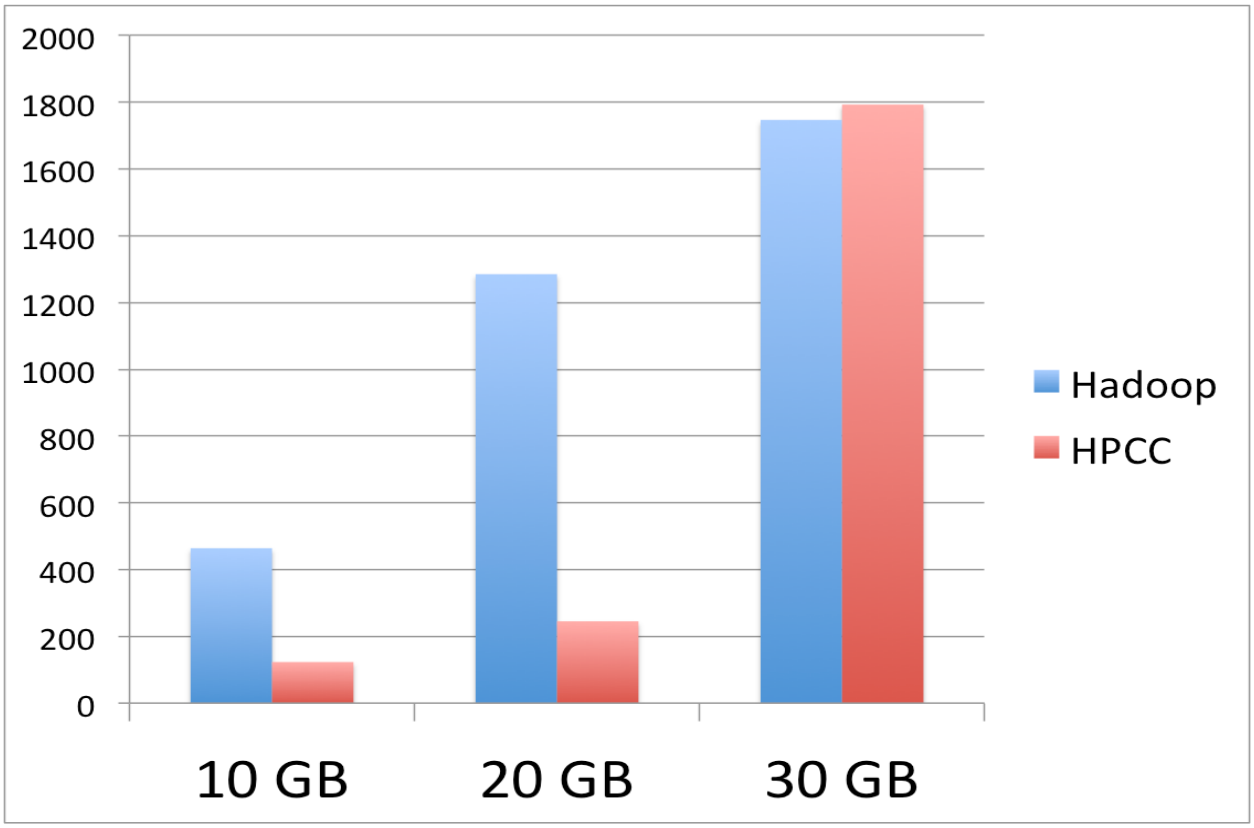

Figure 1: Word Count

The figure above compares the performance of Hadoop vs HPCC for processing the Word Count benchmark. HPCC was found to be faster than Hadoop for 10 GB and 20 GB of data. However, Hadoop was found to be slightly faster than HPCC for processing 30 GB of data. Within 20 GB, HPCC looks linear scalable. Hadoop's performance seems not linearly correlate with data size, but this might be caused by fluctuation of resource of virtual machine cluster.

