RICE UNIVERSITY

Optimized Runtime Systems for MapReduce Applications in Multi-core Clusters

by

Yunming Zhang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Vivek Sarkar, Chair
E.D. Butcher Chair in Engineering
Professor of Computer Science
Professor of Electrical and Computer Engineering

Alan Cox
Professor of Computer Science
Professor of Electrical and Computer Engineering

John Mellor Crummey
Professor of Computer Science
Professor of Electrical and Computer Engineering

Houston, Texas

May, 2014

ABSTRACT

Optimized Runtime Systems for MapReduce Applications in Multi-core Clusters

by

Yunming Zhang

This thesis proposes a novel runtime system, Habanero Hadoop, to address the inefficient utilization of memory on multi-core machines by the Hadoop MapReduce runtime system. The Hadoop runtime duplicates large in-memory data structures on each node, reducing the available memory for the application. This memory inefficiency leads to a scalability bottleneck in problems such as data clustering and classification. The Habanero Hadoop system integrates a shared memory model into the fully distributed memory model of the Hadoop MapReduce system, eliminating duplication of in-memory data structures. Previous work optimizing multi-core performance for MapReduce runtimes focused on maximizing CPU utilization rather than memory efficiency. This thesis explores multiple approaches to improving the memory efficiency of the Hadoop MapReduce runtime. The resulting Habanero Hadoop runtime can increase the throughput and maximum input size for widely-used data analytics applications such as KMeans and hash join by two times.

Acknowledgments

I would first like to express my deepest gratitude to my advisors, Prof. Vivek Sarkar and Prof. Alan Cox for their invaluable support, advice and feedback. The thesis would not have been possible without Prof. Cox and Prof. Sarkar's patient guidance, enthusiasm and support. I want to give special thanks to Prof. Cox for his constant commitment to the project since the very early stage.

I would also like to thank my committee member, Prof. Mellor-Crummey, for his time meeting with me and providing feedback on the project. His comments have made the thesis much better in many ways.

I am grateful to Prof. Chris Jermaine for his help on developing applications and early feedback on the research.

I want to thank fellow students Ruoyu Liu, Mehul Chadha and Max Grossman for their work on developing the hash join application and helpful discussions. Special thanks to Florin Dinu, whose experience and scripts have greatly contributed to the research.

I give my thanks to the Habanero group members for their feedback and help on the thesis project and help since my sophomore year.

Last but not the least, I am extremely grateful to my parents for their support over the years. They always believed in me. Without their trust, I wouldn't be here today.

Contents

Abstract ii

Acknowledgments iii

List of Illustrations vi

List of Tables viii

1 Introduction 1

1.1 Thesis Statement 3

1.2 Contributions 3

1.3 Thesis Roadmap 4

2 Background 6

2.1 MapReduce Programming Model 6

2.2 Hadoop MapReduce Runtime 9

2.2.1 Hadoop MapReduce for Multi-core 10

3 Motivating Applications 15

3.1 Hash Join 15

3.2 KMeans 17

3.3 K Nearest Neighbors 19

4 Habanero Hadoop 22

4.1 Parallel Mapper 23

4.1.1 Design and Implementation 23
4.1.2 Programming Model 25
4.2 Compute Server 28
4.2.1 Design and Implementation 28
4.2.2 Programming Model 33
4.3 Hybrid Approach 34
4.3.1 Design and Implementation 34
4.3.2 Programming Model 37

5 Results 39

5.1 Experimental Setup 39
5.2 Application Benchmarks 41
5.2.1 Hash Join 41
5.2.2 KMeans 50
5.2.3 K Nearest Neighbors 58

6 Related Work 62

6.1 MapReduce Runtime Systems 62
6.2 Data Analytics Using MapReduce 65
6.3 Hash Join Using MapReduce Systems 67

7 Conclusions and Future Work 68

7.1 Future Work 70

Bibliography 72

Illustrations

1.1 Memory Wall for hash join Application 3

2.1 The MapReduce Programming Model 6

2.2 User implemented Mapper Class 8

2.3 User implemented Reducer Class 8

2.4 How Hadoop runs a MapReduce job 10

2.5 Setting the map slot and reduce slot property in mapred-site.xml . . 11

2.6 Hadoop MapReduce on a four cores system 13

2.7 Hadoop MapReduce increasing available memory for each map task on a four cores system 14

3.1 Hash join on multi-core system 16

3.2 KMeans 17

3.3 KMeans on multi-core system 20

3.4 K Nearest Neighbors 21

3.5 KNN on multi-core system 21

4.1 Implementation of the original sequential Hadoop Mapper 23

4.2 Implementation of the ParMapper 24

4.3 Parallel Mapper for multi-core systems 25

4.4 The original programming model extending the Mapper 26
4.5 The new programming model extending the ParMapper 26
4.6 Compute Server for multi-core systems 29
4.7 Hadoop MapReduce on a compute node 30
4.8 Setting the reuse number of each JVM to four 30
4.9 Design and Implementation of the Compute Server 32
4.10 The original programming model using the Hadoop Mapper 34
4.11 The new programming model using the Compute Server 35
4.12 Design and Implementation of the Hybrid Approach 37
4.13 Hybrid approach for multi-core systems 38

5.1 CPU utilization for hash join on a single-node with eight cores 42
5.2 Breaking the memory wall for hash join using the Compute Server on a three-node cluster 43
5.3 Heap memory size on each compute node for hash join 47
5.4 ParMapper and Compute Server's impact on throughput in a three-node cluster 49
5.5 CPU utilization on a single compute node with eight cores for KMeans 50
5.6 Breaking the memory wall for KMeans on a four-node cluster 52
5.7 aggregated heap memory size on each compute node for KMeans . . . 55
5.8 ParMapper and Compute Server's impact on throughput on a four-node cluster for KMeans 56
5.9 Aggregated heap memory size on each compute node for KMeans . . 57
5.10 Breaking the memory wall for KNN on a four-node cluster 59
5.11 Aggregated heap memory size on each compute node for KNN 61

Tables

5.1 Number of total full garbage collections calls on each compute node for hash join 45

5.2 Number of total regular garbage collections calls on each compute node for hash join 46

5.3 Number of total full garbage collections calls on each compute node for KMeans 53

5.4 Number of total garbage collections calls on each compute node for KMeans 54

5.5 Number of total full garbage collections calls on each compute node for KNN 60

Chapter 1

Introduction

Big data computing has become an essential part of web scale companies and scientific communities. For example, when indexing the web, Google stores and processes uncompressed webpages it gathers to create the inverted index data, which is needed to provide web search services. The uncompressed web pages and the inverted index consist of tens of terabytes of data [1]. Scientists also need to find ways to process petabytes of data produced by instruments such as the Large Hadron Collider or astrophysical observations. Many data analytics problems center around clustering together similar data, classifying unknown data based on classified data and joining different data sets. An efficient and scalable big data computing software framework can not only save scientists and programmers thousands of hours and millions of dollars every year, but also allow them to tackle larger problems.

Many of these data intensive applications are trivially parallel and can scale to hundreds and thousands of machines in data centers. MapReduce [2] is a popular software framework that enables programmers to write data parallel programs that can run on thousands of machines with support for automatic concurrency management, locality-aware scheduling and fault tolerance. The Apache Hadoop implementation of MapReduce has been widely adopted due to its scalability, reliability and support from the open source community [3]. Hadoop runtime executes large scale data analytics applications using commodity class computer clusters at a fraction of the cost of expensive supercomputers or high-end servers [1].

The current Hadoop MapReduce implementation uses multi-core systems by decomposing a MapReduce job into multiple map/reduce tasks that can execute in parallel. Each map/reduce task is executed in a separate JVM instance. The number of JVMs created in a single node (machine) can have a significant impact on performance due to their aggregate effects on CPU and memory utilization.

For memory-intensive applications, Hadoop MapReduce's design for exploiting multi-core resources can lead to a performance and scalability bottleneck. Some read-only in-memory data structures used by the MapReduce applications are duplicated across JVMs. Additionally, creating a large number of map tasks incurs non-trivial aggregated memory overhead. As the applications try to solve larger problems, the size of the in-memory data structures increases. When the memory usage of map/reduce tasks starts approaching the memory limit allocated per JVM, the frequency of garbage collection calls increases significantly, leading to a decrease in the system's throughput. For example, a typical hash join application requires each map task to store a copy of the lookup table in memory [4]. To make sufficient memory available to each map task, memory intensive applications are often forced to restrict the number of map task JVMs created to be smaller than the number of cores in a node at the expense of reducing CPU utilization. This effect is shown in Figure 1.1. Throughput drops quickly when there is insufficient memory for the increasingly larger lookup table.

The fundamental reason for the memory inefficiency in Hadoop MapReduce model is that it adopts a fully distributed memory model. Each task on the same machine executes in its own memory space and can only take advantage of a single core efficiently.

The Habanero Hadoop system eliminates the duplication of data structures be-

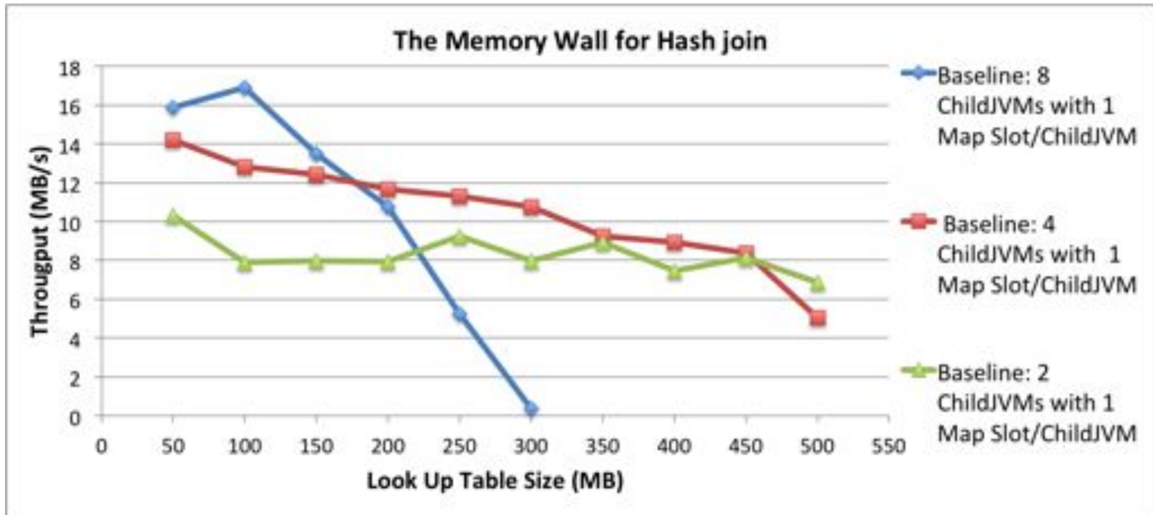

Figure 1.1 : Memory Wall for hash join Application

tween tasks on the same node through enabling intra-task parallelism and creating a shared-memory space between map tasks running on the same compute node. With reduced duplication of data structures, the optimized runtime increases the memory available to each map task and pushes back the memory wall significantly.

1.1 Thesis Statement

The performance of MapReduce runtimes for large-scale data analytics applications can be significantly improved by better utilization of the memory resources in multi-core systems.

1.2 Contributions

This thesis makes the following contributions:

- It provides a detailed study of the memory wall performance bottleneck in pop-

ular memory-intensive data analytics applications, such as KMeans, K Nearest Neighbors (KNN) and hash join running on the Hadoop MapReduce runtime.

- It describes an implementation of the Habanero Hadoop runtime that improves the memory efficiency of the Hadoop MapReduce runtime and breaks the memory wall for the applications by enabling efficient intra-map-task parallelism and creating shared-memory space across map tasks running on the same compute node.
- It presents an evaluation of the performance benefits using the optimized Habanero Hadoop runtime system for KMeans, KNN and hash join.

1.3 Thesis Roadmap

This thesis consists of the following chapters

Chapter 2 provides some background about the implementation of the Hadoop MapReduce runtime. Specifically, the chapter analyzes the current design for utilizing multi-core systems in Hadoop MapReduce and identify the source of the inefficient memory utilization.

Chapter 3 describes the design and implementation of hash join, KMeans and KNN applications on MapReduce systems.

Chapter 4 introduces novel runtime improvements implemented in the Habanero Hadoop system to improve the memory resource utilization in multi-core nodes in the cluster, including Parallel Mapper, Compute Server and Hyrbid approaches.

Chapter 5 evaluates the throughput, memory footprint and CPU utilization of three different applications running on the Habanero Hadoop runtime system.

Chapter 6 discusses related work on applying the MapReduce programming model

on clusters and shared-memory multi-core systems. In addition, the chapter also examines previous work that uses MapReduce for machine learning and hash join applications.

Chapter 7 summarizes the thesis and identifies opportunities for possible future work.

Chapter 2

Background

This chapter explains the MapReduce programming model and examines the implementation of the Hadoop MapReduce runtime. Specifically, we focus on the impact on memory efficiency of running multiple map tasks in parallel. In Hadoop MapReduce runtime, large in-memory data structures used by the popular data analytics applications are duplicated across different map tasks.

2.1 MapReduce Programming Model

The MapReduce programming model divides computation into map, shuffle and reduce phases as shown in Figure 2.1.

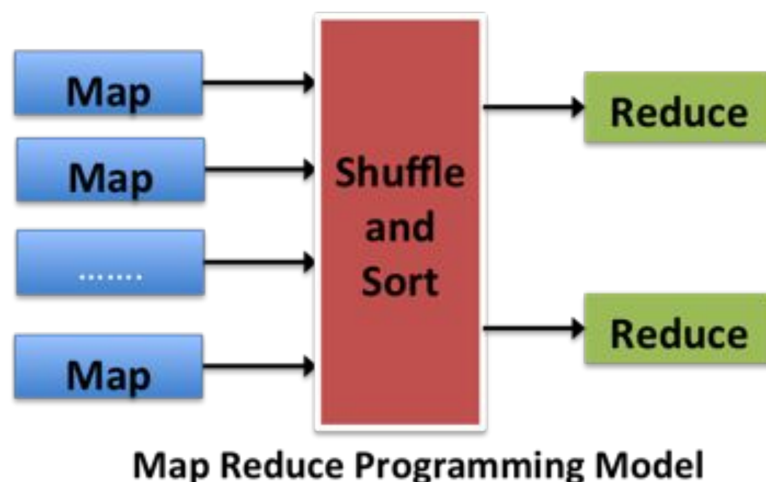

Figure 2.1 : The MapReduce Programming Model

The map phase partitions input data into many input splits and automatically stores them across a number of machines in the cluster. Once input data are distributed across the cluster, the runtime creates a large number of map tasks that execute in parallel to process the input data. The map tasks read in a series of key-value pairs as input and produce one or more intermediate key-value pairs. A key-value pair is the basic unit of input to the map task.

We use the WordCount application, which counts the number of occurrences of each word in a series of text documents, as an example. In the case of processing text documents, a key-value pair can be a line in a text document. The user can customize the definition of a key-value pair. The most important part for the users is to specify what to do with each key-value pair. They do so by extending a Mapper class and implementing a map() function within it, as shown in Figure 2.2. A map task is a process that executing the map function. The map tasks read in a series of key-value pairs as input and produce one or more intermediate key-value pairs. In the WordCount application, the user defines the function that processes each line of text input in the map phase. The user specified map function goes through each word and creates an intermediate key-value pair for each word in the line. The key-value pair consists of the word and a number, one, showing that it has been seen once in the line.

The intermediate key-value pairs are shuffled, grouped together and sorted by key. In WordCount, the key-value pairs that have the same key (the word) are grouped together to form a new key-value pair.

Once the intermediate key-value pairs are grouped and sorted, they are assigned to specific reduce tasks. The reduce stage then takes the grouped intermediate key-value pairs and processes each key-value pair to produce the final output pairs. Again, the

user specifies the function processing each input key-value pair to the reduce task. To do this, one extends a Reducer class and implements a reduce() function, as shown in Figure 2.3. In WordCount, the reduce function counts the occurrences for each word by summing up the set of ones produced each time the word was seen.

```
public class MyMapper extends
    Mapper <LongWritable,Text,
    Text, LongWritable> {

    public void map(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
    }
}
```

Figure 2.2 : User implemented Mapper Class

```
public class MyReducer extends
    Reducer <LongWritable,Text,
    Text, LongWritable> {

    public void reduce(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
    }
}
```

Figure 2.3 : User implemented Reducer Class

2.2 Hadoop MapReduce Runtime

MapReduce runtime is an open source implementation of the MapReduce model first proposed by Google [2]. It automatically handles task distribution, fault tolerance and other aspects of distributed computing, making it much easier for programmers to write data parallel programs. It also enables Google to exploit a large number of commodity computers to achieve high performance at a fraction of the cost of a system built from fewer but more expensive high-end servers [1]. MapReduce scales performance by scheduling parallel tasks on nodes that store the task inputs. Each node executes the tasks with loose communication with other nodes.

