

**An Analysis of Parallel  
Programming Techniques for Data  
Intensive Computation**

by

**Anuradharthi Thiruvankata Ramani**

B.E., Bharathiar University, TamilNadu 2002

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2009

APPROVAL PAGE

Master of Science Thesis

An Analysis of Parallel Programming Techniques for Data Intensive Computation

Presented by

Anuradharthi Thiruvengkata Ramani, B.E.

Major Advisor

John A. Chandy

Associate Advisor

Yunsi Fei

Associate Advisor

Chun-Hsi Huang

University of Connecticut  
2009

## Acknowledgements

I would like to thank and acknowledge my advisor, *Dr. John A. Chandy* for his constant faith in me throughout the long, challenging, yet exciting work on this thesis. I am grateful to his patience, openness to ideas, discussions and ease to work with. I definitely couldn't have done this without his help and guidance. I am thankful to *Sumit ,Janardhan* and *Tina*, my co-workers at SNSL, for their cooperation and also for sharing their ideas with me. Last and most importantly, I would like to thank my parents, my dear husband and co-worker *Ajith*, and my wonderful daughter *Saathvika* for their constant support and encouragement in all my endeavors. I am forever grateful to their love and understanding that has made this thesis happen.

# Contents

<b>List of Tables</b>		<b>vii</b>
<b>List of Figures</b>		<b>viii</b>
<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Parallel Computing . . . . .	3
1.2.1	Architecture Approaches . . . . .	4
1.2.2	Parallel Programming Techniques . . . . .	7
1.3	Data Intensive Application . . . . .	10
1.3.1	Word Count . . . . .	11
1.3.2	BLAT . . . . .	12
1.3.3	Other Examples . . . . .	14
1.4	Outline . . . . .	15
<b>2</b>	<b>Distributed Computation Programming Analysis</b>	<b>16</b>
2.1	Introduction . . . . .	16
2.2	Message Passing Interface - An Overview . . . . .	16

2.3	MapReduce - An Architecture Overview . . . . .	22
2.3.1	Input Reader . . . . .	27
2.3.2	Mapper . . . . .	27
2.3.3	Combiner . . . . .	27
2.3.4	Partitioner . . . . .	28
2.3.5	Reducer . . . . .	28
2.3.6	Output Writer . . . . .	28
2.3.7	GFS . . . . .	28
2.4	Parallel Programming for Data Intensive Computation . . . . .	30
2.4.1	Word Count Specification . . . . .	30
2.4.2	BLAT Specification . . . . .	31
2.5	Summary . . . . .	34
<b>3</b>	<b>Comparison of Message Passing and MapReduce programming</b>	<b>35</b>
3.1	Introduction . . . . .	35
3.2	Characteristics . . . . .	36
3.2.1	Communication . . . . .	36
3.2.2	Partitioning . . . . .	37
3.2.3	Synchronization . . . . .	40
3.2.4	Fault Tolerance . . . . .	40
3.2.5	Programming Languages . . . . .	41
3.3	Programming Styles . . . . .	42
3.4	Summary . . . . .	44
<b>4</b>	<b>Quantitative Analysis</b>	<b>46</b>
4.1	Introduction . . . . .	46

4.2	Timing Analysis . . . . .	47
4.2.1	WordCount . . . . .	47
4.2.2	BLAT . . . . .	50
4.3	Programming Comparison . . . . .	52
4.4	Failure analysis . . . . .	56
4.5	Summary . . . . .	56
<b>5</b>	<b>Active storage in Message Passing and MapReduce</b>	<b>58</b>
5.1	Introduction . . . . .	58
5.2	Active Storage with Object Based Devices . . . . .	58
5.2.1	Object-based Storage Devices . . . . .	59
5.2.2	OSD commands . . . . .	61
5.3	Basic Approach . . . . .	62
5.3.1	Design Methodology . . . . .	62
5.3.2	Sample APIs Developed . . . . .	64
5.3.3	Implementation . . . . .	65
5.3.4	OSD Design Changes . . . . .	66
5.3.5	Results . . . . .	67
5.4	Active storage in message passing and MapReduce . . . . .	68
5.4.1	Architecture Overview . . . . .	69
5.4.2	Application . . . . .	71
5.5	Summary . . . . .	73
<b>6</b>	<b>Conclusion</b>	<b>74</b>
	<b>Bibliography</b>	<b>76</b>

## List of Tables

### Table

2.1	Basic datatypes of mpiJava . . . . .	21
3.1	Comparison of mpiJava and MapReduce characteristics . . . . .	45
4.1	WordCount - Execution time for MapReduce and mpiJava for different input file sizes . . . . .	48
4.2	BLAT - Execution time for MapReduce and mpiJava for different input file sizes . . . . .	51
4.3	Programming comparison of MapReduce and mpiJava . . . . .	55
4.4	BLAT - Execution time for MapReduce with and without a node failure for different input file sizes on 8 compute nodes . . . . .	57
5.1	Object identification numbers . . . . .	61
5.2	Results for 1M Records . . . . .	68