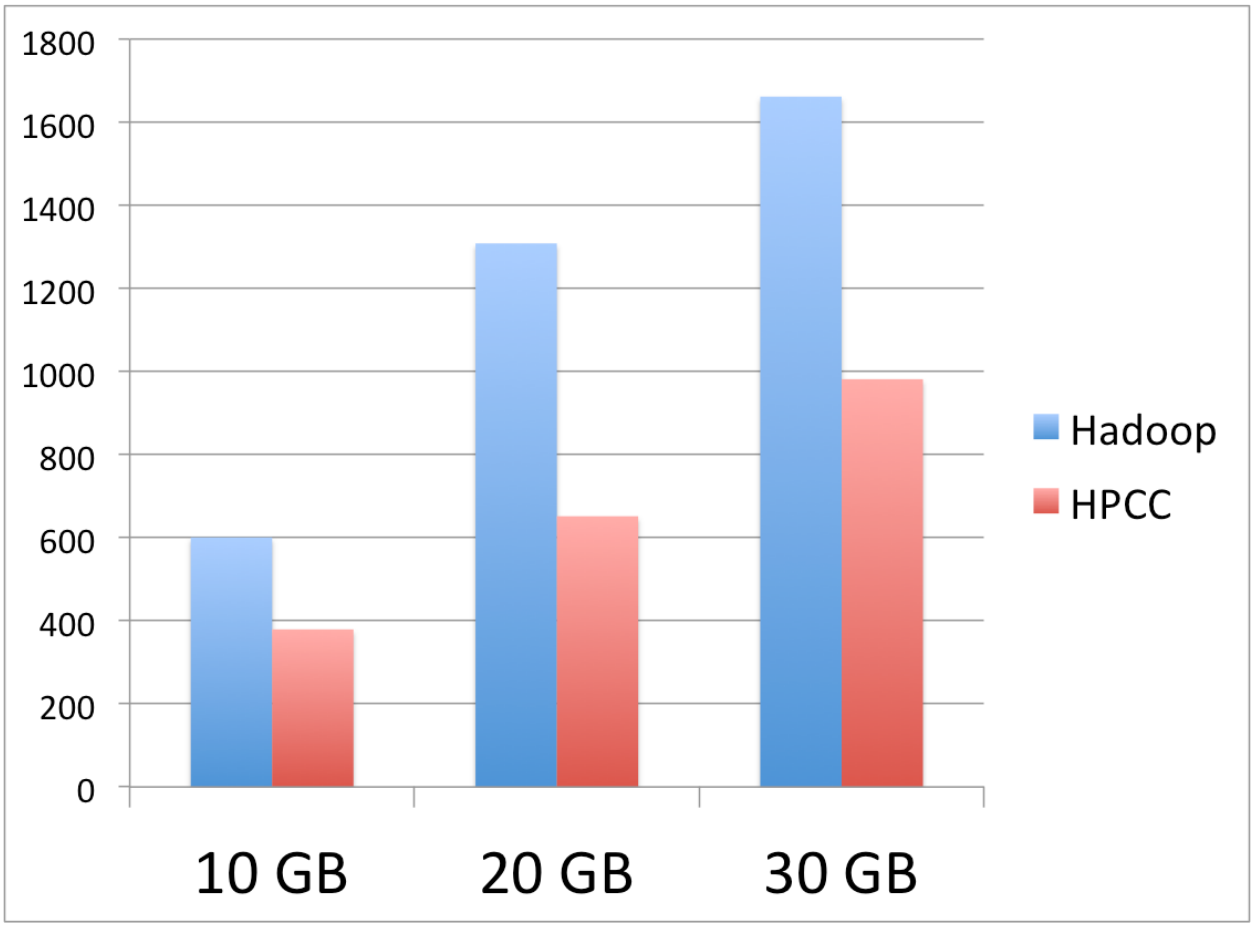

Figure 2: Inverted Index

The figure above compares the performance of Hadoop vs HPCC for processing the Inverted Index benchmark. HPCC was found to be faster than Hadoop. Both of them seem linear scalable within 30 GB.