Hadoop [3] is an open source implementation of MapReduce. To use the Hadoop MapReduce framework, the user first writes a MapReduce application using the programming model we described in the previous section. The user then submits the MapReduce job to a jobtracker, which is a Java application that runs in its own dedicated JVM. The jobtracker is responsible for coordinating the job run. It splits the job into a number of map/reduce tasks and schedules the execution of the tasks across a large number of machines.

On each machine, there is a tasktracker process that is responsible for scheduling and executing map/reduce tasks on the machine. Essentially, the jobtracker assigns map/reduce tasks to tasktrackers and tasktrackers assigns the tasks to different cores. The tasktracker process also runs in its own separate JVM. The system is shown in Figure 2.4.

This thesis largely focuses on improving the execution of the tasks on a single compute node.

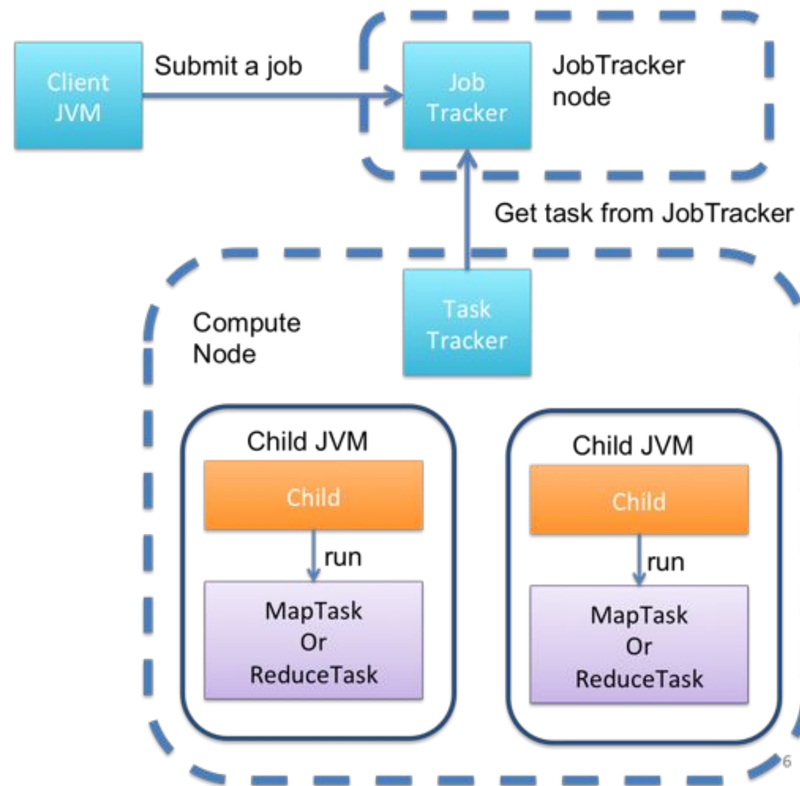

Figure 2.4 : How Hadoop runs a MapReduce job

2.2.1 Hadoop MapReduce for Multi-core

Tasktrackers have a fixed number of slots for map tasks and for reduce tasks. Each slot corresponds to a JVM executing a task. Each JVM only employs a single computation thread. To utilize more than one core, the user needs to configure the number of map/reduce slots based on the total number of cores and the amount of memory available on each node.

The configuration can be set in the mapred-site.xml file. The relevant properties are

mapred.tasktracker.map.tasks.maximum, mapred.tasktracker.reduce.tasks.maximum.

The examples in Figure 2.5 shows how to set four map slots and two reduce slots on each compute node. The setting can be used to express heterogeneity of the machines in the cluster. This setting can be different for each compute node. The reason is that different machines in the cluster can have a different number of cores and differing amounts of memory.

```
<property>
  <name>mapred.tasktracker.map.tasks.maximum</name>
  <value>4</value>
</property>


<property>
  <name>mapred.tasktracker.reduce.tasks.maximum</name>
  <value>2</value>
</property>
```

Figure 2.5 : Setting the map slot and reduce slot property in mapred-site.xml

The Hadoop MapReduce system first starts up using the user specified configurations. Once the system has been started, the tasktracker picks up tasks assigned to it by periodically making heartbeat method calls to the job tracker to see if there are any task ready for execution.

After the tasktracker has been assigned a task, it starts to execute the task. The first step is to copy user's application code packed in a JAR file to the local node's filesystem and unpack it into a local working directory. Then a taskrunner instance is created by the tasktracker to run the task. To prevent bugs in the user-defined map and reduce functions from crashing the tasktracker process, a new JVM is launched by the tasktracker to execute the map/reduce task. As a result, each map/reduce task is executed in a separate JVM instance. The number of JVMs created in a single

node (machine) can have a significant impact on performance due to their aggregate effects on CPU and memory utilization.

There is a tendency to spawn a large number of tasks, and thus JVMs, to improve CPU utilization in multi-core systems. For example, it is not uncommon to create 24 map tasks on an 8-core machine. However, this approach is only effective for non-memory-intensive applications because if each task takes up a significant amount of memory, it is difficult to create a large number of tasks on each compute node.

Since each JVM is isolated in its own memory space, each Map task running in its own JVM shares no memory with other map tasks on the same node. As a result, data structures used in map tasks are duplicated across JVMs, including in-memory, read-only data structures needed by map/reduce applications. For memory-intensive applications, Duplication of identical data structures can significantly reduce the total amount of memory available to each map task because the duplicated in-memory, read-only data structures needed by the application can take up a lot of memory space, as shown in Figure 2.6.

For example, a typical hash join application requires each map task to store a copy of the lookup table in memory [4]. Duplicating the lookup table will decrease the amount of memory available to each map task.

To make sufficient memory available to each map task, memory intensive applications are often forced to restrict the number of JVMs created to be smaller than the number of cores in a node at the expense of reducing CPU utilization. For example, in a machine with four cores and 4 GB of RAM, the system needs to create four map tasks to use the four cores. However, if 1 GB RAM is insufficient for each map task, the Hadoop MapReduce system can create only two map tasks with 2 GB RAM available to each task. With two map tasks, the runtime system utilizes only

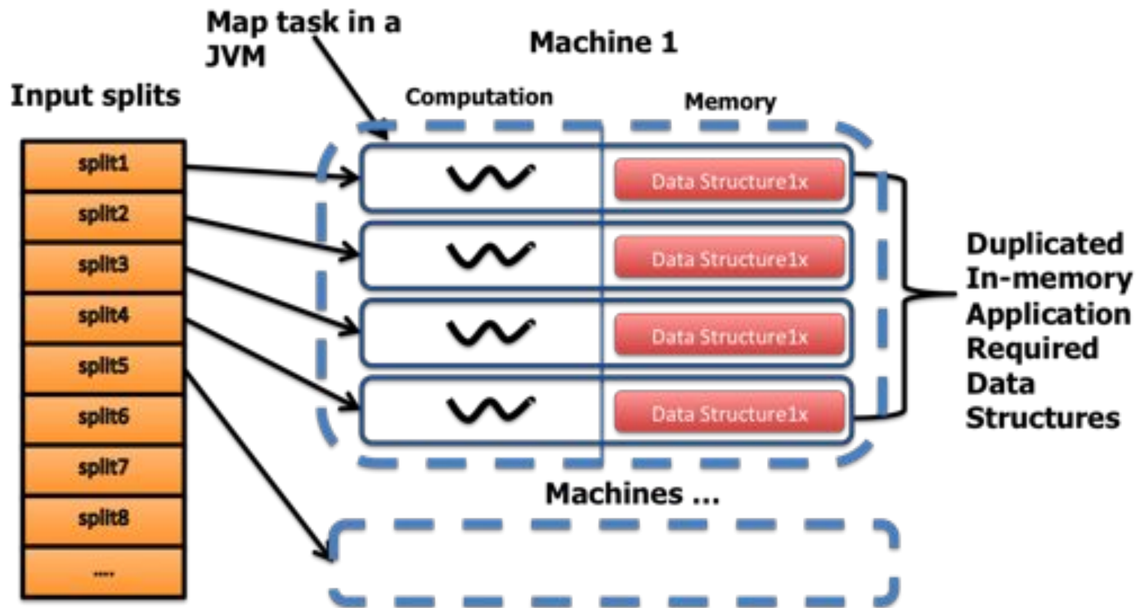

Figure 2.6 : Hadoop MapReduce on a four cores system

two of the four available cores or 50 percent of the CPU resources. This is shown in Figure 2.7.

In the next chapter, we will examine the design and implementation of three popular memory-intensive data analytics applications.

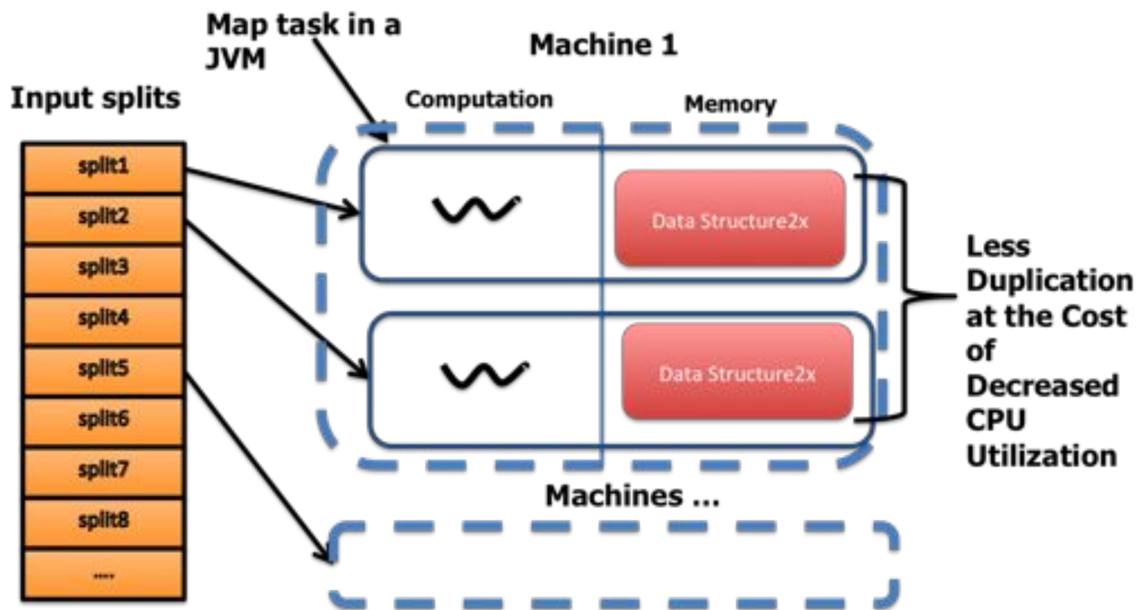

Figure 2.7 : Hadoop MapReduce increasing available memory for each map task on a four cores system

Chapter 3

Motivating Applications

The large memory footprint of read-only in-core data structures used by hash join, KMeans and K Nearest Neighbors (KNN) limit the number of separate processes that can run on a multi-core node. In this chapter, we describe the implementations of three popular memory-intensive MapReduce applications: Hash join [5], KMeans [6, 7] and KNN. We characterize these applications according to their memory usage, I/O and computation intensity.

3.1 Hash Join

Hash join is an important application for combining data from different sources. A MapReduce implementation of hash join is essential for analyzing large data sets. For example, hash join plays a crucial role in analyzing terabytes of unstructured logs [5]. It is also an fundamental part of distributed databases such as Pig [8] and Hive [9].

Hash join takes as input two tables, a data table S and a reference table R, and performs an equi-join on a single column of the two tables, as shown below

$S \bowtie R, with|S| \gg |R|.$

It is often the case that the size of one table S is much larger than the other data set R [4, 10, 5]. Since the larger data set S is too large to fit in one compute node, the hash join application has to divide up the data set S to process it across compute nodes in parallel. The smaller data set R is small enough that it can be loaded into

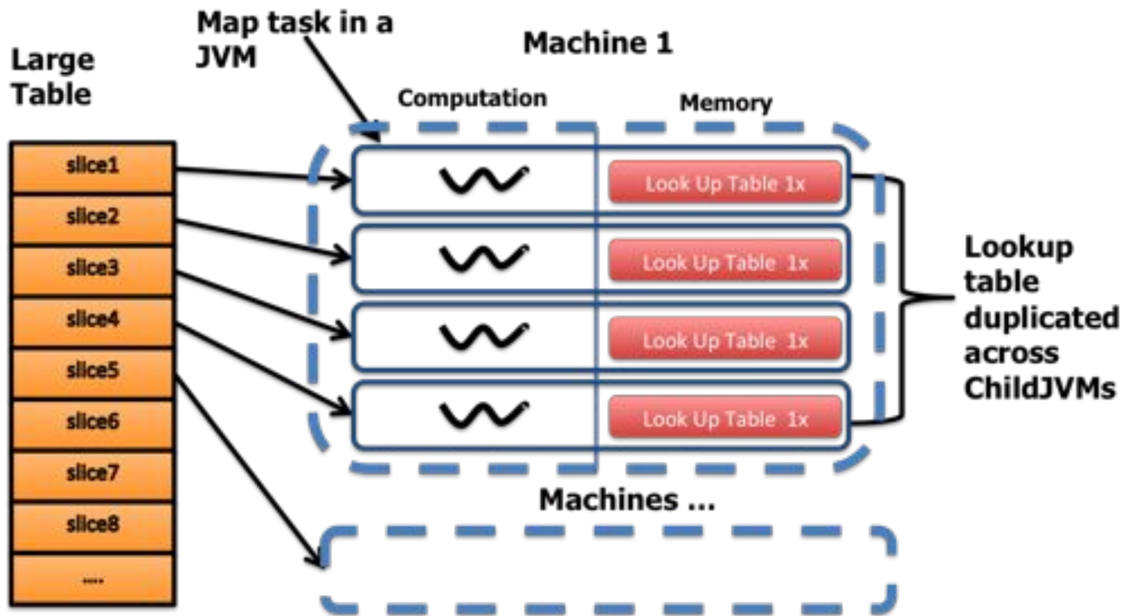

Figure 3.1 : Hash join on multi-core system

the memory of a single compute node. The implementation only needs to use the map phase of the MapReduce job.

The Hadoop MapReduce runtime splits up S into small pieces and uses them as input to each map task. It broadcasts the smaller R to every map task and loads R into a hash table, as shown in Figure 3.1.

Each map task reads in one key-value pair from the split of S at a time and queries the hash table containing R to see if there is a match. If there is a match, then the matching key-value pair is outputted. At the end of the map phase, the output key-value forms the output table.

It is implemented as a broadcast join by Blanas et al [5], Fragmented Replicated Join [4] in Pig [8] and MapSide join [10] in Hive [9]. We use a version that is a variant of the broadcast join.

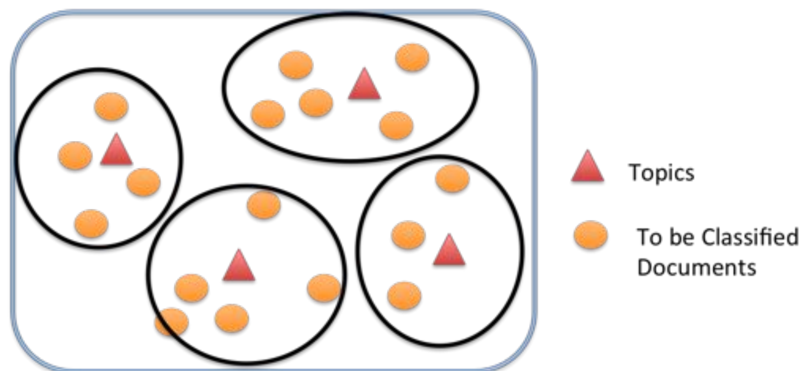

Figure 3.2 : KMeans

The memory footprint of each map task is large because the hash table containing R takes up a lot of memory [10]. Hash join is not a compute intensive application as each query to the hashtable takes constant time. The overall time complexity of the Hash join application is $O(|S|)$.

3.2 KMeans

Clustering algorithms group together similar items. Many clustering algorithms are widely used in data analytics applications, including KMeans, FuzzyKMeans and LDA [6, 7]. KMeans is a representative clustering application that is often used to genereate topics in databases of documents. It takes as input the parameter k, and partitions a set of n sample objects into k clusters.

For example, one popular application of KMeans is finding topics in news articles. The n sample objects are news articles that we want to cluster into topics. The k centroids are the topics as shown in Figure 3.2.