# List of Figures

## Figure

1.1	Parallel Computing Architecture. <i>Adapted from [BFG<sup>+</sup>95]</i> . . . . .	5
2.1	MPI Architecture. <i>Adapted from [MSW<sup>+</sup>05]</i> . . . . .	18
2.2	Principal Classes of mpiJava. <i>Adapted from [BCKL98]</i> . . . . .	20
2.3	MapReduce Execution Overview . . . . .	23
2.4	GFS Architecture. <i>Adapted from [hdf]</i> . . . . .	29
4.1	WordCount - Execution time vs Number of compute nodes - 90MB and 5.4GB input file . . . . .	49
4.2	BLAT - Execution time vs Volume of data on 8 compute nodes . . . . .	53
4.3	BLAT - Execution time vs number of compute nodes - 1.9GB and 938MB input file . . . . .	54
5.1	Comparison of traditional and OSD storage models [DHH <sup>+</sup> 06] . . . . .	60
5.2	Basic structure of Active Disk computation via OSD . . . . .	63
5.3	Active storage programming model for message passing . . . . .	70
5.4	Active storage programming model for MapReduce . . . . .	70



## Abstract

*Data intensive applications attempt to extract valuable information from large amounts of data, but the ability to scalably and efficiently do so remains a intimidating challenge. Efficient distributed parallel programming techniques are the key to achieving the performance requirements for such computations. The most common cluster data parallel programming methodology is to use message passing to communicate between computation nodes. Recently, the MapReduce model has been introduced as a more efficient mechanism to enable easy development of parallel applications to process large amounts of data on large clusters. In this paper we analyze and compare the performance of MapReduce and message passing in general and specifically for two data intensive computations: (a) Word-Count and (b) Blast Like Alignment Tool (BLAT). This thesis also contributes to analyzing active storage on an iSCSI OSD standards-based object oriented framework. The concept of performing portions of computation directly at disk drives can drastically reduce network contention and greatly improve efficiency of large parallel systems.*

*To my family...*

# Chapter 1

## Introduction

### 1.1 Motivation

Computer science today is rapidly moving towards parallelism as a means to prevail against increasing problem sizes and the declining rate of clock speed improvements. Parallel programming was developed as a means of improving performance and efficiency. Parallel programming is a strategy to execute large and complex tasks faster by breaking the computation into parts, each of which can be executed concurrently by being assigned to multiple workers and coordinating the workers. The instructions in each part have the ability to run simultaneously on different CPUs that can exist on a single machine, or they can be on a set of computers connected through network.

The exponential growth of data has led to challenges in the computing field. In order to meet the increased computing challenges we must be able to sample, aggregate and perform timely analysis of the data. The advancement in data intensive computing has favored to collect large amounts of data, taking us close to solving some complex problems such as genomic searches and computation, data mining applications such as marketing, surveillance, and fraud detection, and global climate change.

In order to process large amounts of data, the computations have to be distributed across

multiple machines in order to finish in a reasonable amount of time. This could be achieved by employing smaller cost efficient machines to carry out a part of the split cumbersome computation. The main issues in carrying out a distributed computation are:

- Parallelize the computation,
- Distribute the data, and
- Handle failures.

These important issues in distributed computing collude to complicate the original simple computation with large amounts of complex code to deal with these issues. The exploding data and ever-increasing need to integrate data leads to build large data-intensive applications. Building large data-intensive applications to handle exponentially growing data, to process, analyze and extract valuable information in real time is a daunting challenge. The most important aspect of a large data-intensive application development includes process flow and communication, flexibility, synchronization, extensibility, scalability, stability, availability and manageability.

Not only are parallel programs faster, they can also be used to solve problems on large datasets using both local and non-local resources. When we have a set of computers connected on a network, we have a vast pool of CPUs, and we often have the ability to read and write very large files with a distributed file system. Parallel computing requires

- Multiple processors (workers),
- Network link between multiple processors,
- An environment to create and manage parallel processing which include the administrator (to handle multiple workers)

- A parallel programming paradigm, and finally
- An algorithm to split the problem into pieces for multiple workers to execute.

As a reaction of complexity, choosing between different distributed parallel programming techniques to accommodate exponentially growing data with a finite amount of time to do the required analysis becomes difficult. The major contribution of this work is to analyze the two major distributed cluster parallel programming techniques - namely MapReduce and message passing for data intensive applications.