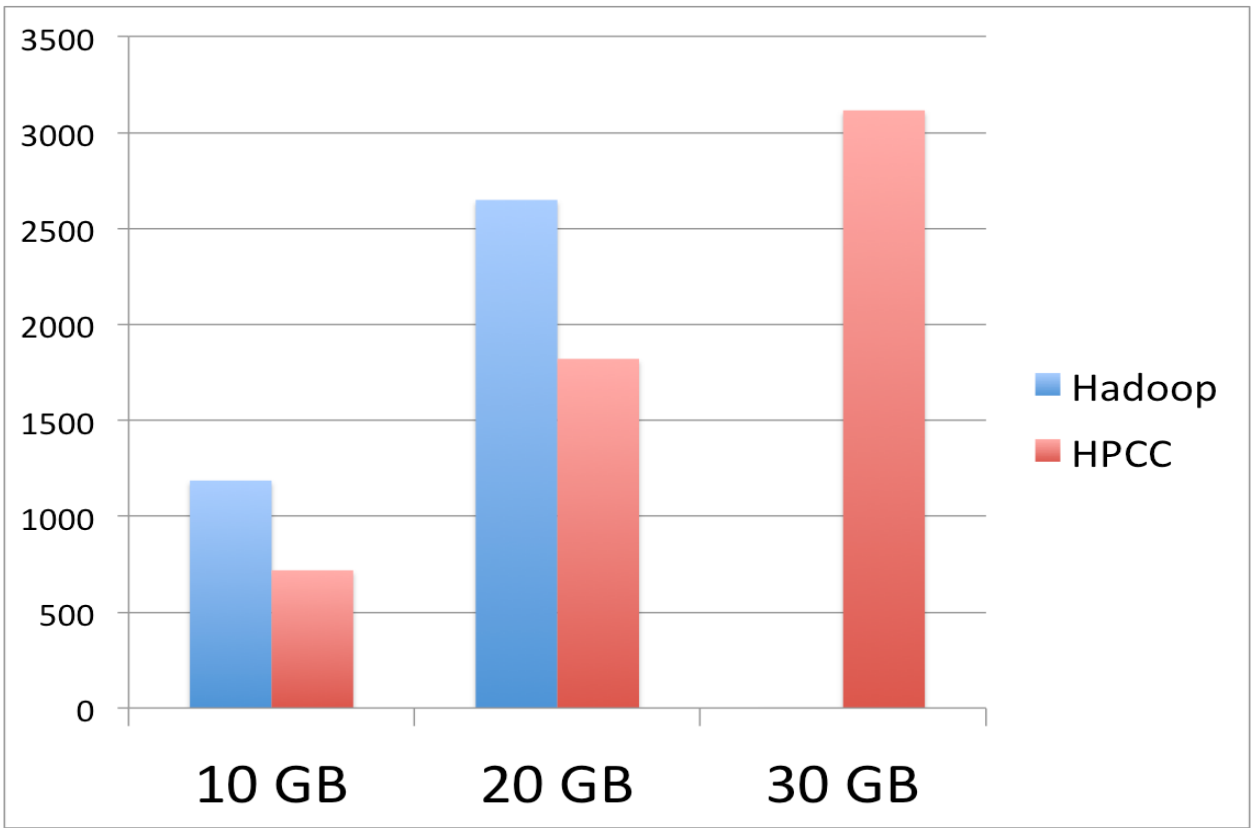

Figure 3: Adjacency-List

The figure above compares the performance of Hadoop vs HPCC for processing the Adjacency-List benchmark. HPCC was found to be faster than Hadoop for 10 GB and 20 GB of data. However, Hadoop failed to execute the processing of 30 GB of data for the Adjacency-List. Both of them seem linearly scalable within 30 GB when they perform normally.