The algorithm first chooses k random objects as centroids. Then, it assigns every sample object to a cluster that it is most similar to. Once all the sample objects have

been assigned to a cluster, the algorithm recalculates the centroid location of each cluster. The process repeats until the centroid value stabilizes.

Since each sample object can be processed independently, the algorithm splits the sample objects into subgroups and clusters samples in each subgroup separately in parallel.

We use an implementation of KMeans following a widely adopted design [6, 7]. Each of the n objects is represented as a sample vector. The cluster centroids are represented as a vector as well. In the Map phase, each map task first reads in a file containing centroid data and creates an array containing all the cluster centroid vectors. Next, each map task reads in a subgroup of the n sample vectors. A map task calculates the similarity between each sample vector with all k centroids and assigns the sample vector to the most similar cluster. We use Euclidean distance for measuring the similarity of two vectors. At the end of each map task, the algorithm creates a partial sum of the sample vectors and the number of sample vectors in each cluster.

In the reduce phase, the algorithm adds up the partial sums of each cluster and divides it by the number of samples in each cluster to calculate the new center coordinates of the cluster.

Every MapReduce job recalculates the cluster centroids. Multiple MapReduce jobs are chained together to iteratively improve the quality of the cluster centroids. The algorithm terminates when the centroids have stabilized or when it reaches a maximum number of iterations.

The KMeans application is very memory-intensive because each map task needs to keep the cluster centroids data in memory. A map task performs only read operations on the cluster centroids data. Since each map task is running in a separate JVM

with no shared memory space among the JVMs, the read-only data are duplicated multiple times as shown in figure 3.3. This duplication reduces the amount of memory available to each map task, limiting the number of clusters KMeans can generate.

For many clustering applications, the number of clusters K can be very large. Mahout [7] suggested the user to use 2000 clusters for a database of a million news articles. The number is calculated based on the assumption that there are about 500 news articles published about every topic. We will need more clusters if we assume fewer news articles are published about each topic or if we are working with a larger database. In addition, scientific applications, including clustering astronomical data, need to generate tens of thousands of clusters.

As a result, a memory efficient runtime can allow scientists and programmers to perform clustering operations with a large number of clusters without incurring any penalty to the throughput.

The KMeans application is compute intensive. For n sample objects and k clusters, you need to perform $O(|n| \times |k|)$ similarity computations.

3.3 K Nearest Neighbors

Classification is another popular class of machine learning applications that use a large in-memory data set [11]. One of the most widely used classification algorithm is the K Nearest Neighbors algorithm (KNN).

KNN uses two data sets, a query set Q and a training set T. It classifies the elements of Q into different categories by comparing every element with the already classified training set T. It chooses the K closest elements in T, as shown in Figure 3.4.

The Hadoop MapReduce runtime splits up T into small pieces and uses them as inputs to the map tasks. In the map phase, the algorithm first loads into memory

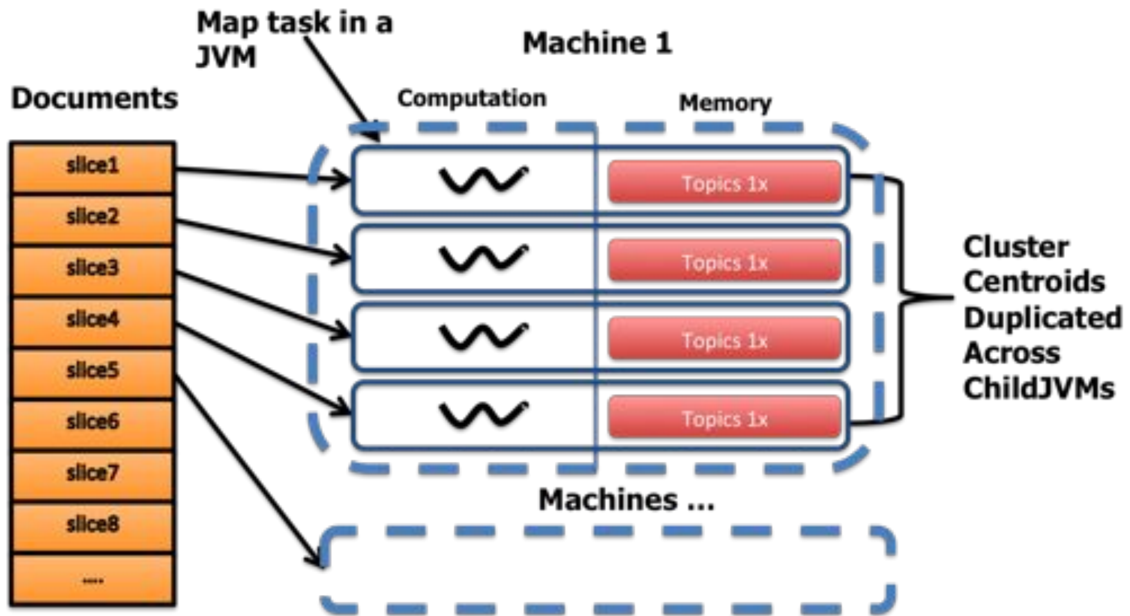

Figure 3.3 : KMeans on multi-core system

the complete query set Q, as shown in Figure 3.5. Next, for each element in Q, it calculates the distance to every element in the split of T. The application uses a priority queue to store partial results containing only the top K most similar elements in T. In the reduce phase, the reduce tasks aggregate all the partial top K elements to compute the overall top K elements in T for each element in Q.

KNN is a memory intensive application because it keeps the query set Q in the memory. KNN is a more compute intensive application than Hash join because it performs distance calculations between every point in Q and every point in T, leading to a time complexity of $O(|Q| \times |T|)$.

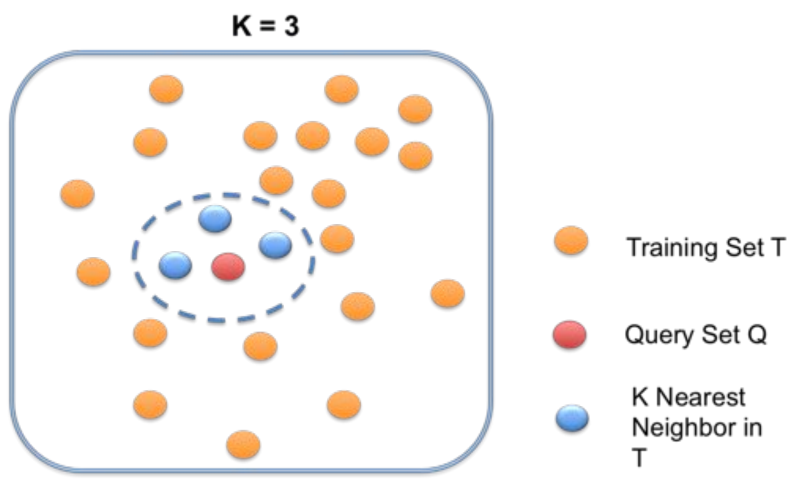

Figure 3.4 : K Nearest Neighbors

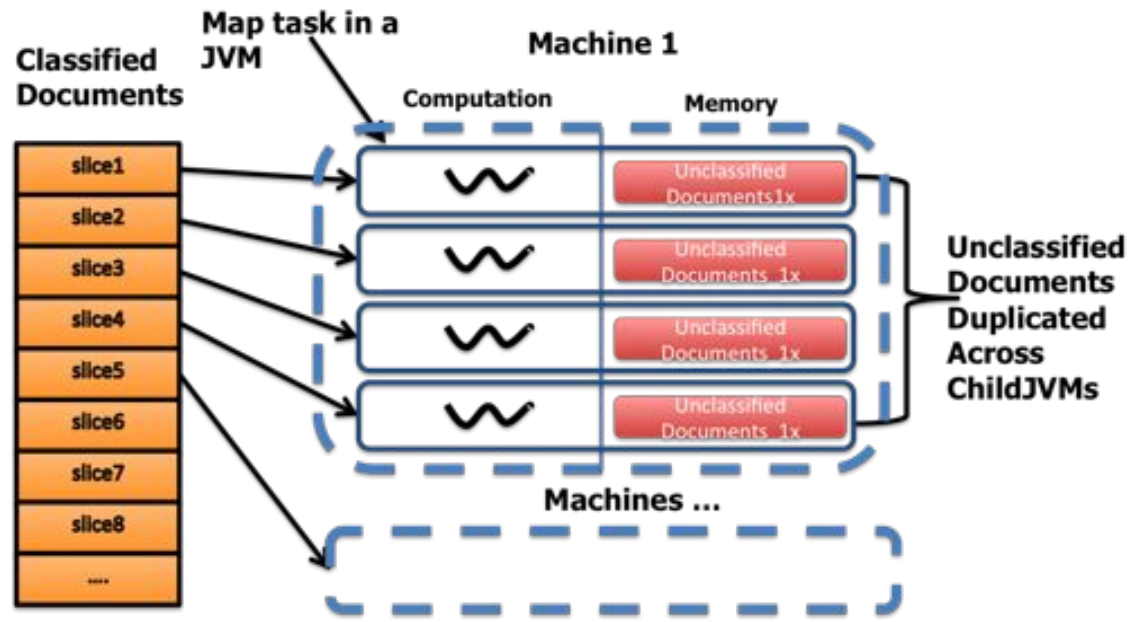

Figure 3.5 : KNN on multi-core system

Chapter 4

Habanero Hadoop

This chapter describes the design, implementation and programming model of the approaches that we explored in Habanero Hadoop. They are ParMapper, Compute Server and Hybrid approaches. These approaches improves the memory efficiency of the multi-core system without incurring any penalty to the throughput of the system.

We focus on optimizing the map phase of the applications because the map phase dominates the computation time for popular data analytics applications, including KMeans, KNN and hash join. The optimizations we implemented should be easily applied to the reduce phase as well.

We started with the ParMapper approach. The ParMapper approach is effective for compute intensive applications such as KMeans and KNN. However, the limited I/O capability of ParMapper is inadequate for more I/O intensive applications such as hash join.

To overcome the I/O bottleneck in ParMapper, we introduce the Compute Server approach that runs multiple map tasks in the same JVM, allowing the map tasks to perform I/O operations in parallel. The Compute Server also creates a shared memory space across different map tasks to eliminate duplications of the identical data structures.

Finally, we explore a hybrid approach that achieves a good balance between I/O, CPU utilization and memory efficiency.

4.1 Parallel Mapper

4.1.1 Design and Implementation

Because each Hadoop map task can only utilize a single computational thread, the tasktracker has to create a large number of map tasks to utilize multiple cores, resulting in the duplications of data structures across map tasks. The ParMapper subdivides a single map task into multiple sub tasks and executes the sub tasks in parallel using multiple computational threads. This way, the tasktracker only needs to create one parallel map task to fully utilize multiple cores. Running a single parallel map task on each compute node avoids the duplications across a large number of map tasks on each node.

To explain the implementation of the ParMapper, we first describe the implementation of the original Hadoop Mapper. By default, mappers sequentially generate key-value pairs from their input split. Every time a key-value pair is generated, the map task immediately processes it using the user defined map function. This design is inherently sequential as the map task has to finish processing one key-value pair before moving on to process the next one. The sequential way of processing the input key-value pairs is shown in Figure 4.1.

```
while (hasAvailableKeyValuePair){
    map(currentKey, currentValue)
}
```

Figure 4.1 : Implementation of the original sequential Hadoop Mapper

The ParMapper improves upon the original Hadoop mapper by subdivide the input key-value pairs to the map task into chunks and process different chunks in

parallel, as shown in Figure 4.2. The ParMapper also generates parallel tasks dynamically and overlaps I/O and computation .

```
while (hasAvailableChunkKeyValuePair){
    async{
    for( pair in Chunk)
        map(currentKey, currentValue)
    }
}
```

Figure 4.2 : Implementation of the ParMapper

To overlap I/O and computation, the ParMapper dedicates a single I/O thread to prefetch key-value pairs into a buffer while other worker threads are executing map tasks. To do this, the runtime allocates a new buffer for each async task.

To generate dynamic task parallelism, the I/O thread starts an asynchronous task to process a buffer once it is full. The algorithm for choosing the buffer size is described below. A separate buffer is used for each worker thread to allow the JVM to free up buffers in completed tasks.

To load balance across multiple cores, the ParMapper needs to choose a good task granularity for each worker thread. The granularity of each task is decided by the buffer size. Since the execution time for each call to a map function for each pair is different from application to application, there is not a fixed buffer size that is good for all applications. We need to adaptively select a buffer size that will achieve good performance. To do this, the main thread first reads in a small number of input key-value pairs as a sample chunk and records the time it took to process the sample chunk. Based on an empirically chosen desired running time for each chunk, the runtime calculates a good buffer size. The implementation of ParMapper is shown in

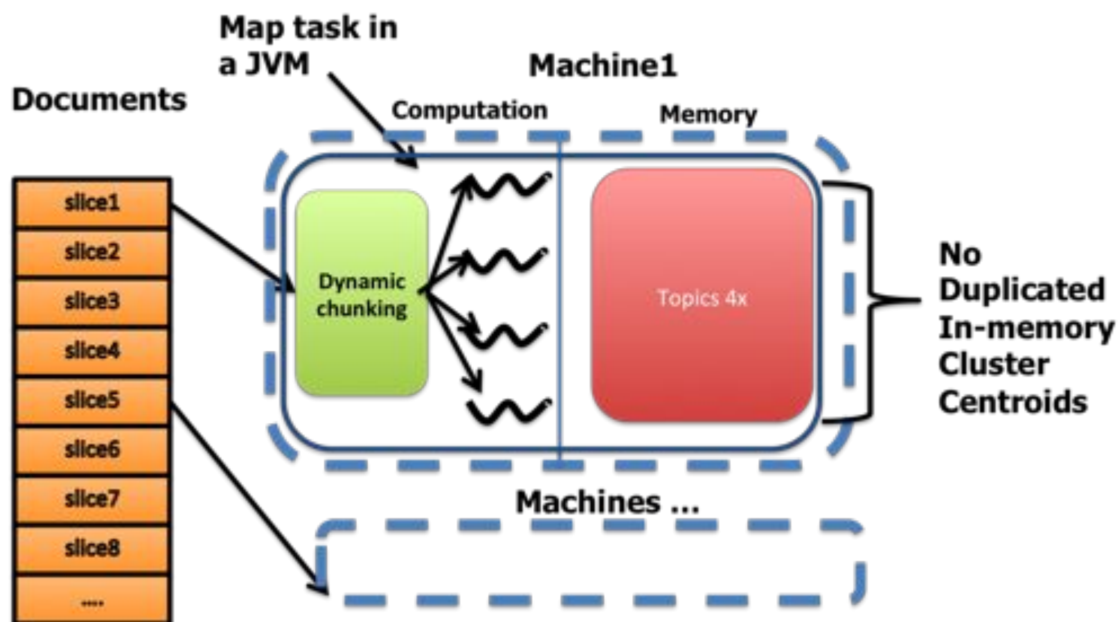

Figure 4.3 : Parallel Mapper for multi-core systems

Figure 4.3.

The ParMapper also improves CPU utilization for some compute intensive tasks. The runtime dynamically subdivides each map task so that the granularity of tasks assigned to each core is smaller than the original map task. The improved granularity contributes to better load balance across cores. The effect is shown in Chapter 5 for the K Nearest Neighbors application.

4.1.2 Programming Model

The programming model for using the ParMapper is very straightforward as well. We created a ParMapper class that can be extended by the user, just like a regular Mapper class. The ParMapper automatically handles decompositions and parallelization of individual map tasks. The users simply need to extend the ParMapper class instead

of the Hadoop Mapper class to take advantage of the intra-task parallelism. The multi-threading is completely transparent to the user. The old programming model using Hadoop Mapper is shown in Figure 4.4 and the new programming model is shown in Figure 4.5.

```
public class MyMapper extends
    Mapper<LongWritable,Text,
    Text, LongWritable> {

  public void map(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
  }
}
```

Figure 4.4 : The original programming model extending the Mapper

```
public class MyMapper extends
    ParMapper <LongWritable,Text,
    Text, LongWritable> {

  public void map(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
  }
}
```

Figure 4.5 : The new programming model extending the ParMapper

The ParMapper requires user-provided map functions to be thread-safe, so that multiple input key-value pairs can be processed in parallel. Many MapReduce applications, for example grep, wordcount satisfy this constraint, as they don't share any

state when processing different key-value pairs. For applications, including KMeans and hash join, they share an in-memory data structure when processing different key-value pairs. However, since the map function is only performing read operations on the shared data structure, the map task can execute data-race free.

In the cases that there are write operations performed in the map function on a local variable, the users would need to use some locking mechanism to make sure there is no data race from parallel calls to the map function.