In this thesis, we first explore the two chosen parallel programming techniques. Details of the architecture and working have been dealt with in these sections. Next, we discuss in depth about the two data intensive applications namely BLAT (BLAST Like Alignment Tool) and WordCount which would employ the two parallel programming techniques to benefit in computation. BLAT is a bioinformatics tool employed for genomic search based on a query and WordCount which analyses a text file and gives the count of words, number of lines and frequency of word occurrence in the input file. We would then analyze the results for the data intensive application and compare the techniques from the obtained result.

## **1.2 Parallel Computing**

In the early days of computing, all the programs were serially executed. A program had a sequence of instructions, and each instruction was executed one after the other in order. All of these instructions were run on a single processor. Parallel computing was developed as a means of improving performance, speed and efficiency.

The evolution of parallel computer architectures has recently created new trends and challenges for both parallel application developers and end users. Parallel computing is a form of computation in which many calculations are performed simultaneously, operating

on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel). Parallel Computing are categorized by the level of parallelism employed with multi-core and multi-processor comprised of many processing units within a computer, while clusters, massively parallel processors (MPP), and grids use multiple computers to work on the same task.

Parallel computer programs are more difficult to write than sequential ones, because the simultaneous executions introduces several new potential hazards like data dependencies which can lead to the race conditions and other software bugs. Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel. Communication and synchronization between the different subtasks is typically one of the greatest barriers to getting good parallel program performance. The speed-up of a program as a result of parallelization is given by Amdahl's law. In the next section we would study the various architecture approaches of parallel computing. We would then categorize the two chosen parallel programming techniques, namely the message passing and MapReduce. We would learn about the advantages and disadvantage of each of the architectures.

### **1.2.1 Architecture Approaches**

Parallel computing systems can be built on top of different hardware architectures, the main three as shown in Figure 1.1.

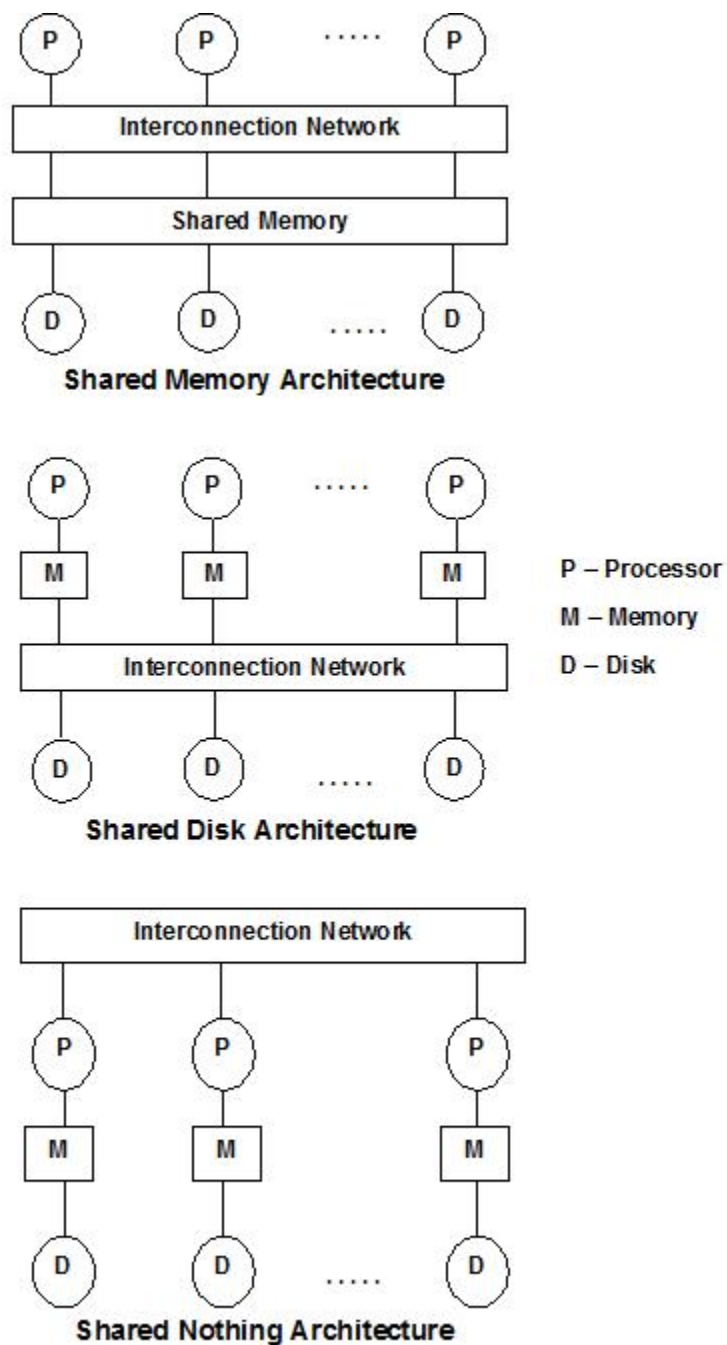


Figure 1.1: Parallel Computing Architecture. Adapted from [BFG<sup>+</sup>95]

### **1.2.1.1 Shared Memory**

In the shared-memory systems, multiple processors share a common central memory. With this architecture, communication among the multiple processors is through the shared memory, thus there is very little message overhead. In addition, the platform required to establish parallel computing system is considerably less complex. Consequently, many of the commercial parallel computing systems available today are based on the shared-memory architecture.