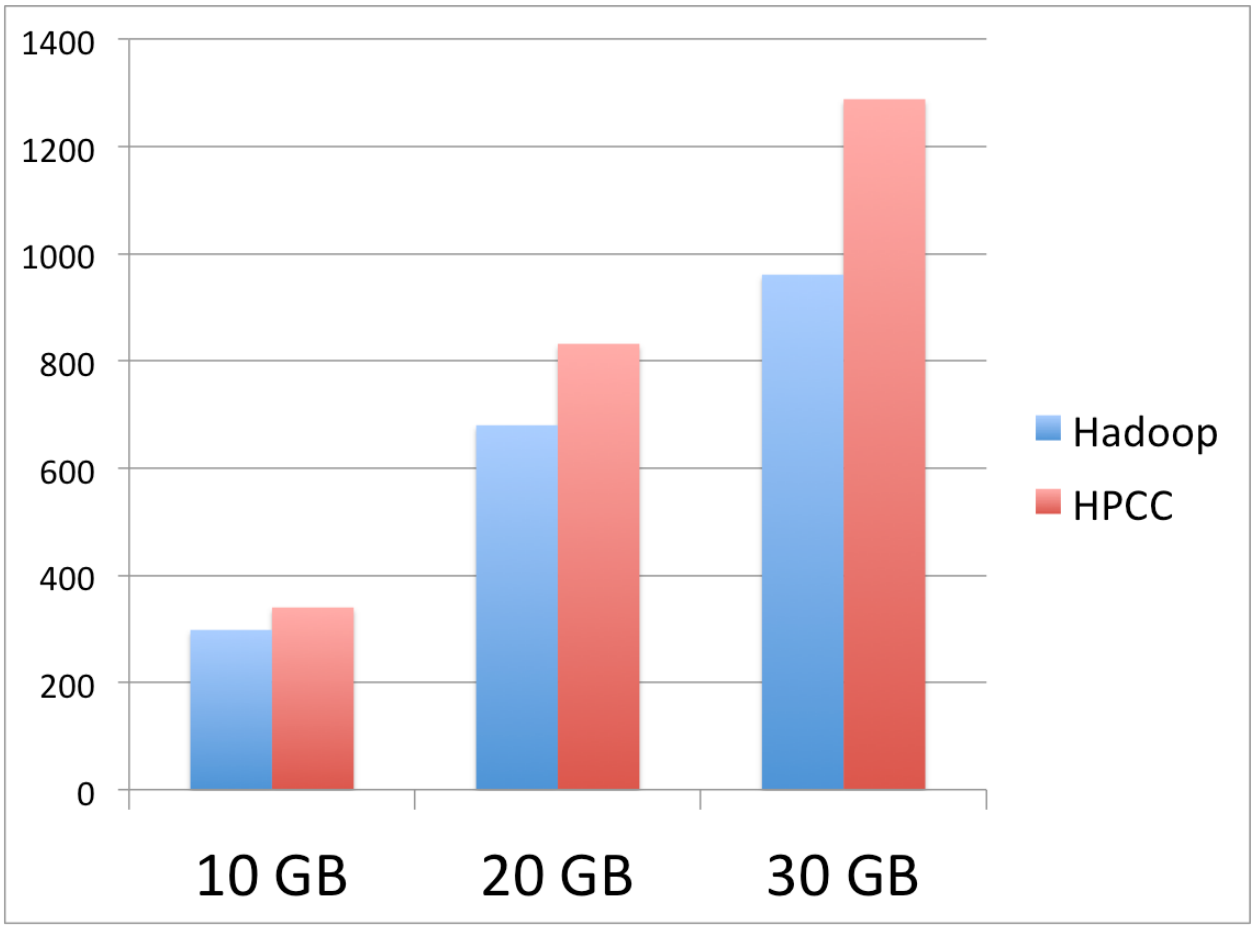

Figure 4: Self-Join

The figure above compares the performance of Hadoop vs HPCC for processing the Self-Join benchmark. Hadoop was found to be faster than Hadoop. Both of them look linearly scalable.