The ParMapper approach is efficient for compute intensive applications such as KMeans and KNN because a single I/O thread can generate enough tasks to saturate multiple cores. However, the parallel mapper approach can not take full advantage of the CPU resources for data analytics applications that are I/O intensive, such as hash join. For the I/O intensive applications, a single I/O thread can become a performance bottleneck.

We were unable to incorporate parallel I/O threads in the ParMapper approach because the Hadoop Distributed File System enforces synchronization on the input files to each map task. For example, each read access to the input file needs to update the logs on how many bytes were read. The number of bytes read is in turn used for tracking the progress of each map task. The additional synchronization prevents us from building an efficient parallel I/O implementation for each individual map task.

Another disadvantage to building parallel I/O suport for a single map task is the possible bottleneck on reading files from disk. Since the input split to a map task is likely to be a single file residing in the local file system. Multiple simultaneous seeks to the same file in the disk will not speed up the reading of data from the file.

4.2 Compute Server

4.2.1 Design and Implementation

The ParMapper uses a single map task to keep the system's CPU and I/O resources fully utilized. In the Compute Server appraoch, we try to exploit inter-task parallelism by running multiple map tasks in parallel withint the same JVM.

Running multiple map tasks in parallel achieves good I/O and computation resource utilization. Multiple map tasks can read different input split files in parallel without worrying about contention from reading the same input split file in the disk.

The aggregation of map tasks essentially achieves parallel I/O and deserialization of key-value pairs without modifying HDFS. Furthermore, running multiple map tasks at the same time can overlap I/O with computation. For instance, when one map task is loading input key-value pairs, another map task can be processing the key-value pairs, keeping the CPU busy. This overlap effect is very evident when we later analyze the CPU over time graphs for hash join in Chapter 5.

However, we need to create a shared memory space between different map tasks running on the same compute node to improve the memory efficiency of the system, as shown in Figure 4.6. The need for an efficient shared memory space across tasks is noted by Gillick et al. [11]. The fundamental problem with creating this shared memory space is that in the cluster MapReduce model, each map task is designed to be stateless and isolated in its own process.

This led to the design of a persistent Compute Server that runs multiple map/reduce tasks in parallel in the same Compute Server JVM.

The actual implementation of the Compute Server requires deeper understanding of how Hadoop MapReduce works. Figure 4.7 shows the current design and imple-

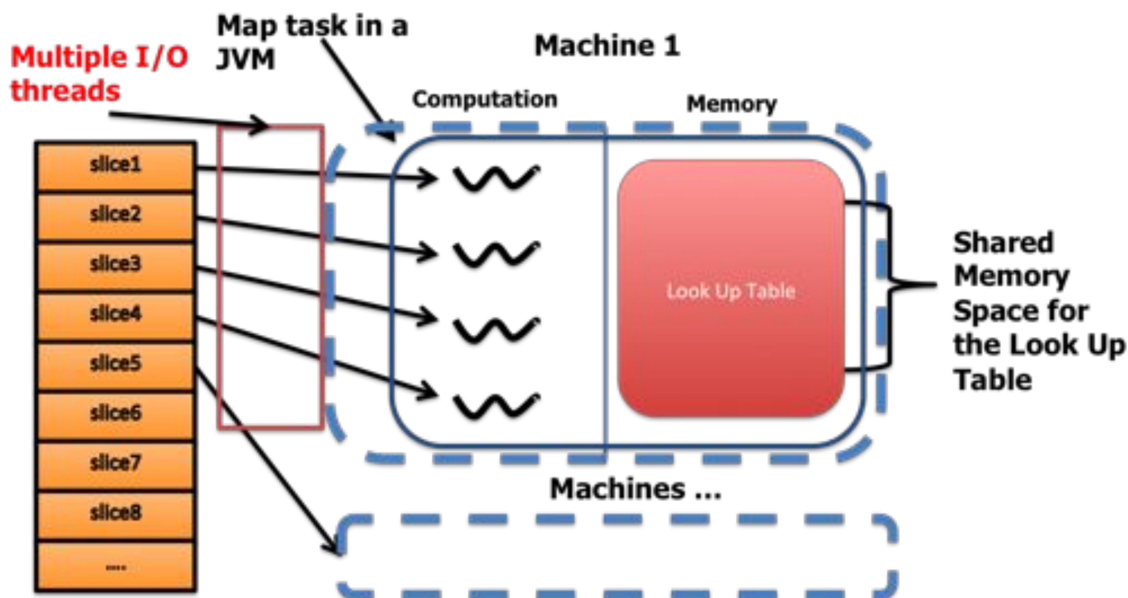

Figure 4.6 : Compute Server for multi-core systems

mentation of Hadoop MapReduce on a compute node. The tasktracker is responsible for scheduling and managing the map/reduce tasks running on the compute node. Each tasktracker has a JVM Manager class inside that manages the tasks assigned to the compute node. The JVM Manager uses a taskrunner to launch a Child process (highlighted in orange) to execute a map/reduce task.

After the Child JVM has been launched from the JVMManager, the Child process running inside is responsible for acquiring the task specific information from the TaskTracker using a RPC "umbilical" interface. The Child Process then uses the task specific information to copy the files required by the tasks, including the input split file for the task from HDFS to a local directory.

The first step towards building a Compute Server was to make the Child JVM persistent. By default, each JVM is launched to execute a task but then quickly shut down once the task has finished execution. This design makes sure that each

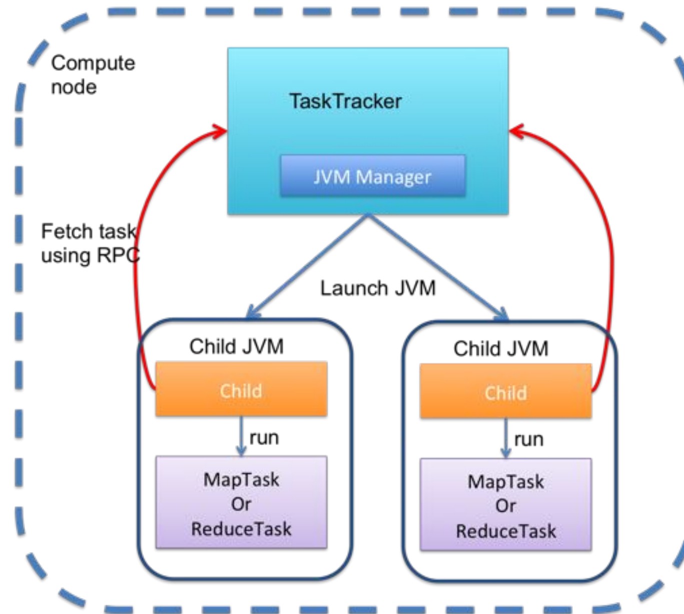

Figure 4.7 : Hadoop MapReduce on a compute node

task can start from a clean environment, unaffected by other long running processes. However, this policy creates a significant overhead by starting one JVM for every task. To alleviate this issue, Hadoop MapReduce has the option to set the number of times a JVM can be reused in the configuration file. The values are set using the following properties, as shown in Figure 4.8.

```
<property>
    <name>mapred.job.reuse.jvm.num.tasks</name>
    <value>4</value>
</property>
```

Figure 4.8 : Setting the reuse number of each JVM to four

By setting the value of mapred.job.reuse.jvm.num.tasks to four, it means that the

ChildJVM will execute 4 tasks before it shuts down. If you set the value to -1, it will reuse the JVM so long as there are tasks to execute. In order to make our Compute Server persistent throughout the job, we set this option to -1 to enable unlimited reuse.

Once we enabled unlimited reuse of the Child JVMs, we have the oppourtunity to perform optimizations that enable reuse of read-only in-memory data structures between tasks that execute in the same Child JVM sequentially. Since each map task needs to load an in-memory data structure before it can begin the processing of the key-value pairs, it is possible that we have the first task load the memory structure once and use it for all later map tasks executing in the same Child JVM. To do this, we simply declare the data structure as static. Before we load the data structure, we check if the data structure is already loaded. If it is already loaded, then we no longer need to spend time on deserializing data from the HDFS file and loading the data structures into memory. This optimization can lead to a quite significant performance improvement for memory-intensive but not very compute intensive applications. For example, hash join takes more than one third of the time of each map task loading the lookup table into memory. By using the optimization combining static data structure and Child JVM reuse, we can improve hash join's performance by one third. However, this optimization doesn't require the Compute Server and can be done in the original Hadoop MapReduce as well.

Once we have a persistent ChildJVM, the second step is enabling multiple tasks running in a Compute Server in parallel. The design of the Compute Server is shown in Figure 4.9. The goal is to have multiple Child threads running in the Compute Server JVM. To do this, we go back to JvmManager, where the ChildJVM is first launched. Instead of launching a ChildJVM, we modified the JvmManager source code to launch

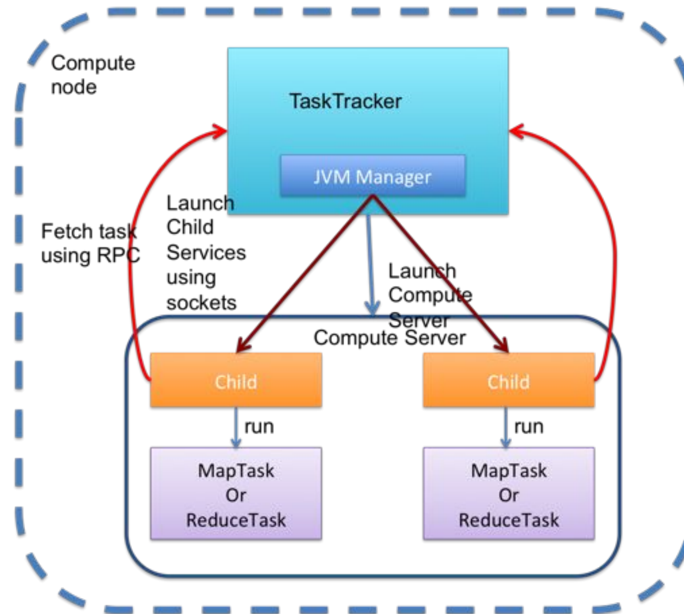

Figure 4.9 : Design and Implementation of the Compute Server

a Compute Server. Once the Compute Server is launched, the JvmManager starts a Child thread inside the Compute Server JVM through a socket connection. Currently, we hard coded the port numbers to establish the socket connection for the Compute Server. Different port numbers and JVMs are used for Map and Reduce tasks. The Child threads (highlighted in orange) inside the Compute Server gets the JvmId from the socket connection and uses it to fetch tasks through the original RPC ”umbilical” interface the same way the original ChildJVM operates. As a result, we can isolate the modifications to the level of ChildJVM and JvmManager because the tasktracker can not tell the difference between the Compute Server and the orignal multi ChildJVM scheme.

To achieve this, we modified the JvmManager's implementation when launching a Child task. We also created a multi-threaded server implementation to handle the

socket connection requests from JvmManager. In addition, the implementation of the Child process has to be modified, including removing static global variables, so that multiple Child processes can run in a single Compute Server.

With the Compute Server, we have multiple map tasks running in the same JVM and the same memory space. To share the in-memory data structures across different map tasks, we simply declare a data structure as static. In Java, static data structures are shared across different instances of the same class. In our case, since all the computation in the map tasks happen in the mapper class, the data structure can be shared across map tasks after it is declared as a static variable in the user's mapper implementation.

The next step is creating a synchronization scheme to make sure that the application the data structure required by the application is loaded into the memory only once. Currently, we achieve this by setting up a critical section in the set up phase of each map task using locks. Only the first thread to enter the critical section will load the data structure into memory. All future set up phases will do nothing if a check shows that the data structure is already loaded into the memory.

4.2.2 Programming Model

The Compute Server is largely a runtime level change. The existing code will work unchanged. However, to take advantage of the memory efficiency provided by the Compute Server, the large in-memory read-only data structures need to be declared as static in the user code. Furthermore, in the set up phase of each map task, a critical section needs to be set up to make sure that a data structure is set up only once.

Currently the runtime requires the users to modify their MapReduce application

code to eliminate the duplication of data structures. However, we believe in the near future, we can design a customized Mapper class that does the static variable declaration and critical section implementation for the users. Figures 4.10 and 4.11 show the changes needed to use the Compute Server.

```
public class MyMapper extends
    Mapper <LongWritable,Text,
    Text, LongWritable> {

    private DataStructure datastruct;

    public void setup(Context context){

       load the datastruct;

    }


    public void map(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
    }
}
```

Figure 4.10 : The original programming model using the Hadoop Mapper

4.3 Hybrid Approach

4.3.1 Design and Implementation

The original Hadoop MapReduce design forced users to create a large number of map/reduce tasks to create enough parallelism that can keep the multi-core systems' I/O and CPU resources fully utilized.

```
public class MyMapper extends
    Mapper<LongWritable,Text,
    Text, LongWritable> {

    private static DataStructure datastruct;


    public void setup(Context context){
      lock.lock();
      if (datastruct not loaded in memory){
       load the datastruct;
      }
      lock.unlock()

    }

    public void map(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
    }
}
```

Figure 4.11 : The new programming model using the Compute Server

In this chapter, we have described the alternative ParMapper and Compute Server approaches to improve the memory efficiency of the MapReduce runtime without sacrificing the CPU utilization.

The ParMapper approach allows us to control the number of computational threads used by a single map task. This approach works well for compute intensive applications. However, it can't fully utilize the I/O resources for less compute-intensive applications because each map task can only use a single I/O thread.

On the other hand, the Compute Server implementation enables multiple map tasks to run in parallel without duplicating the in-memory read-only data structures. Users can create a large number of map tasks to saturate the I/O and CPU resources.

This approach is good for hash join, which is an I/O intensive application.

However, this is not the case for more compute-intensive applications. Despite the fact that we have removed duplication of certain large in-memory read-only data structures required by map tasks, creating a large number of map tasks can sometimes still lead to a large memory footprint. Take KMeans for example, there are data structures used by the applications that cannot be shared across map tasks such as output cluster centroid partial sums. These data structures act as accumulators of the outputs from map tasks.

Furthermore, our current implementation has trouble scaling the performance with a large number of map tasks running in the Compute Server for compute intensive applications. As we will show in the results section, running eight map tasks in the Compute Server is a lot slower than running eight map tasks in eight separate ChildJVMs. It could be due to synchronizations on writing intermediate key-value pairs to disks. At this stage, we don't know where the performance overhead is from.

To minimize the impact of map task overheads, we need to keep the number of map tasks small to achieve the best memory efficiency. By combining the ParMapper and Compute Server approaches, we can achieve good CPU and I/O utilization for compute intensive applications like KMeans using only a few map tasks. The hybrid approach is shown in Figure 4.12. This approach is better than using a ChildJVM with ParMapper because it uses less memory.

Essentially, the combination of the two approaches give us the ability to control the number of I/O and computational threads used by the map tasks.

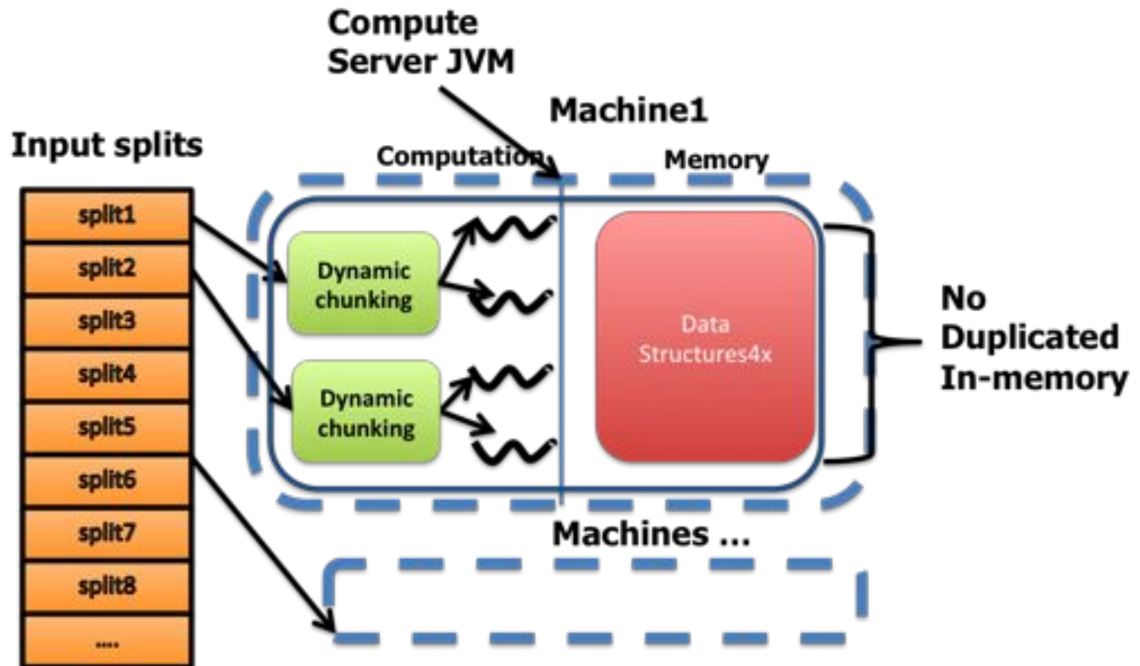

Figure 4.12 : Design and Implementation of the Hybrid Approach

4.3.2 Programming Model

The programming model of the hybrid approach is a combination of the ParMapper and the Compute Server approach. The user extends the ParMapper class and makes sure that there is only one initialization of the shared data structure in the setup method, as shown in Figure 4.13.

```
public class MyMapper extends
    ParMapper<LongWritable,Text,
    Text, LongWritable> {

    private static DataStructure datastruct;


    public void setup(Context context){
      lock.lock();
      if (datastruct not loaded in memory){
       load the datastruct;
      }
      lock.unlock()

    }

    public void map(LongWritable key,
            Text value,
            Contextcontext)  {
               ...
    }
}
```

Figure 4.13 : Hybrid approach for multi-core systems

Chapter 5

Results

This chapter presents our experimental evaluation of the Habanero Hadoop runtime. We studied three widely used applications, KMeans, K Nearest Neighbors and hash join described in Chapter 3. For each application, we demonstrate the improved memory efficiency and throughput running it on the Habanero Hadoop system.