The message passing interface(MPI) falls under this category. The principal MPI-1 had no shared-memory concept and the next version namely, the MPI-2 has a limited shared-memory concept. Nonetheless, MPI, one of the parallel programming techniques we analyze are regularly run on shared-memory computers since MPI encourages memory locality.

### **1.2.1.2 Shared Disk**

In shared-disk systems, the multiple processors, each with its private local memory, share a pool of disks. Shared-disk systems usually doesn't scale well for smaller machines. Shared-disk systems overcome the central memory access bottleneck as in shared-memory system, but introduce the difficult problem of controlling all processors access to all disks. This can be especially difficult when the system comprises of a large number of processors and disks. The system should have some means to make sure that all nodes in the system, have a consistent view of data as it changes.

In addition, shared disks present the most challenging task of transaction management, as there arises a need to co-ordinate global locking activities (without the help of shared-memory) and to synchronize log writing among all processors. Shared-disk usually is available for applications and services requiring only modest shared access to data as well



as applications or workloads that are very difficult to partition. The shared-disk has the advantages of quick adaptability to changing workloads, high availability and reduced data partitioning.

#### **1.2.1.3 Shared Nothing**

With the shared-nothing architecture, each processor has its own memory as well as one or more local disks. And, each node in the cluster has its own independent subset of data it can work on independently without encountering resource contention from other nodes. Except for the communication network, no other resources are shared among these processors. Shared-nothing does not have the memory access bottleneck problem, nor does it have the problem of interconnecting a large number of processors and disks. So shared-nothing architecture offers excellent scalability and can be built on simpler and cheaper hardware with ideal data partitioning. The major complexity in supporting the shared-nothing architecture is the requirement of breaking a problem into multiple subproblems sent to different nodes in the system and merging the results generated by the multiple nodes.

The second parallel programming technique, MapReduce we analyze falls under the shared-nothing category. MapReduce is a wonderful example of specialized data store that runs on a massive shared-nothing architecture. At the same time it keeps the computation near the data. The shared-nothing architecture is the most cost-effective and promising architecture for high-performance parallel programming technique.

### **1.2.2 Parallel Programming Techniques**

The first step in developing an Parallel program [Ble96] is to identify a task that can be partitioned and run concurrently on multiple machines. The task should be able to

handle and manage the partitioned input data. The main things that have to be considered when choosing a parallel programming technique for an application are the type of parallel architecture being used and the type of processor communications used. Based all of these considerations we have mainly chosen to analyze two of the Parallel computing systems namely, message passing and MapReduce.

### **1.2.2.1 Message Passing Interface**

Message Passing Interface is a specification for message passing. By itself, it is not a library but more of a specification of what a library should be. Message passing is an approach that makes the exchange of data cooperative among a set of interconnected computers. MPI provides essential synchronization, and communication between the interconnected computers. MPI remains one of the dominant models used in high-performance parallel computing today. The main goals of message passing are practical, portable, flexible and efficient communication.

MPI [Bar] transforms itself to virtually any distributed memory parallel programming model. MPI is commonly used to implement some shared memory models, such as Data Parallel, on distributed memory architecture. Originally, MPI was designed for distributed memory systems like MPP (Massively Parallel Processor). As the shared-memory systems with the SMP/NUMA (Symmetric Multiprocessor/Non-Uniform Memory Access) architectures became popular, MPI implementations for these platforms emerged. Now-a-days MPI is used on just about any parallel architecture not limited to massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks.

One of the basic drawbacks of using the message passing technique is that all parallelism is explicit. The programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using the available MPI constructs. In MPI-1 the num-

ber of tasks dedicated to run a parallel program is static. This prevents new tasks being dynamically spawned during run time. But this feature has been enhanced in MPI-2 which remove the static process model of MPI-1. It provides routines to create new processes. The details of the architecture and the working are dealt in the following chapters.