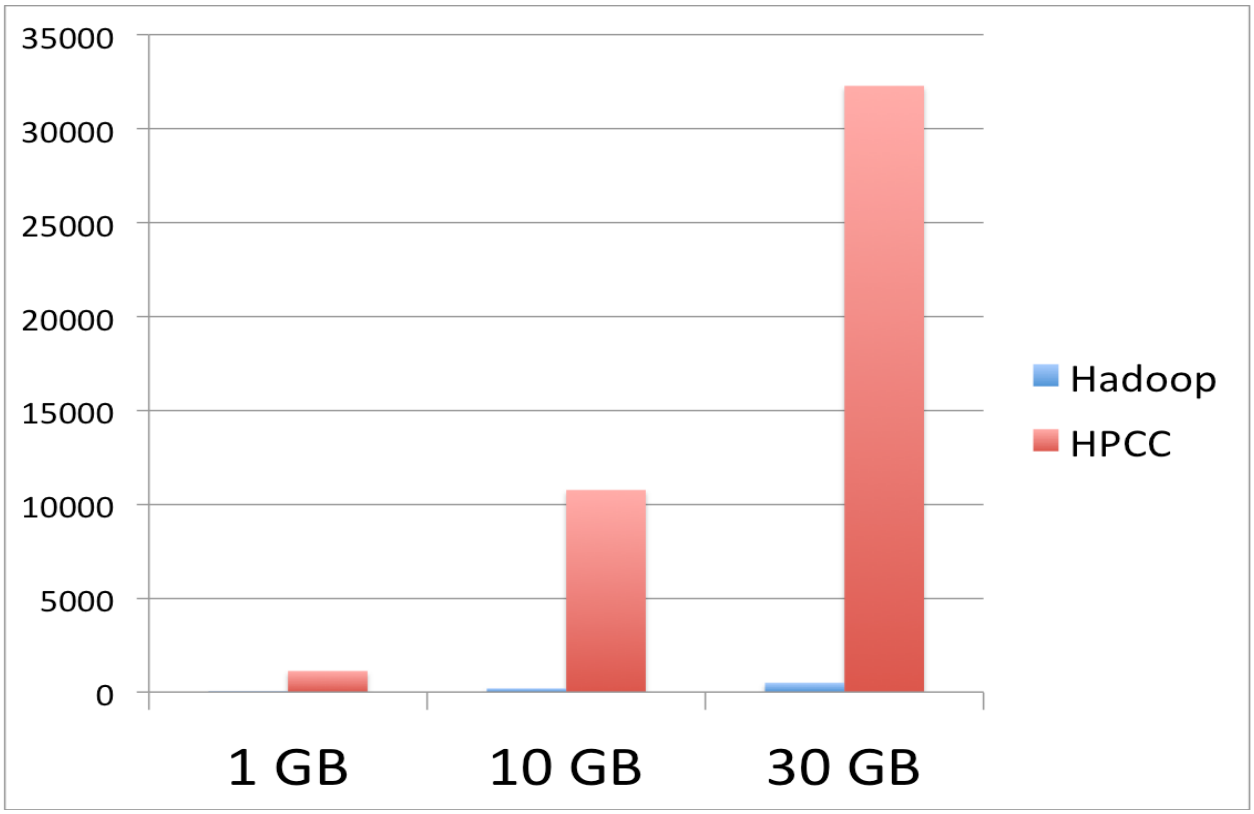

Figure 5: Histogram- Movies

The figure above compares the performance of Hadoop vs HPCC for processing the Histogram-Movies benchmark. Hadoop was found to be much faster than HPCC for 10, 20 and 30 GB of data.