5.1 Experimental Setup

We run our tests using a cluster of five nodes. This cluster consists of four compute nodes and one jobtracker node. The jobtracker node is responsible for hosting the client JVM and jobtracker. It does not execute any map/reduce tasks. Each compute node has two quad-core 2.4 GHz Intel Xeon CPUs and 8 GB memory. We use Java 1.8.0 and Hadoop 1.0.3 to conduct the experiments. All of the experiments are conducted on top of the Hadoop Distributed File System (HDFS). The same 32 MB block size is used for all applications to rule out the impact of block size. Different nodes are connected using an Infiniband network switch.

The hash join application uses three compute nodes and the KMeans application uses all four compute nodes. We believe the scalability of the cluster is not a big issue here because the runtime optimizations will improve the performance of every single compute node. As a result, the memory efficiency and throughput improvement for large-scale inputs should be able to scale as more compute nodes are added to the

cluster.

For the baseline, we used the unmodified Hadoop MapReduce system. The first baseline configuration uses eight map slots in Hadoop MapReduce to keep the CPU fully utilized, since each compute node has eight cores. In this configuration, the heap size limit was set to 1 GB for each Child JVM to simulate a commodity class machine with 8 GB available RAM for eight cores. We also used another baseline configuration that uses four map slots and 2 GB heap size for each Child JVM. The second configuration sacrifices CPU utilization to make more memory available to each map task.

For the Habanero Hadoop system, we evaluated the three approaches described in Chapter 4. We tested different configurations using the sequential ChildJVM with ParMapper, parallel Compute Server with Hadoop Mapper and parallel Compute Server with ParMapper (Hybrid approach).

We evaluated the performance of the baseline and Habanero Hadoop system on three applications KMeans, K Nearest Neighbors and hash join. For each application, we first show a CPU utilization over time graph to demonstrate the compute and I/O intensity of the application. Next, we show a throughput over the in-memory data structure size graph to demonstrate the improved the memory efficiency and throughput of the Habanero Hadoop system. Finally, we have a third graph showing the aggregated heap size of each configuration for different runs with increase in-memory data structure size.

The details of the applications's implementation is described in Chapter 3.

5.2 Application Benchmarks

5.2.1 Hash Join

The first application is hash join. The implementation of the application is described in Chapter 3.

We ran the hash join application with number of map slots set to different values using the original Hadoop MapReduce runtime system. The number of map slots is the maximum number of map tasks that can run on the compute node in parallel. We performed a hash join operation on a 400 MB table with a smaller 200 MB table. The 200 MB table is loaded into the memory of each map task.

Figure 5.1 shows the CPU utilization over time graph for hash join on a single compute node with eight cores. The goal of the graph is to show that hash join is an I/O intensive application. The hash join is a map-only MapReduce job. As a result, the CPU utilization graph only shows the utilization over the map phase.

We first focus on the red line (a single Hadoop map slot) in Figure 5.1. The single map task sees periodic bursts of high CPU utilization. At first, the lines shows a burst to almost 400% CPU utilization. The first burst is because it is loading in a copy of the in-memory data structure. We have data showing that as the CPU utilization reaches its first peak around eight seconds, the memory footprint also increases steadily and stabilizes after the CPU utilization drops. It goes beyond 100% CPU utilization because the JVM is performing garbage collection in parallel. After the first peak, the CPU utilization peaks to close to 100% intermittently because the system is only processing the key-value pairs during the peak. The map task is waiting on I/O to read input key-value pairs during the periods when the CPU utilization is close to 0%. As we can see, almost half of the time of the hash join application is spent on

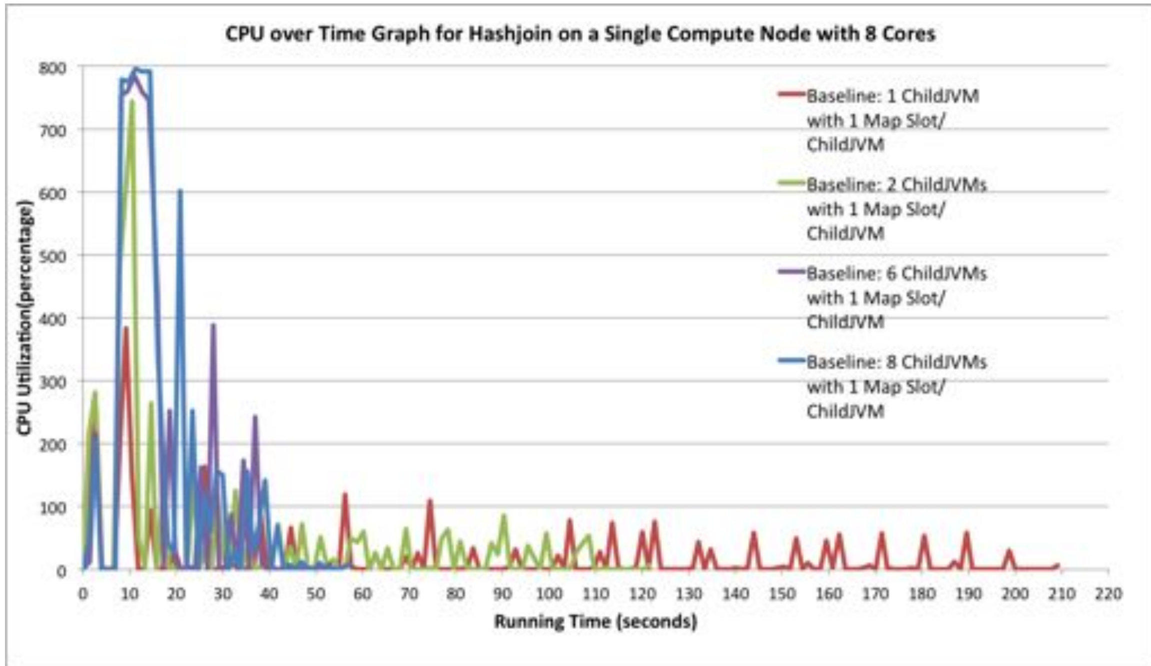

Figure 5.1 : CPU utilization for hash join on a single-node with eight cores

waiting for I/O operations. Thus, the CPU over time data shows that hash join is an I/O-intensive operation and the performance bottleneck is the I/O of the multi-core systems.

Next, we look at the green line which uses two Hadoop map task slots, capable of running two map tasks in parallel. The green line utilizes I/O more effectively by overlapping communication and computation. The overlap is shown as the gaps between peaks in CPU utilization are much shortened. However, the peak CPU utilization after the initial burst is still capped around 100%. The overall running time also shortened to almost half of the red line (using a single map task).

Using six Hadoop map slots (the purple line), the Hadoop MapReduce runtime is able to achieve much better CPU utilization through improved I/O utilization. The peak utilization goes to around 400% even after the initial burst. This runtime can

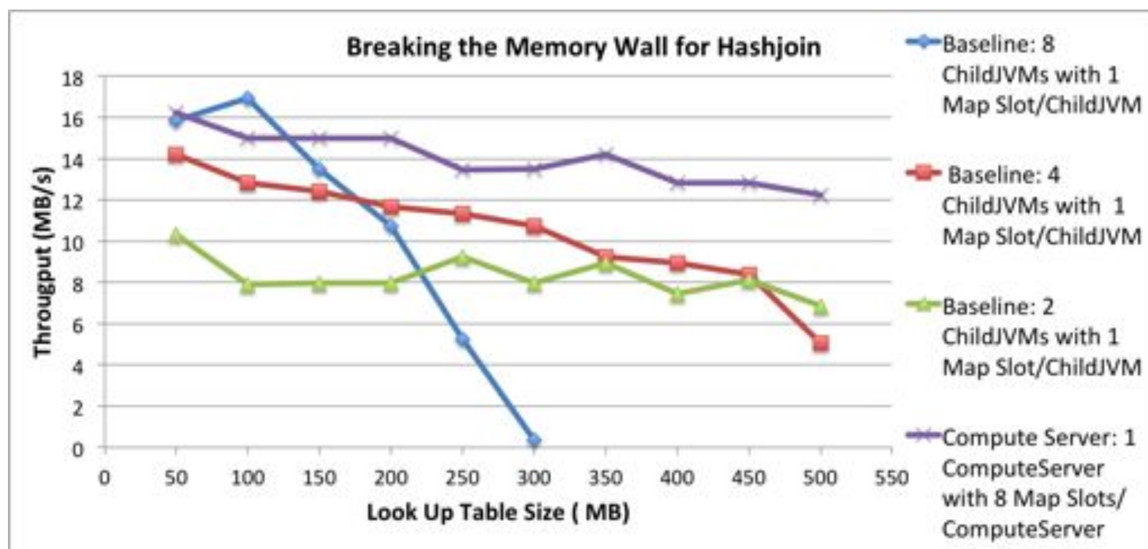

Figure 5.2 : Breaking the memory wall for hash join using the Compute Server on a three-node cluster

use multiple I/O threads in multiple parallel map tasks to saturate the I/O of the multi-core system. The blue line shows that using eight map slots does not make too much of a difference in terms of the performance of the application compared to using six map slots.

Figure 5.2 shows the Compute Server approach can break the memory wall. We ran the experiment in a three compute node cluster with eight cores and 8 GB memory on each node. The x axis represents the size of the lookup table and the y axis represents the throughput of the system as the larger table is streamed through the map tasks. The MB/second measure represents how many megabytes of the larger table have completed the join operation with the smaller lookup table. The larger table that we are using as the input is 800 MB.

When there is enough memory, the throughput of each configuration for hash join should be relatively stable. The reason for the stable throughput is that we

are streaming through a constant size table (800 MB) and for each element in the big table, we are performing a search in the hash table constructed from the smaller lookup table. The search operation is constant time regardless of the lookup table size. The details are explained in Chapter 3.

We first examine the throughput of the baseline configurations. The first baseline configuration (blue) has eight ChildJVMs with 1 map slot per ChildJVM. Since there are a total of 8 GB memory available on a compute node, 1 GB memory is available to each task. The second baseline configuration with four map slots configuration (red) has four ChildJVMs with one map slot per ChildJVM. This configuration utilizes only four of the eight cores but has 2 GB memory available to each task. The baseline configuration with two map slots baseline configuration (green) has 4 GB memory available to each map task.

Since hash join is an I/O intensive application, the throughput improves as we increase the number of I/O threads, in this case, the number of map slots. Moving from the top of the figure to the bottom of the figure, eight ChildJVMs with one map slot per ChildJVM has better throughput than the other two baseline configurations with four map slots and two map slots.

As the size of the lookup table increases, each map task consumes more heap space to keep the lookup table data structure in memory. When the heap memory available to each ChildJVM executing a map task is used up, the garbage collection activity increases significantly. There are two types of garbage collections, a full garbage collection (full GC) and a regular garbage collection (regular GC). A full garbage collection usually takes much longer than a regular garbage collection call. Furthermore, a full GC often stops the execution of the map task, whereas the regular GC can sometimes be done in parallel with the map program. Thus, the full GCs

hurt the performance much more than the regular GCs.

Next, we show that the throughput drop for baseline configuration with eight map slots at around 300 MB is due to a significant increase in garbage collection activity.

Lookup Table Size	2 ChildJVM with 1 Map Slot/ChildJVM	4 ChildJVMs with 1 Map Slot/ChildJVM	8 ChildJVMs with 1 Map Slot/ChildJVM	1 Compute Server with 8 Map Slots/Compute Server
50	0	0	0	0
100	0	0	24	0
150	0	0	74	0
200	0	12	152	0
250	0	29	675	0
300	0	36	14522	0
350	0	44		0
400	6	61		0
450	7	72		0
500	12	232		0

Table 5.1 : Number of total full garbage collections calls on each compute node for hash join

Tables 5.1 and 5.2 show the number of full garbage collection calls and regular garbage collection calls on one compute node in the cluster. We used the "-verbose:gc" flag to collect garbage collection information.

As highlighted in Tables 5.1 and 5.2, the number of full GC calls at 300 MB increased by 20x, regular GC calls increased by 8x for eight ChildJVM with one map slot per ChildJVM. Similarly, at 500 MB the number of full GC calls increased by three times for four ChildJVM with one map slot per ChildJVM. The increase in garbage collection activity corresponds to the drop in throughput for the two configurations. When garbage collection activity takes up more than 99% of the execution time, the JVM crashes and the task fails to finish.

Lookup Table Size	2 Child-JVM with 1 Map Slot/Child-JVM	4 Child-JVMs with 1 Map Slot/Child-JVM	8 Child-JVMs with 1 Map Slot/Child-JVM	1 Compute Server with 8 Map Slots/Compute Server
50	536	704	1009	843
100	540	719	1104	856
150	545	735	1257	853
200	552	780	1426	855
250	558	820	1782	859
300	563	846	14126	868
350	565	889		870
400	589	941		889
450	599	985		885
500	610	1023		892

Table 5.2 : Number of total regular garbage collections calls on each compute node for hash join

On the other hand, the Compute Server approach experiences no significant drop in throughput as the lookup table size increases. The Compute Server approach with eight map slots per Compute Server uses all 8 GB memory without any duplicated copies of the data. As a result, the Compute Server approach has much more memory available to each map task than the baseline configurations. With the improved memory efficiency, the throughput of the Compute Server approach is relatively stable as the lookup table size increases.

In Tables 5.1 and 5.2, we can see that the number of full garbage collection calls stays at 0 and the number of regular garbage collection calls increases slowly from 800 but is capped at 900. These data show that the throughput is stable because garbage collection activity is not a performance bottleneck for hash join using the Compute Server approach for the lookup table sizes we tested.