### 1.2.2.2 MapReduce

MapReduce [DG08, had] is a programming framework which was inspired by the concept of Map and Reduce from Lisp a functional language. In Lisp, a map task takes a function and sequence of values as input. The map function then executes the function by taking each value in order. The reduce function then combines all the elements of the sequence using a binary operator. MapReduce was inspired by these concepts. The MapReduce paradigm is summarized in the following quote [DG04]:

*“The MapReduce Computation takes a set of key/value input pairs, and produces a set of key/value output pairs. The user defines two functions in the MapReduce library for computation: map and reduce. Map, a user-defined program takes a set of input values and produces an intermediate key/value pair. The reduce function another user program accepts the intermediate key and set of values for that key. It merges them to a smaller set of values. Typically the number of output files are same as the number of reduce workers employed and each reduce invocation produces one such file. The intermediate values are supplied to the reduce function via an iterator. So it makes possible to handle large lists of values which do not fit in memory.”*

The Map and Reduce functions are typically of the format:

$$\begin{aligned} \text{map } (k1, v1) &\rightarrow \text{list}(k2, v2) \\ \text{reduce } (k2, \text{list } (v2)) &\rightarrow \text{list}(v2) \end{aligned}$$

It was developed by Google as a mechanism for processing large amounts of raw data. These data was from the crawled web pages or documents and web request logs. The data was so large that it could not be done on any single machine. It must be done by distributing the data across thousands of machines to be done in a reasonable amount of time. This distributed computation implies parallel computing, as the same computation was done on each CPU but with a different dataset. MapReduce is an abstraction that enables the Google engineers to perform these as simple computations, while hiding the details of parallelization, data partitioning, data distribution, load balancing and fault tolerance. Some of the notable advantages using MapReduce framework are automatic parallelism and distribution, fault-tolerance, I/O scheduling, and status monitoring. The most important advantage when handling large datasets being the automatic parallelization and distribution of processing.

### 1.3 Data Intensive Application

Data Intensive Computing [dic, Bry, EPF08] is acquiring, analyzing, and processing volumes of data. The ever advancing technology provides the ability to handle these large data sets with economical storage, and faster bandwidth. These data sets evolve from a variety of applications ranging from scientific research(e.g., bioinformatics, climate change), transactional data (e.g., payroll, accounting) and environment(e.g., ozone layer, water acoustics). The challenge of these large amount of data is to extract valuable information in real-time. This problem could be split up as

- (1) accommodate ever-increasing data
- (2) process and obtain valuable information
- (3) convert information into human readable data

Data intensive computing requires a well structured set of principles. Many data-intensive applications can be parallelized to achieve the required output specification. The data-intensive applications demand reliability, high fault tolerance, real-time response and availability.

In this thesis, we mainly concentrate on two data intensive applications - the simple Word Count application and the much more complicated BLAT application. Word Count is derived from a MapReduce application and has been ported to MPI and BLAT was an initial MPI application that has been ported to MapReduce. We would then employ the parallel programming techniques, message passing and MapReduce tailored for these two data intensive applications. The results from these experiments would be presented and analyzed in the chapter 4. The following section briefs the data intensive applications.

### **1.3.1 Word Count**

WordCount is an application that counts how often a word occurs, the total number of words, lines and characters in a text file. Knowing the numbers of words is sometime important, for instance when we are required to stay certain minimum or maximum bounds. These applications may include a submission of conference or journal paper, a thesis report or a story to be published. Currently there is a rapidly growing trend among businesses to analyze their increasing volumes of URL access log files and analyze web server logs to find popular URLs. This could enable them to acquire valuable information regarding

the sources that have been using their website for various purposes. In the case of both the message passing and MapReduce we look at the input is text files and the output as text files, and the in the output, each line of which contains a word and the count of how often it occurred, separated by a tab.

### **1.3.2 BLAT**

The second data intensive application to be considered is BLAT (BLAST Like Alignment Tool) [Ken, Ken02]. BLAT is one of the most widely used bioinformatics tool. This is commonly used to look up the location of a sequence in the genome or determine the exon structure of an mRNA and also performs a rapid mRNA/DNA and cross-species protein alignments. BLAT is an alignment tool like BLAST, but it has a different structure. When the BLAT was used in real world, due to sequence divergence rates over evolutionary time – the BLAT designed for DNA works well within humans and primates, while BLAT designed for the protein sequences continues to find good matches within terrestrial vertebrates and even earlier organisms for conserved proteins. Among the human genomic sequences, protein BLAT gives a much better picture of gene families than DNA BLAT.

BLAT is very accurate and performs mRNA/DNA alignments 500 times faster than existing alignment tools. It also does protein alignments 50 times faster. On DNA, BLAT works by keeping only an index of an entire genome in memory rather than the genome itself. These indexes takes less than a gigabyte of memory, thus enabling BLAT to provide higher performance on a reasonably prices Linux machine. Thus, the target database of BLAT on which the query is executed is not a set of GenBank sequences, but instead an index derived from the assembly of the entire genome. So the obtained index – which uses less than a gigabyte of RAM – consists of all non-overlapping 11-mers except for those heavily involved in repeats. Thus the smaller size of index means that BLAT is far more

easily mirrored. BLAT for the execution on a DNA sequence is designed to quickly find sequences of 95% and greater similarity of length 40 bases or more. This BLAT of DNA may miss more divergent or short sequence alignments. BLAT designed for execution on protein sequences finds sequences of 80% and greater similarity of length 20 amino acids or more.