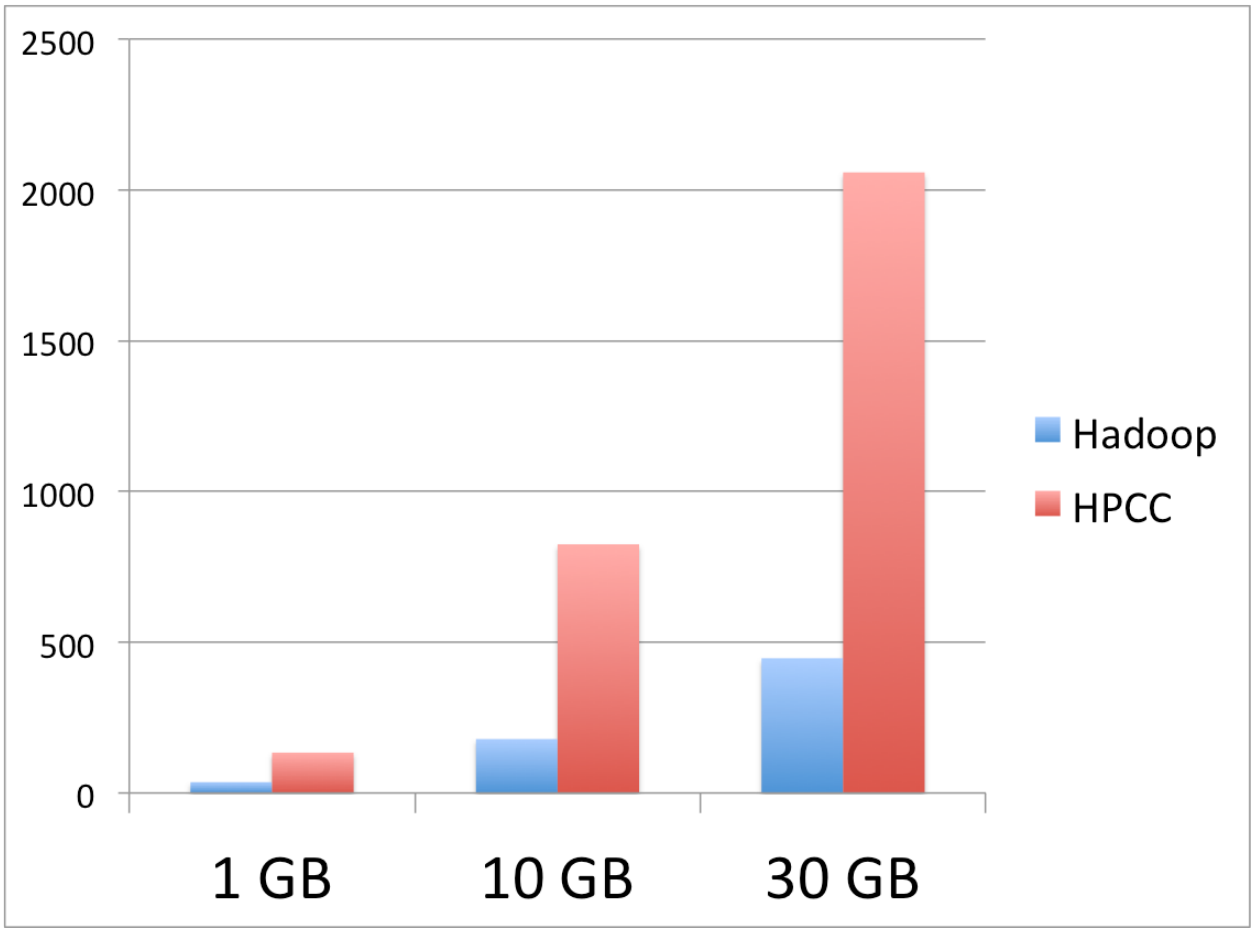

Figure 6: K-Means

The figure above compares the performance of Hadoop vs HPCC for processing the K-Means benchmark. Hadoop was found to be much faster than HPCC.