Apart from pushing back the memory wall, Figure 5.2 also shows that the Compute Server approach with eight map tasks has better throughput than the best available

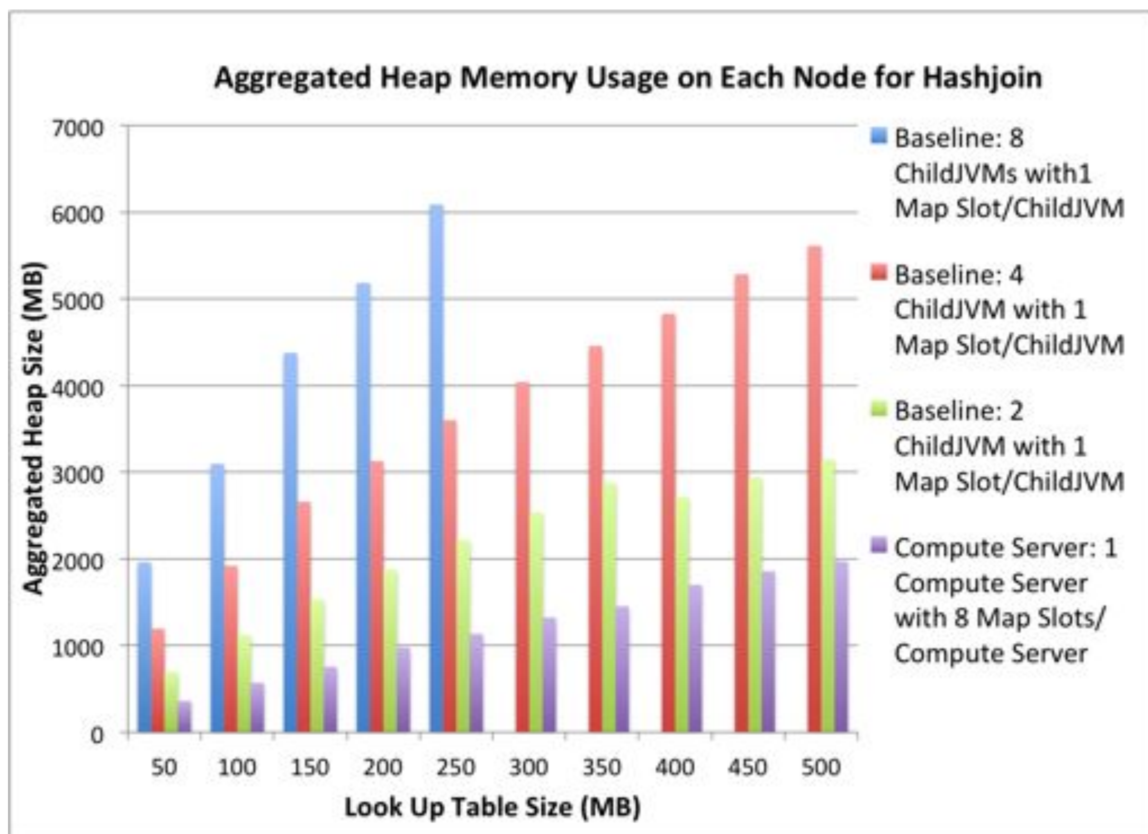

Figure 5.3 : Heap memory size on each compute node for hash join

baseline configuration for larger table sizes. For example, at 300 MB, the Compute Server approach is 50% faster than four ChildJVMs with one map slot per ChildJVM and at 500 MB, the Compute Server improved the throughput by about two times as compared to two ChildJVM with one map slot per ChildJVM.

To show the impact of duplicating the in-memory data structures and the memory savings of the Compute Server, we produced clustered bar charts for the aggregated heap memory size on one compute node for the different configurations in Figure 5.3.

First, we notice that the aggregated heap size of all the configurations is linear in the lookup table size because we load the content of the lookup table text data into

a hash table.

Furthermore, the bar chart shows that duplicating the lookup table results in large heap memory usage. The eight ChildJVMs with one map slot per ChildJVM configuration uses close to two times heap memory more heap memory than the four ChildJVMs with one map slot per ChildJVM because it has twice as many copies of the lookup table stored in memory. Similarly, the heap size of the four ChildJVMs with one map slot per ChildJVM configuration is much larger than that of the two ChildJVMs with two map slots configuration.

The Compute Server approach (purple) has the least aggregated heap memory footprint because different map tasks share only a single copy of the in-memory look up table. This approach reduces the memory footprint of hash join by as much as six times.

So far, we have shown that the Compute Server approach can push back the memory wall, achieve better throughput for large scale problems and improve the memory utilization on each compute node for hash join. Next, we evaluate the performance of ParMapper and the Compute Server with different values for the number of map slots.

In Figure 5.4, we first evaluate the performance overhead of the Compute Server over the original ChildJVM. To do this, we compare the performance of using a ComputeServer with one ParMap slots per Compute Server and a ChildJVM with one ParMap slots per ChildJVM. The throughputs for both configurations are very similar, demonstrating that the Compute Server has little overhead compared to the ChildJVM approach when executing with a single ParMap slot. The throughput of the Compute Server with four ParMap slot per Compute Server (red) is similar to that of four ChildJVMs with one map slot per ChildJVM (blue), showing that a

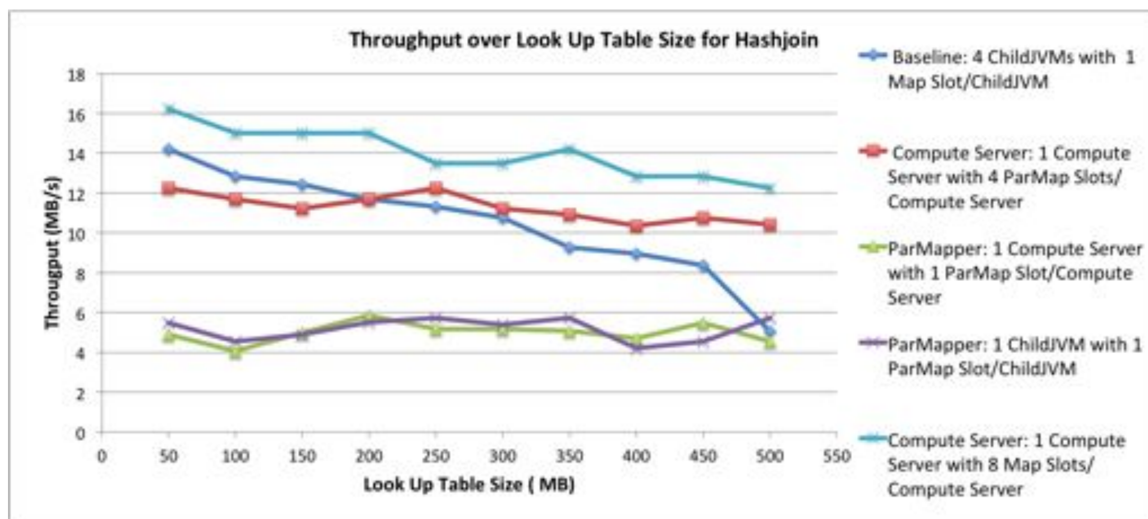

Figure 5.4 : ParMapper and Compute Server's impact on throughput in a three-node cluster

single Compute Server has no significant overhead compare to multiple ChildJVMs for hash join.

Additionally, the comparison between the Compute Server with four ParMap slots per Compute Server (red) and the Compute Server with four Hadoop map slots per Compute Server shows that ParMapper has no performance advantage over the original sequential Hadoop Mapper for hash join. The ParMapper performs poorly because a single I/O thread is unable to generate enough parallel tasks to saturate multiple cores for I/O-intensive applications.

In summary, Figure 5.4 shows that the best configuration for hash join running on eight-core nodes is using the Compute Server with eight map slots per Compute Server. This configuration reduces the heap memory usage by as much as six times and improves the throughput for large lookup table by two times.

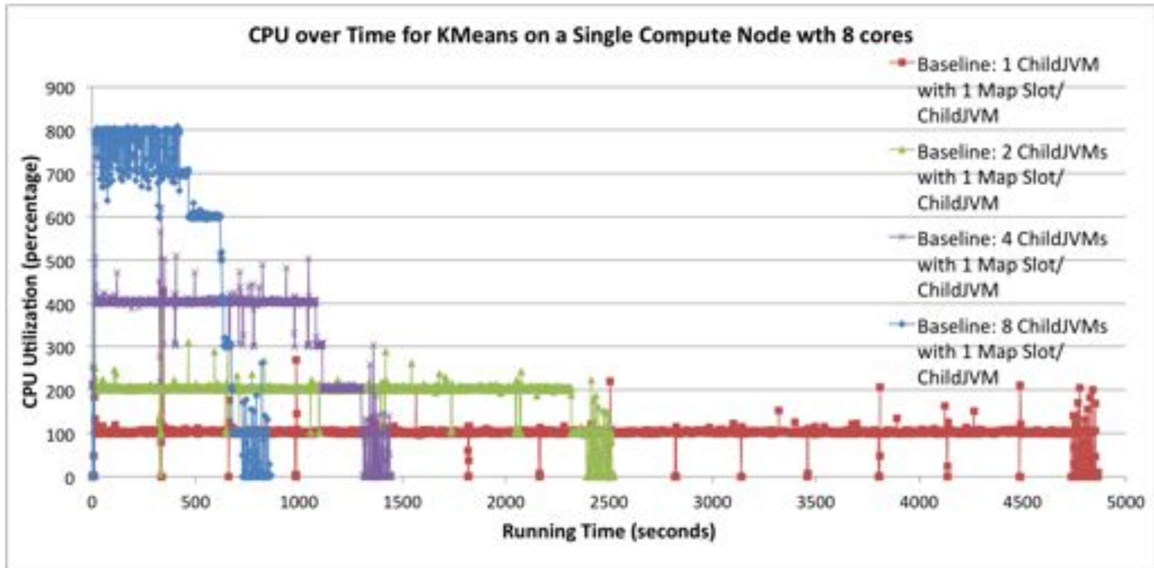

Figure 5.5 : CPU utilization on a single compute node with eight cores for KMeans

5.2.2 KMeans

We first analyze the CPU utilization over time graph in Figure 5.5 to show the performance characteristics of the KMeans application.

The input document size is 55 MB and the cluster data size is 30 MB. The experiment is conducted on a single compute node with eight cores and 8 GB of memory.

The KMeans application has a map phase and a reduce phase. In a single compute node, the shuffle phase is insignificant because there is no transfer of data across different nodes. The map phase has consistent full CPU utilization and the reduce phase has fluctuating CPU utilization. The chart shows that the map phase dominated the time of the execution.

We first focus on the one ChildJVM with one map slot configuration. It almost consistently utilizes the CPU at 100% from 0 to 4700 seconds, because a single map slot can only use a single computation thread. The consistent 100% utilization shows

that the application is compute-intensive in the map phase and a single I/O thread can keep a single core busy. There are a few periodic drops in the CPU utilization due to the the gap between the end of one map task and the start of another map task. From 4700 seconds to 5000 seconds, we can see the CPU utilization fluctuates between 0% and 200% because the single Reduce task running on the core has invoked a lot of parallel garbage collections and the reduce tasks are not as compute-intensive as the map phase.

Then, we evaluate the scalability of the application using more I/O and computation threads. The Baseline two ChildJVMs with one map slot per ChildJVM keeps the two cores busy by achieving a stable 200% CPU utilization. In the same way, the map phase performance almost scales to eight cores. Around 5200 seconds, there is a drop in CPU utilization for eight ChildJVMs and one map slot per ChildJVM configuration because there are not eight map tasks left to keep the CPU fully utilized.

In summary, Figure 5.5 demonstrates that KMeans' execution time is dominated by the compute-intensive map phase. The performance scales as more computation threads are used by the Hadoop MapReduce runtime.

Next we show that the Habanero Hadoop runtime can break the memory wall for the KMeans application in Figure 5.6.

There are 200 MB of document vector data with about 4 KB per document vector. The total number of documents is around 51200. The document vectors are generated from the "20 newsgroups" data, ensuring that the document vectors are representative of real news documents.

Since the application is streaming the documents through the cluster data, the y axis is the throughput of the system represented by the number of documents processed per second. At each iteration, all of the documents in the database are

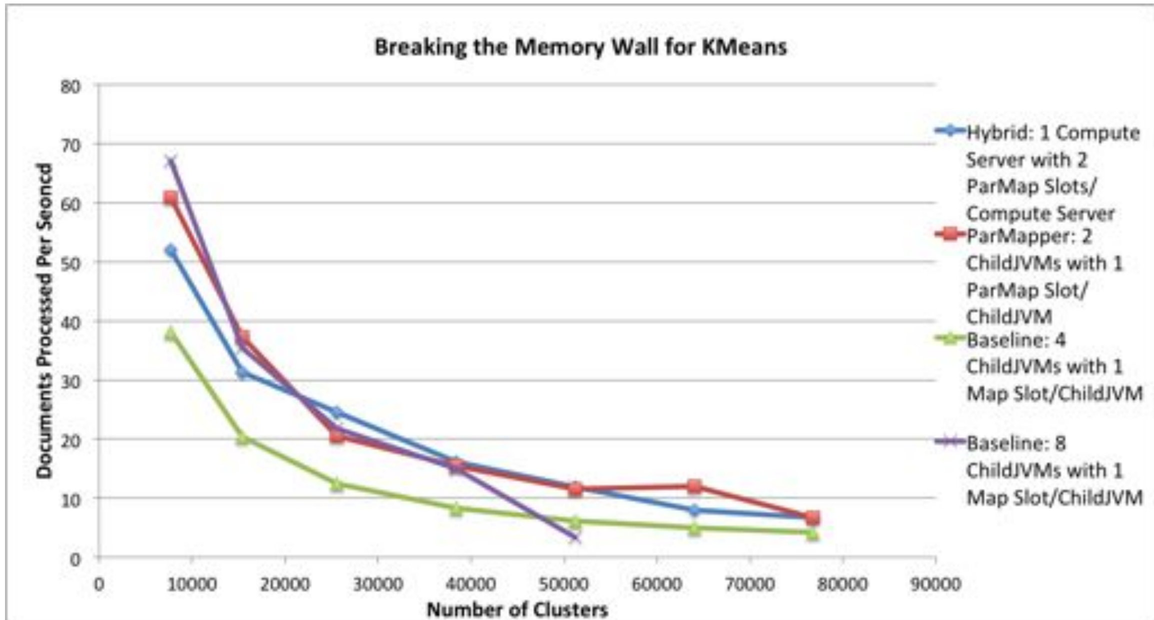

Figure 5.6 : Breaking the memory wall for KMeans on a four-node cluster

compared against topic centroids and assigned to different clusters. For each document processed by the map function, we compare it against all of the clusters to find the cluster that is most similar to the content of the document. As a result, the throughput decreases as we increase the number of clusters because it takes longer to calculate the similarities with all the clusters. As we described in Chapter 3, the sequential computation complexity is $O(|n| \times |k|)$ where n is the number of documents and k is the number of clusters. The x axis is the number of topics, which is linear in the size of the in-memory data structure.

The memory wall can be observed in the baseline eight ChildJVMs with one map slot per ChildJVM configuration. As shown in Figure 5.6, at 51200 clusters, the throughput drops out of the inverse curve. In contrast, the baseline four ChildJVMs with one map slot per ChildJVM stays on the inverse curve because more memory

is available to each map task. The additional memory available extends the memory wall for the baseline four ChildJVMs configuration.

The drop in throughput at 51200 clusters is due to increased garbage collection activity. In Tables 5.3 and 5.4, we can see a 380 times increase in the number of full GCs, two times increase in the number of regular GCs for eight ChildJVMs with one map slot per ChildJVM configuration at 51200 clusters.

Number of Clusters	1 Compute Server with 2 ParMap Slot/Compute Server	2 ChildJVMs with 1 ParMap Slot/Child-JVM	4 ChildJVMs with 1 Map Slot/Child-JVM	8 ChildJVMs with 1 Map Slot/Child-JVM
7680	0	0	3	26
15360	1	4	20	30
25600	3	11	24	219
38400	7	18	94	409
51200	13	27	211	155584
64000	15	41	312	
76800	21	67	411	

Table 5.3 : Number of total full garbage collections calls on each compute node for KMeans

In contrast, we can see that the one Compute Server with two ParMap slots per Compute Server and two ChildJVMs with one ParMap slot per ChildJVM configurations perform a relatively small number of full GC and regular GC calls. The one Compute Server with two ParMap slots per Compute Server configuration performs less than 21 full garbage collection calls. As a result, the throughput of the two configurations are still on the inverse curve when processing large numbers of clusters.

The ParMapper and Hybrid approaches can fully utilize the CPU resources with better memory efficiency. As a result, the blue and red lines stay on the inverse

Number of Clusters	1 Compute Server with 2 ParMap Slot/Compute Server	2 ChildJVMs with 1 ParMap Slot/Child-JVM	4 ChildJVMs with 1 Map Slot/Child-JVM	8 ChildJVMs with 1 Map Slot/Child-JVM
7680	6187	3923	41968	39836
15360	10372	6470	83911	67017
25600	17685	11019	47881	46534
38400	25573	17798	73443	72062
51200	31788	21086	50949	174138
64000	42773	25364	62120	
76800	50551	32695	75824	

Table 5.4 : Number of total garbage collections calls on each compute node for KMeans

curve. At 51200 clusters, both the red and blue lines avoided the memory wall. The performance of the two configurations scaled to 76800 clusters. For cluster sizes larger than 51200, the Hybrid and ParMapper configurations improve the throughput of the baseline with four ChildJVMs and one map slot per ChildJVM by two times.

For the most part, there is no significant performance difference using two Child-JVMs with one ParMap slot per ChildJVM and one Compute Server two ParMap slots per Compute Server configurations. However, there is an unexpected throughput improvement at around 64000 clusters for two ChildJVMs with one ParMap slot per ChildJVM over one Compute Server and two ParMap slots per Compute Server configuration.

Finally, we show the heap memory size in Figure 5.7 for the different configurations. The heap size data are collected during the test runs shown in Figure 5.6. The total memory is estimated by averaging the heap sizes at different times of the map phase.