BLAT can be used by experts to run large batch jobs and make internal parameter sensitivity changes by installing command line BLAT. The indexing of the genomic sequence provides a good advantage over the other genome matching tools. The index of the genome is used to find areas of probable homology, which can then loaded into memory for a detailed alignment. Protein BLAT is also designed to work in a similar manner, except with 4-mers rather than 11-mers. The protein index of a genomic sequence takes a little more than 2 gigabytes.

On a practical approach, BLAT has several advantages over Blast:

- Faster execution and reduced time
- Ability to handle simultaneous queries in FASTA format
- Resulting output has five convenient options and very flexible
- Block alignment is detailed in natural genomic order.

The input file format for the BLAT application is the FASTA format. In bioinformatics, the FASTA format is a text-based format for representing either nucleic acid sequences or peptide sequences, in which base pairs or amino acids are represented using single-letter codes. The format also allows for sequence names and comments to precede the sequences. The simplicity of FASTA format makes it easy to manipulate and parse sequences using text-processing tools.

The genomic sequence is the database file on which a matching alignment called the query is performed. The BLAT application deals with these database file and the query file. The BLAT first reads the database file and creates an index of the genome. The index is in the memory. The query file contains the sequence that has to be matched or aligned to the genome. Then the BLAT performs the matching of the query to the created genomic index. Both the database and query input files are FASTA format. The resulting output of BLAT is flexible. By default the output is a simple tab-delimited file which describes the alignment. The output only specifies the alignment position but does not include the sequence of the alignment itself. Building an index of the genome typically using the BLAT takes 10 to 15 minutes.

When BLAT is run on a genome set, it is analogous to a database being queried. The working of the BLAT application is similar to the grep application which does the query matching on the available database. The resulting output would be the sequences names, matched sequence position, and matching length from the database FASTA file.

### **1.3.3 Other Examples**

As parallel computing become larger and faster, it becomes feasible to solve problems that previously took too long to run. Parallel computing is used in a wide range of fields, from bioinformatics (to do protein folding) to economics (to do simulation in mathematical finance). Some common examples that can be considered applicable to the distributed parallel programming techniques are Count of URL Access Frequency, Matrix Multiplication, Inverted Index, Distributed Sort, and Graph traversal.



## 1.4 Outline

This thesis has been organized into 4 chapters. The first chapter gives the introduction and lays the background for this work. It also briefly discusses the significance of MapReduce and mpiJava. Section 1.2 elaborates the two parallel computing techniques with the architecture overview and details the data intensive applications. Chapter 2 details the architecture overview of the two parallel programming techniques namely, message passing and MapReduce. Chapter 2 also provides the specifications for tailoring the parallel programming techniques for the chosen data intensive computation. Chapter 3 describes the distributed application programming analysis and partitioning techniques for the large data sets. Chapter 4 elaborates the performance measurements of MapReduce and Message Passing for the chosen applications along with the timing analysis between the two programming techniques. In the following Chapter 5 we will discuss how both MPI and MapReduce can be mapped to an active storage system. Finally, in Chapter 6, we conclude with our ideas for future work.

## **Chapter 2**

# **Distributed Computation Programming Analysis**

### **2.1 Introduction**

The Data-Intensive Computing program aids in increasing our understanding of the capabilities and limitations of data-intensive computing. The criteria to choose among the available data-intensive computing technique are the amount of data involved, the computing capabilities and the underlying storage. The distributed parallel programming techniques which we employ should have high-level parallelism, easier programming, high reliability, higher performance, and economical.

In the following section, we describes in detail the two parallel programming techniques we have selected to analyze for the chosen data-intensive applications. We would go in depth about the architecture and working principles.

### **2.2 Message Passing Interface - An Overview**

Message passing is the most commonly used paradigm for cluster-based parallel programming. The Message Passing Interface (MPI) [mpi95, mpi97] is the standard used to code parallel computing applications based on this paradigm. Derived datatypes, dynamic process management, point-to-point and collective communication are supported by

MPI. This is provided through a shared, group-wide communicator, instead of socket pairs. The MPI interface was developed with the intent of comprehending all of the available message-passing constructs and features of various MPP and networked clusters so that programs would execute on any type of system. MPI's portability feature supports copying, compiling and execution of a program written for one architecture on another without modification. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

Single Program Multiple data (SPMD) is one of the ways to achieve parallel computing where a single program is split across a group of processes to execute concurrently with localized data. MPI is designed to support SPMD model. So when developing an application to be executed on a single MPP, then MPI has the advantage of higher communication performance and the ever desired ease of portability. MPI has a much richer set of communication functions which makes it advantageous to use on an application with special communication mode requirements.