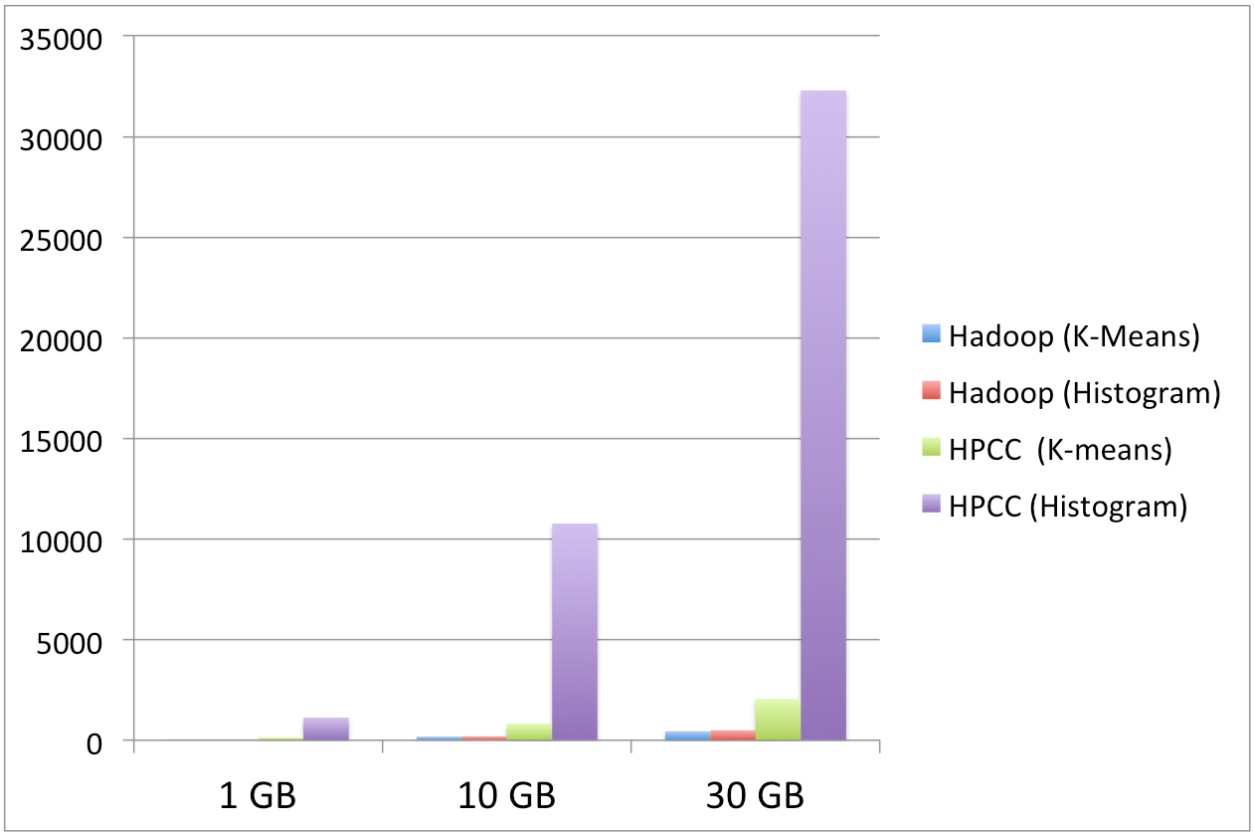

Figure 7: Histogram and K-Means

K-means and Histogram use the same input data, but the difference between their result is notable, so the figure above compares the performance of Hadoop vs HPCC for processing the K-Means benchmark and Histogram benchmark. Hadoop was always found to be faster than HPCC. In terms of HPCC alone, HPCC performed much faster in K-Means task than in Histogram task.

Discussion and Concluding Remarks

According to the results, we see that HPCC is quite fast when the requirement for data preprocessing is low. This is observed in Word-Count (10 G/20 G), Inverted-Index and Adjacency-List. In Word-Count, HPCC does not need to keep reading data a lot of times to transform it (only one split transformation is required). In Inverted-Index, one more DEDUP with SORT is introduced to eliminate duplicate records, so its performance is worse than Word-Count's. These results might also be because of more output from Inverted-Index(word:filename instead of word:word-count), but we do not believe this to be a major factor due to the high input/output ratio of the scripts. The preprocess for Adjacency-List involves "AGGREGATE" to scan the input files twice and an additional third time to eliminate records that do not meet the set criteria, which is more than what is demanded by Word-Count and Inverted-Index. This can explain why the difference between HPCC and Hadoop is smaller.