We first note that the configuration using eight ChildJVMs with eight Map Slots

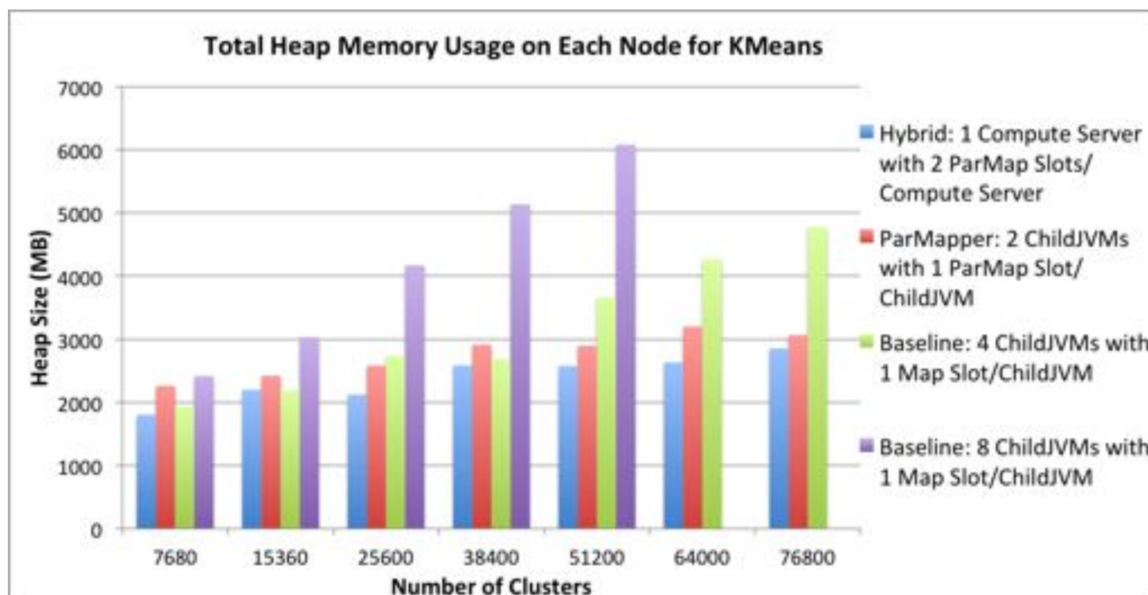

Figure 5.7 : aggregated heap memory size on each compute node for KMeans

uses the most heap memory. The large memory footprint is because there are eight map tasks running in parallel, duplicating the cluster data eight times. In addition, there are other in-memory data structures associated with each map task such as the partial sum for calculating the new cluster data. As a result, running eight map tasks in parallel results in a large memory footprint. Reducing the total number of map slots for baseline configurations to four gives a good reduction in aggregated heap memory usage, but reduces CPU utilization.

The ParMapper (red) and Hybrid (blue) approaches result in significantly less heap memory usage. The Hybrid approach (blue) uses less heap memory than the ParMapper approach (red) because the Compute Server eliminates duplicated in-memory data structures. In addition, ParMapper uses more memory than the Hadoop Mapper. This is because the key-value pairs are loaded into separate buffers to enable parallel execution, whereas the original Hadoop mapper reuses the memory buffer for

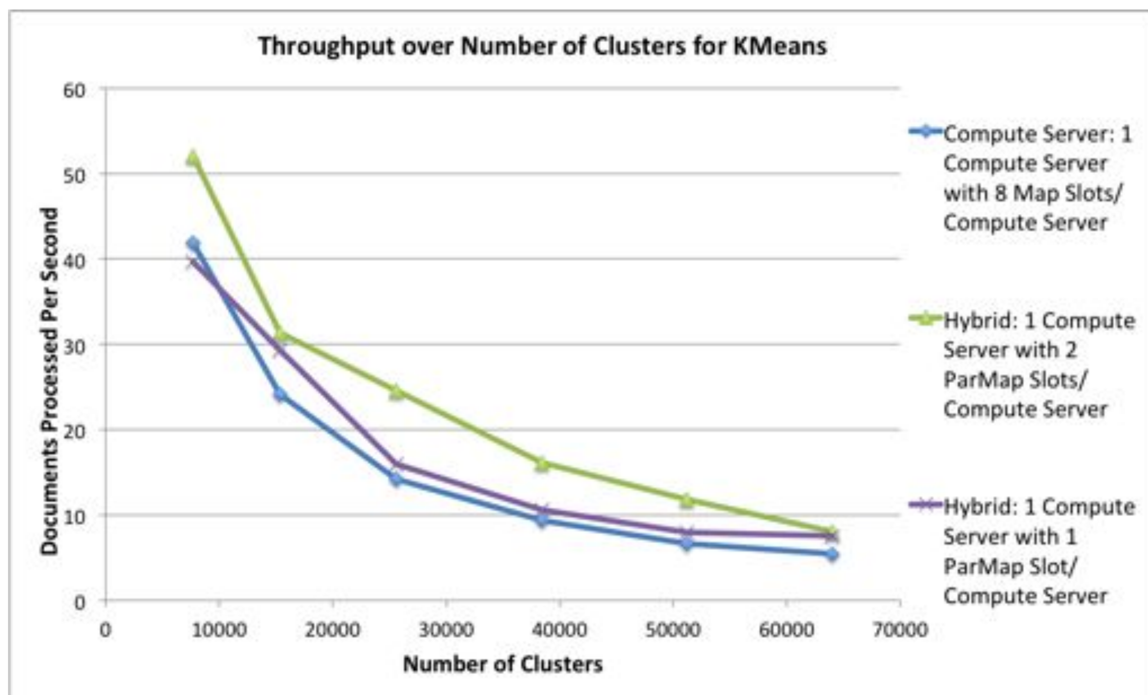

Figure 5.8 : ParMapper and Compute Server's impact on throughput on a four-node cluster for KMeans

the current key-value pair to store the next key-value pair.

Figure 5.8 shows that a single ParMap slot can not keep the CPU fully utilized. The Hybrid approach using one Compute Server with one ParMap slot per Compute Server (purple) has significantly lower throughput than that of the Hybrid approach with two ParMap slots per Compute Server. The Hybrid approach with two ParMap slots per Compute Server can better overlap the computation of one map task with the I/O activities of the other map tasks, achieving high CPU utilization.

Next, we analyze the impact on memory utilization of increasing the number of ParMap and Map slots per Compute Server in Figure 5.9. The Compute Server approach with eight map slots per Compute Server (blue) has the largest aggregated heap memory usage. The Hybrid approach with two ParMap slots per Compute

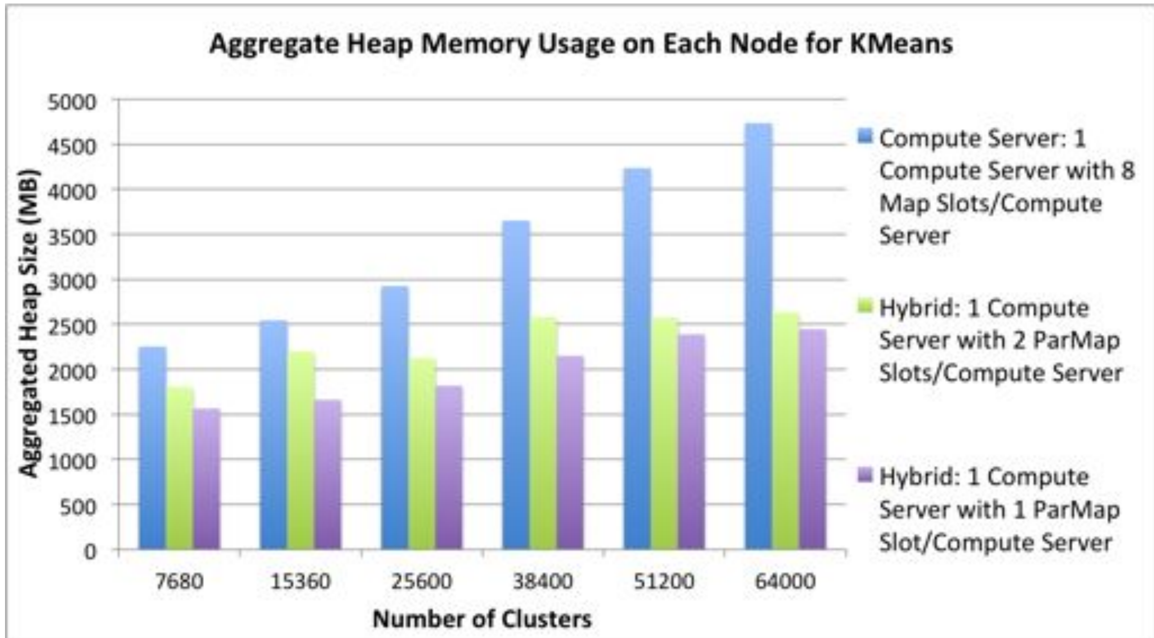

Figure 5.9 : Aggregated heap memory size on each compute node for KMeans

Server (green) has a larger memory footprint than the Hybrid approach with one ParMap slots per Compute Server (purple). Figure 5.9 shows that the aggregated heap memory usage increases as the number of ParMap/Map slots increases. The increase in memory usage is because each ParMap/Map task has an in-memory local partial sum data structure that is proportional to the number of clusters. As a result, even with the Compute Server approach, more ParMap/Map tasks slots per Compute Server leads to a larger memory footprint.

In summary, we show that the most efficient Hybrid approach for KMeans runs a small number of ParMap tasks that keeps all eight cores busy. In this case, two ParMap slots can achieve full CPU utilization with the least amount of required memory. The Hybrid approach with two ParMap slots is also the most efficient overall approach for running the KMeans application. The Hybrid approach running

with our experimental set up can reduce the heap memory usage by two times and improve the throughput by two times when generating a large number of clusters.

5.2.3 K Nearest Neighbors

The performance characteristics of K Nearest Neighbors (KNN) are very similar to those of the KMeans application, as described in Chapter 3. The best approach for KMeans with one Compute Server with two ParMap slots per Compute Server is also the most efficient approach for KNN.

In this section, we show that the one Compute Server with two ParMap slots per Compute Server approach can push back the point at which we hit the memory wall and reduce the memory footprint of the application.

In Figure 5.10, the baseline configuration running eight ChildJVMs with one map slot per ChildJVM stays on the inverse curve up to 76800 documents. This curve shows that the CPUs are efficiently utilized up to 76800 documents. At 89600 documents, the throughput of the baseline configuration running 8 ChildJVMs drops out of the inverse curve, showing that the memory wall for the configuration is around 89600 documents.

On the other hand, the Hybrid approach with two ParMap slots per Compute Server and the ParMapper approach with two ChildJVMs with one map slot per ChildJVM stays on the inverse curve even when processing 102400 documents. This shows that the two configurations can push back the memory wall. Moreover, the throughputs of the Hybrid and ParMapper approaches are actually better than the Baseline with eight map slots before the memory wall because ParMapper can break map tasks into smaller subtasks, improving the load balance across multiple cores.

Table 5.5 shows that at 89600 documents, the Baseline configuration with eight

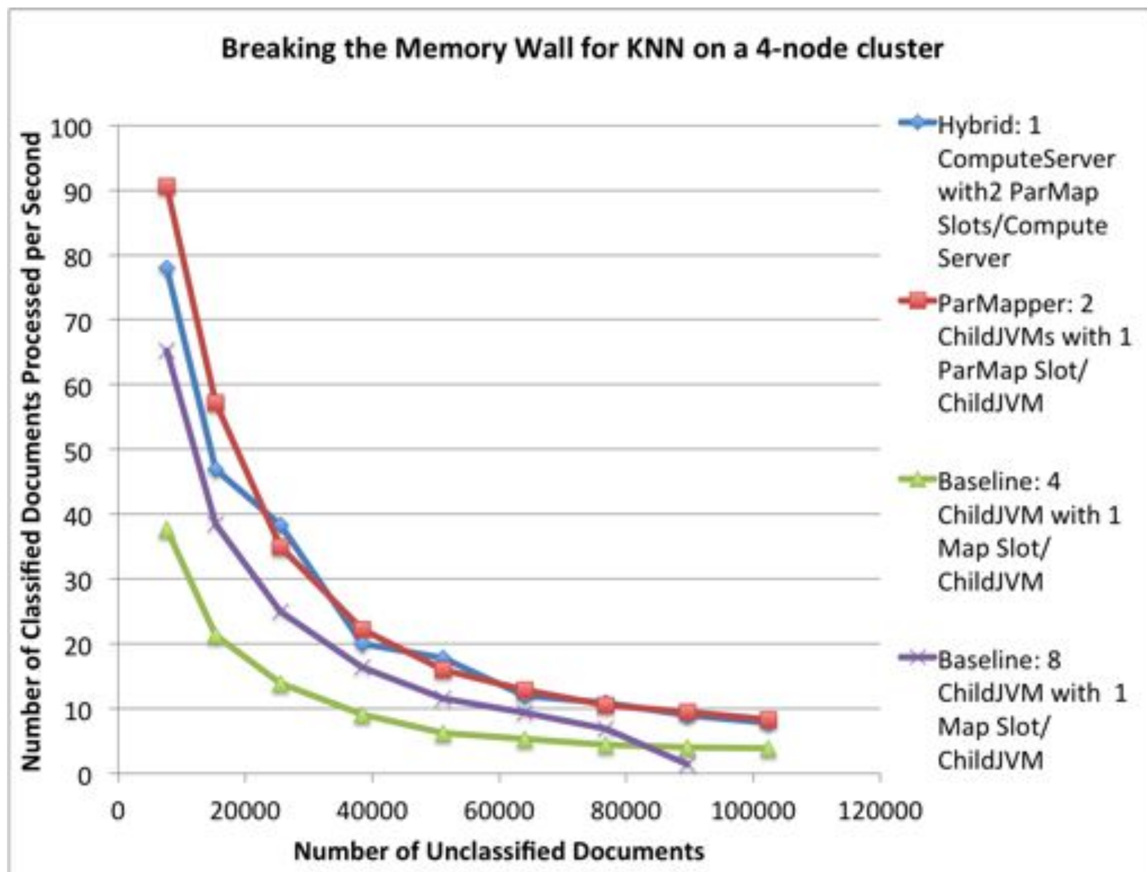

Figure 5.10 : Breaking the memory wall for KNN on a four-node cluster

ChildJVMs performs a 783 times more full GCs. The increase in full GC activity is responsible for the drop in throughput of the configuration. In contrast, the numbers of full GCs for either the Hybrid and ParMapper approach is very small.

Figure 5.11 shows that the Hybrid and ParMapper approaches reduce the memory footprint of the KNN application on each compute node. The baseline with eight ChildJVMs (purple) has the largest memory footprint because it duplicates the data structures eight times. The baseline with four ChildJVMs configuration has less aggregated heap memory usage because the in-memory data structure is duplicated

Number of Clusters	1 Compute Server with 2 ParMap Slot/Compute Server	2 ChildJVMs with 1 ParMap Slot/Child-JVM	4 ChildJVMs with 1 Map Slot/Child-JVM	8 ChildJVMs with 1 Map Slot/Child-JVM
7680	0	0	0	0
15360	0	0	1	23
25600	0	0	7	21
38400	1	1	25	178
51200	0	5	38	327
64000	2	8	43	360
76800	2	8	59	483
89600	4	10	87	378578

Table 5.5 : Number of total full garbage collections calls on each compute node for KNN

only four times. The ParMapper approach (red) uses almost the same amount of heap memory as the baseline with four ChildJVMs because ParMap tasks use more memory than the regular Hadoop map tasks. The Hybrid approach (blue) keeps only a single copy of the data structure in-memory, achieving the best memory efficiency in all four configurations.

Using our experimental set up with eight cores per node, the Hybrid approach with two ParMap slots per Compute Server reduces the heap memory usage of KNN by three times and improves the throughput for a large number of documents by two times.

In the big data age, the performance of MapReduce runtimes for large-scale data analytics applications largely depends on efficient utilization of the memory resources in multi-core systems. In this chapter, we show that the Habanero Hadoop system can significantly improve the memory efficiency and throughput of popular data analytics applications.