The architecture of the MPI [MSW<sup>+</sup>05] is shown in figure 2.1. This shows two components of the MPI architecture: message passing and process management. The upper layers of message-passing are implemented using MPI code. MPI provides the implementation of point-to-point messages, in-built and user-defined data types, communicators and collective operations. Its interface with lower layers are through the Abstract Device Interface (ADI). The ADI layer consists of a set of data structures and functions specified by the implementation. The ADI layer mainly deals with the MPI requests (messages) and functions to send, receive, and manipulate these requests.

MPI's [GFB<sup>+</sup>04], [FK98] implementation language in general is different from the language it supports at runtime. Most MPI implementations are a combination of C, C++ and assembly language, and target C, C++, and Fortran programmers. MPI-CH and MPI-Ram

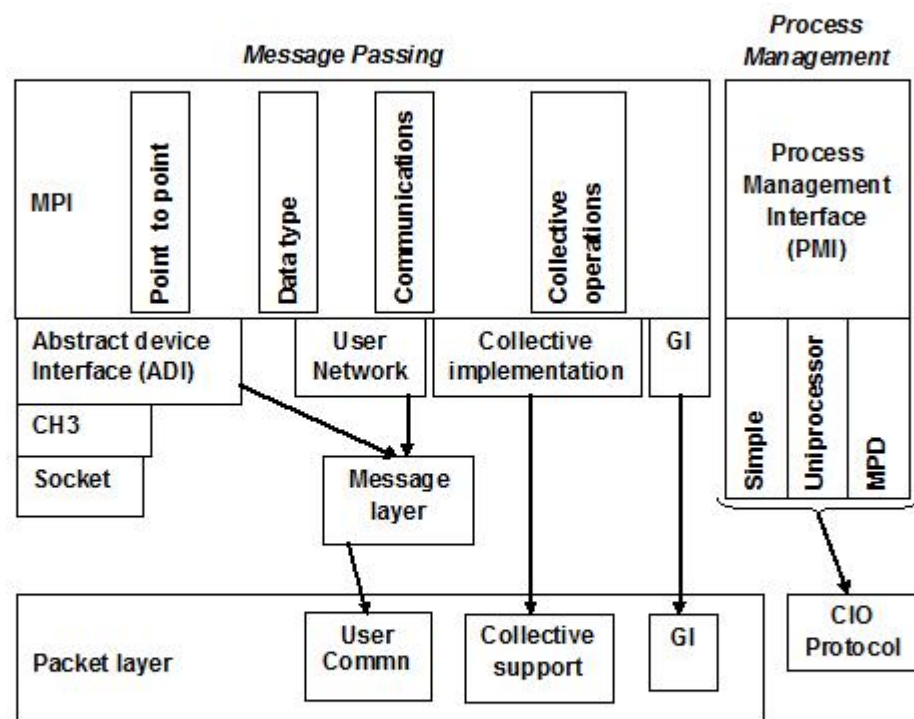


Figure 2.1: MPI Architecture. *Adapted from [MSW<sup>+</sup>05]*

are libraries that have implemented MPI functions in C and C++. However, Java programs cannot use these libraries directly, for this reason mpiJava was designed to bridge Java applications to the underlying MPI-CH or MPI-Ram functions. To achieve the promised high communication performance, some of the notable forfeits were the lack of interoperability between two MPI implementations, and lack of ability to write a fault tolerant application. One of the guarantee MPI specification provides is ability to quit the program on error. The mpiJava [BCKL98], [mpia] interface is designed to provide essential virtual topology, synchronization, and communication functionality between a set of computers in a language-independent and language-specific way. To achieve the desired maximum performance, each CPU is assigned only a single process during runtime.

The main reason for mpiJava to emerge was that, in 1990's there has been a great deal of interest in using Java for scientific and engineering computation, and in particular for parallel computing [jav96], [Fox97], [Fox98]. Reason behind this was that Java, was a simple, efficient and platform-neutral language. Java has some in-built packages for communication, notably an easy-to-use interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. These features interest the parallel programmers. mpiJava was a implementation of the Java binding through a set of wrappers that use the Java native methods interface (JNI) to call existing MPI implementations.

The existing MPI standard is explicitly object-based implementation. The C and Fortran MPI bindings rely on "opaque objects" which can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. But in the enhanced MPI-2 standard, the C++ binding collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The mpiJava [CFKL], [CGJS00] API follows this model, importing the structure of its class hierarchy directly from the C++ binding. The major classes of mpiJava are

illustrated in Figure 2.2.