For HPCC, if data-preprocessing requirement is high, the performance decreases dramatically, especially when the transformation propagates to many more records than the original set. This is just what K-means and Histogram showed. In Histogram, one original record will propagate to around 2000 records in average. A point to note is that K-means and Histogram share the same input data, and the data pre-processing is also similar. However, we truncated data in K-means, so the results of K-means is much better than the Histograms'.

Another notable thing is that Hadoop fails to complete Adjacency-List on 30 GB data. The failure of Hadoop is due to the memory shortage. We encountered similar issues for Adjacency-List on 10 GB and 20 GB when we used 128 MB as block size for Hadoop. HPCC's performance does not scale linearly for Word-Count on 30 GB data. We did not find a very clear root cause but we think it may be related to memory. We monitored the memory on our cluster during the execution of Word Count on 30 GB and found that it becomes nearly exhausted and server hardly responded. Although the performance of HPCC was not good on 30 GB data, HPCC was more robust and Hadoop was observed to be more sensitive to memory shortage.

Besides the performance results, the programing models of Hadoop and HPCC is also worthy of discussion. The MapReduce programming model works on the general assumption that the input to output ratio is high i.e. large datasets are processed to generate smaller final values. However, we cannot control the order in which the maps or reductions are run. Provided each Map and Reduce is independent of the ongoing Maps and Reduces, the operations can be run in parallel. Also, the reduce operations can not take place until the map operations have completed processing.

In ECL programing model, the declarative characteristic and the concept of "order-less execution" distinguish ECL from other imperative programming language. These new concepts demand a new mindset, but it is not challenging when we think of them as a relationship between main function and other functions invoked by the main function. It is more challenging to adapt to dataset manipulation mindset. On the other hand, ECL provides lots of libraries to do data mining or machine learning, so it can save a lot of time instead of reinventing wheels for such kind of project. Moreover, we think ECL offers a powerful function "DISTRIBUTE" to re-distribute data across the cluster during runtime. It is convenient when one data analysis needs to analyze data on different criterions. In a Hadoop job, there is only one chance (partition step) to do the same, but we are free to do it any number of times in HPCC.

Acknowledgments

We would like to show our gratitude to Prof. Vincent W. Freeh for his guidance during the course of this project and his support for providing us with a virtual machine cluster as the testbed of our project. We also thank Hui Guo for his valuable review on our project proposal and the Intermediate project report.

References

[1] Ahmad, Faraz, et al. "Puma: Purdue mapreduce benchmarks suite." (2012).

[2] MapReduce from Wikipedia (http://en.wikipedia.org/wiki/MapReduce)

[3] MapReduce from Apache Foundaton (http://wiki.apache.org/hadoop/MapReduce)

[4] Hadoop MapReduce Next Generation - Setting up a Single Node Cluster (http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleCluster.html).

[5] Hadoop MapReduce Next Generation - Cluster Setup (http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html).

[6] White, Tom. "Hadoop: The definitive guide. " O'Reilly Media, Inc., 2012.

[7] Boca Raton Documentation Team. "HPCC System ECL Programmers Guide".

[8] Boca Raton Documentation Team. "HPCC System ECL Language Reference".

[9] Boca Raton Documentation Team. "HPCC System Standard Libarary Reference".

[10] Boca Raton Documentation Team. "HPCC System Installing & Running the HPCC Platform".

[11] The Apache Software Foundation - Hadoop MapReduce Tutorial (http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html).

[12] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.
http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[13] Middleton, A., and Ph D. LexisNexis Risk Solutions. "Hpcc systems: Introduction to hpcc (high-performance computing cluster)." *White paper, LexisNexis Risk Solutions* (2011).

[14] Middleton, A. "Hpcc systems: data intensive supercomputing solutions." *White paper, LexisNexis Risk Solutions* (2011).