Using our set up with eight-core nodes, the Compute Server approach reduces the

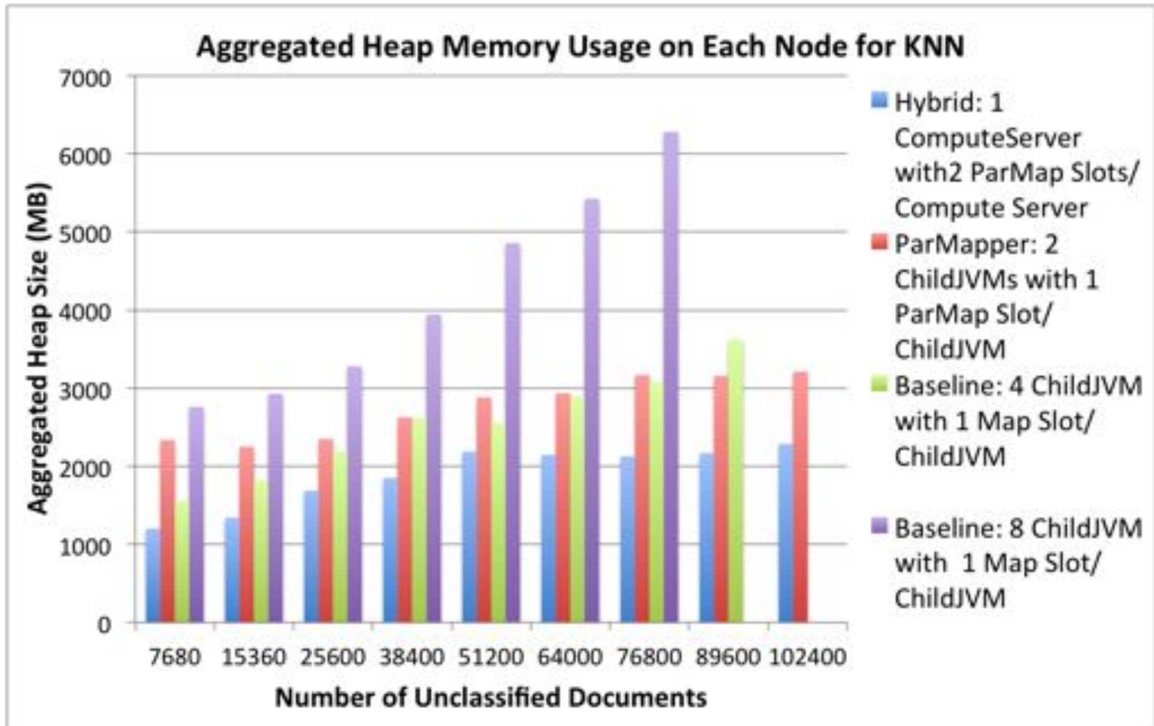

Figure 5.11 : Aggregated heap memory size on each compute node for KNN

heap memory usage by as much as six times and improves the throughput by two times for hash join. For KMeans, the Hybrid approach can reduce the heap memory usage by two times and increase the throughput by two times when generating a large number of clusters. For KNN, the Hybrid approach reduces the heap memory usage of KNN by three times and improves the throughput for a large number of documents by two times.

The results also show that the most efficient approaches for I/O-intensive applications, such as hash join, and compute-intensive applications, such as KMeans, are often different.

Chapter 6

Related Work

6.1 MapReduce Runtime Systems

Parallel programming has always been a difficult task for mainstream programmers who want to take advantage of the parallell processing power in multi-core machines and in clusters with thousands of machines. As a result, writing correct, efficient and scalable programs traditionally falls on the shoulders of a small group of experts.

To enable average programmers to harness the power of parallel and distributed computing in a large cluster environment, Google proposed the MapReduce programming model [12]. It allows users to make high level specifications about concurrency and locality, and provides an efficient runtime system that can automatically handle task distribution, execution and fault tolerance. The MapReduce model is rooted in the idea of using functional scan primitives or paralel prefix schemes as a tool for writing parallel programs as first explored by Blelloch [13] and Ladner et al. [14].

The MapReduce model is a very restrictive model. It divides up task execution into two phases, the Map and the Reduce phase. The computations are essentially stateless and can be easily parallelized without much communication between tasks. It is very suitable to a large number applications, such as Inverted Index and Grep, and it can achieve scalable performance at a reasonable cost. To enable fault tolerance and data locality, the runtime system is built on top of the Google File System (Ghemawat et al. [15]).

The success of the MapReduce model at Google has led to the open source project Hadoop MapReduce [3]. The system is implemented using Java based on the idea of Google's MapReduce paper [2]. The system also provides an easy to use interface with an efficient, scalable and reliable run time system. The Hadoop MapReduce system is built on top of its own distributed file system, the Hadoop File System [16].

The Google and Hadoop MapReduce model (cluster MapReduce model) both employ process level parallelism by assigning each task to a process. For example, Hadoop MapReduce spawns a separate JVM for each map and reduce task. Since different processes have no shared memory space, the tasks are executed in a fully distributed memory model. This thesis aims to integrate some shared memory space across different tasks on the same compute node. In my optimized model, a process can execute several tasks.

The popularity of the cluster MapReduce model has prompted others to try out the MapReduce programming model in a shared memory multi-core system.

Phoenix [17] is the first attempt to build a MapReduce runtime system that allows programmers to efficiently utilize multi-core systems. Similar to my optimizations, the Phoenix system uses threads to spawn parallel map/reduce tasks on the same machine. It also used shared-memory buffers for low overhead communication. The runtime provides tolerance for task failures and is optimized for multi-core systems by selecting the right task unit size. It showed that an optimized MapReduce runtime could achieve scalable performance for a number of applications in a shared-memory system.

The Phoenix system is built for a single machine. The design of the system was not concerned with scaling the performance across a large number of machines. On the other hand, the Habanero Hadoop system supports execution on a large number

of machines.

The Habanero Hadoop system developed for this thesis implemented several optimizations mentioned in the Phoenix paper such as key-value pair prefetching and a dynamic framework that discovers the best input size for each task. Overall, the Phoenix system focuses on maximizing the CPU resources of the shared memory system whereas this work focused on efficient utilization of the memory resources. The Phoenix implementation is not concerned with duplicating in-memory data structures across map tasks. But this work focuses on eliminating the duplicated data structures

Further attempts to improve shared memory MapReduce on large-scale NUMA machines have been explored in Phoenix Rebirth systems [18]. The original Phoenix system targeted 24 to 32 hardware threads, where the optimized Phoenix system proposed in the paper can scale up to 256 hardware threads. However, this thesis focused on commodity computers that are small scale multicore systems, whereas the optimized Phoneix Rebirth system can scale up to 128 cores. As a result, many of the optimizations outlined in the paper are not very useful for this thesis.

Other MapReduce runtime optimizations for multi-core has been explored by Metis [19] and TiledMapReduce [20].

Metis [19] augmented the Phoenix work by focusing on designing efficient data structures to store intermediate key value pairs. The new proposed data structures support an efficient global, parallel sort/group-by function for the intermediate key value pairs, which can speed up MapReduce applications. My optimization uses the original shuffle and sort mechanism in the Hadoop MapReduce system to maintain the cluster scalability of the runtime.

Tiled Map Reduce (TMR) [20] is similar to this thesis because it also focused on significantly reducing the memory footprint of the MapReduce applications. However

it takes a different approach to achieving improved memory efficiency by dividing up a big MapReduce job into a number of independent sub-jobs and reusing the intermediate data structures among the sub-jobs. TMR optimizes MapReduce mainly at the programming model level by limiting the data to be processed in each MapReduce job. In contrast, this thesis reduces the memory footprint through improved runtime system design.

Phoenix, Metis and TMR systems [17, 18, 20, 19] all looked into applying MapReduce in a single multi-core machine. On the other hand, Habanero Hadoop focuses on creating a shared memory space for Map tasks running on the same compute node and at the same time maintains the ability to run the application on a large number of machines.

All the optimizations implemented in the Habanero Hadoop system, including creating a shared memory space between map tasks, reduce the memory footprint of certain popular MapReduce data analytics applications.

In the next section, we look at previous work on optimizing data analytics applications in MapReduce runtimes. Some of the work motivates our optimizations.

6.2 Data Analytics Using MapReduce

A study of using MapReduce for popular data analytics applications has been done by Chengtao Chu et al. [21]. This paper showed that popular machine learning applications, such as KMeans and Logistic Regression that fit the Statistical Query model could be efficiently implemented in the MapReduce model. This paper's performance results show that the parallel versions of these applications can achieve good performance on a single multi-core system.

A natural extension of Chengtao Chu et al. [21]'s work would be to achieve

good performance for the MapReduce machine learning applications across multiple machines. Gillick et al. [11] used the distributed Hadoop MapReduce [3] in a 80 node cluster to perform some machine learning applications. They classified MapReduce applications into a few categories: Single Pass Learning, Iterative Learning and Query-based Learning with Distance Metrics. Our work improved the performance of two applications from these categories, KMeans and K Nearest Neighbors. The Gillick et al. paper [11] inspired the Apache Mahout [7] project, which manages a collection of optimized Hadoop map reduce machine learning algorithms. Zhao et al. [6] put together an implementation of KMeans that is widely used.

Our work improves the performance of the data analytics applications implemented following the design described in Chengtao Chu et al. [21] and Gillick et al. [11].

The findings in Gillick et al. [11] motivates our work. One major issue identified by their study is that Hadoop MapReduce lacks the ability to support efficient access to the common data used in the machine learning applications. The common data are usually model parameters or query examples. The paper noted explicitly that because each map task runs in its own virtual machine, many read only common data structures are duplicated. The duplication resulted in an overhead of efficient sharing of the data structures. The paper called for a shared memory model on each compute node to eliminate the duplication. This is the exact problem that motivated the Habanero Hadoop's ParMapper and Compute Server approaches. ParMapper exploits parallelism within a single map task and the Compute Server creates a shared memory space among map tasks in the same compute node.

Several runtime systems have been proposed for improving iterative MapReduce applications [22, 23, 24]. Spark [22] uses Resilient Distributed Datasets (RDDs) to

keep intermediate data structures in memory for successive iterations of MapReduce jobs. This approach avoids redundant disk I/O operations and provides good fault tolerance guarantees. All of these systems focused on reducing the latency between multiple iterations. However, this thesis focuses on optimizing the performance in a single iteration.

6.3 Hash Join Using MapReduce Systems

Improved memory efficiency can also benefit table join applications. The paper by Blanas et al. [5] discussed various approaches to table joins using MapReduce. This paper also evaluated the performance of repartition join, broadcast join, semi-join and per-split Semi-Join. In many cases, the most efficient MapReduce join performs the join operations exlusively in the map phase. It is implemented as broadcast join by Blanas et al. [5], Fragmented Replicated Join [4] in Pig [8] and MapSide join [10] in Hive [9]. The optimized Habanero Hadoop runtime focuses on improving the performance of the map side hash join.

Chapter 7

Conclusions and Future Work

We are living in the age of Big Data. The amount of data that we can collect and store explodes everyday with more gene sequence data available, more photos uploaded and more news articles and blogs posted. It is apparent that we need new software technologies to process these unprecedented amount of data.

To process these large amounts of data, the MapReduce model emerged as a widely adopted programming model for performing computations in large scale clusters. The MapReduce runtimes provide an easy to program interface and an efficient, scalable and fault tolerant runtime system.

However, when the size of the problem increases, sometimes it is not enough to add more machines into a cluster. In many popular applications, the amount of memory available to each node is an important factor in solving large scale problems. For example, in hash join applications that join two tables, the lookup table is often loaded into the memory of each map task. The more memory available to each map task, the larger the lookup table can be. Another example is clustering algorithms, such as KMeans, where the cluster data needs to be kept in memory. Lastly, for many single pass learning applications such as K Nearest Neighbor, the query parameters often need to be stored in the memory.

In this thesis, we first performed a detailed evaluation of the performance characteristics of the above applications, including the computation intensity, memory usage and scalability on the Hadoop MapReduce runtime. We identified the mem-

ory bottlenecks and source of the inefficiency in the Hadoop MapReduce applications. Duplicated copies of data are kept in memory and a large number of map tasks create large memory overhead.

This thesis introduced an optimized runtime that can significantly improve the memory efficiency of each compute node in the cluster, allowing bigger problems to be solved. To do this, we explored two different approaches.

The first approach is creating an efficient ParMapper that uses multi-threading within a single map task with minimum modifications to the existing Hadoop MapReduce programming model. The ParMapper reduces the number of map tasks needed to utilize multiple cores by using a single I/O thread to feed input key-value pairs to multiple computation threads. We have shown that the ParMapper approach can significantly reduce the number of map tasks needed to utilize eight cores for compute intensive applications such as KMeans and KNN. In this way, we reduce the duplication of in-memory data structures and the overhead associated with a large number of map tasks.

The experimental results on eight core nodes show that the ParMapper approach can reduce the memory footprint of KMeans and KNN by as much as two times and improve the applications' throughput by two times for large cluster sizes. Different set ups with different number of cores might achieve different performance improvements.

However, the ParMapper approach achieves no better performance than Hadoop Mapper for I/O intensive applications such as hash join because the single I/O thread is insufficient to utilize the computation threads.

As a result, we developed the Compute Server approach that runs multiple map tasks in the same JVM to create a shared memory space between different map tasks running in the same JVM. The Compute Server can utilize multiple I/O threads from

multiple map tasks with minimum memory overhead.

Our results show that the Compute Server approach reduces the memory footprint for hash join by six times and achieves two times throughput improvement for lookup tables greater than 400 MB in a four-node cluster with the set up we described in Chapter 5.

To further improve the memory efficiency of KMeans and KNN, we explored the Hybrid approach that runs ParMappers in the Compute Server. The Hybrid approach can achieve good CPU utilization with minimum memory overhead for compute-intensive applications.

7.1 Future Work

We observed that for different applications, the best approach to reduce memory footprint without incurring any penalty to the CPU utilization is different. For hash join, the best configuration uses one Compute Server with eight map tasks per Compute Server to more fully utilize the I/O capability of the system. For KMeans, the best configuration uses a Hybrid approach that runs two ParMap tasks in a Compute Server.

The next step for this research is to automate the process of selecting the best configuration. We need to set up performance counters to measure the I/O and computation characteristics of the application at runtime. Using the information collected, the runtime automatically chooses the best configuration.

Currently, we have to disable speculative execution when using the Compute Server because the Compute Server can not handle a task being killed by tasktracker. When the runtime kills a map task, it kills all the tasks executing in the Compute Server by sending a KILL signal to the Compute Server JVM process. In the process,

other map tasks that are not supposed to be killed are also purged. We need to implement a design for the runtime to safely kill a task running in a Compute Server.

Bibliography

[1] L. A. Barroso, J. Dean, and U. Hölzle, "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro*, vol. 23, pp. 22–28, Mar. 2003.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[3] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[4] "Fragment Replicated Join in Pig." http://wiki.apache.org/pig/PigFRJoin, 2009.

[5] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A Comparison of Join Algorithms for Log Processing in MapReduce," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 975–986, ACM, 2010.

[6] W. Zhao, H. Ma, and Q. He, "Parallel K-Means Clustering Based on MapReduce," in *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, (Berlin, Heidelberg), pp. 674–679, Springer-Verlag, 2009.

[7] R. Anil, S. Owen, T. Dunning, and E. Friedman, *Mahout in Action*. Manning Publications Co. Sound View Ct. 3B Greenwich, CT 06830: Manning Publications, 2010.

[8] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of MapReduce: the Pig experience," *Proc. VLDB Endow.*, vol. 2, pp. 1414–1425, Aug. 2009.

[9] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A Warehousing Solution over a MapReduce Framework," *Proc. VLDB Endow.*, vol. 2, pp. 1626–1629, Aug. 2009.

[10] L. Tang, "Hive Join Optimization." `https://www.facebook.com/notes/facebook-engineering/join-optimization-in-apache-hive/470667928919`, 2010.

[11] D. Gillick, A. Faria, and J. DeNero, "MapReduce: Distributed computing for machine learning." CS 262A class project report, Department of Electrical Engineering and Computer Sciences, University of California - Berkeley, December 18, 2006. `http://www1.icsi.berkeley.edu/~arlo/publications/gillick_cs262a_proj.pdf`.

[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM*, vol. 27, pp. 831–838, Oct. 1980.

[14] G. E. Blelloch, "Scans As Primitive Parallel Operations," *IEEE Trans. Comput.*, vol. 38, pp. 1526–1538, Nov. 1989.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*,

SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2010.

[17] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, (Washington, DC, USA), pp. 13–24, IEEE Computer Society, 2007.

[18] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 198–207, IEEE Computer Society, 2009.

[19] Y. Mao, R. Morris, and M. F. Kaashoek, "Optimizing MapReduce for Multicore Architectures," *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, no. MIT-CSAIL-TR-2010-020, 2010.

[20] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 523–534, ACM, 2010.

[21] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y.

Ng, "MapReduce for Machine Learning on Multicore," in *Advances in Neural Information Processing Systems 19* (B. Schölkopf, J. Platt, and T. Hoffman, eds.), pp. 281–288, 2006.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[23] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), pp. 810–818, ACM, 2010.

[24] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," *Proc. VLDB Endow.*, vol. 3, pp. 285–296, Sept. 2010.