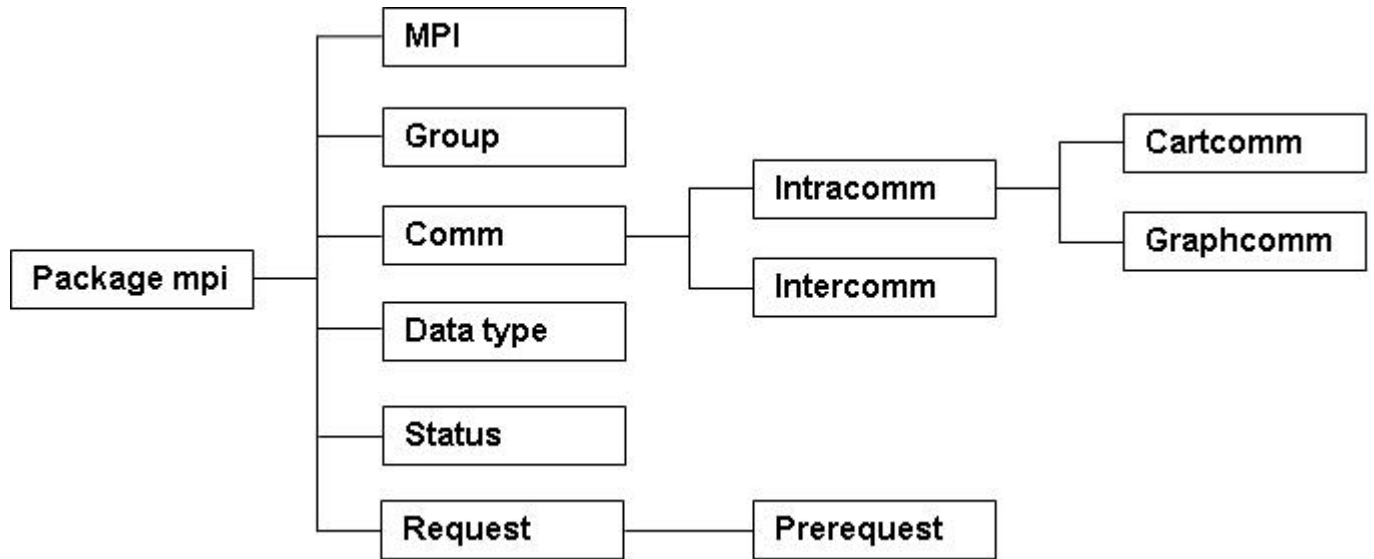


Figure 2.2: Principal Classes of mpiJava. *Adapted from [BCKL98]*

The basic class “MPI” has only static members. This module contains the global services, such as initialization of MPI, and many global constants, including the default communicator COMM\_WORLD. The most important class in the mpiJava package is the communicator class “Comm”. The basic communication functions in mpiJava fall under the members of Comm or its subclasses. As always in MPI, a communicator stands for a “collective object” logically shared by a group of processors. The processes manages, typically by addressing messages to their peers through the common communicator. The standard send and receive operation of MPI fall under the “Comm” class with interfaces as shown below:

```

public void Send(Object buf, int offset, int count, Datatype datatype, int dest, int tag)
public Status Recv( Object buf, int offset, int count, Datatype datatype, int source, int tag)

```

In both send and receive, the actual argument corresponding to “buf” must be a Java

array. In all mpiJava calls [CGJ<sup>+</sup>98], [CFZL97], the buffer array argument is followed by an “offset” that specifies the element in the array where the message actually starts. Another class which deals with the the type of the elements in the message buffers passed to send, receive, and all other communication functions, is the “Datatype” class. Some of the basic datatypes are predefined in the package. These mainly correspond to the primitive types of Java, shown in Table 2.2.

<b>MPI Datatype</b>	<b>Java Datatype</b>
MPI.BYTE	byte
MPI.CHAR	char
MPI.SHORT	short
MPI.BOOLEAN	boolean
MPI.INT	int
MPI.LONG	long
MPI.FLOAT	float
MPI.DOUBLE	double
MPI.PACKED	

Table 2.1: Basic datatypes of mpiJava

The mpiJava provides all the derived datatype constructors of standard MPI, with only one limitation: it places significant restrictions on its binding of “MPI\_TYPE\_STRUCT”. Similarly, MPI destructors can be absorbed into Java object destructors, which are called automatically by the Java garbage collector. Also, the explicit calls to the MPI destructor functions are typically omitted from the Java user interface (they are absorbed into finalize methods). There are some exceptions made for the “Comm” and “Request” classes. “MPI\_COMM\_FREE” is a collective operation, and the user must ensure that calls are made at consistent times by all processors involved—the call can’t be left to the garbage collector. Similar is the case for “MPI\_REQUEST\_FREE”.

However, even mpiJava has not been able to handle and recover faults automatically.

mpiJava has static nature, wherein, the processes are created during the initialization and assigned process numbers, every process keeps details of every other processes, new processes cannot be generated during runtime. This static nature of mpiJava favors in achieving the higher performance. The main features include the checkpointing done at frequent intervals and restarting a compute node on failure. This can be quite inefficient when involving a larger system. When the mpiJava tasks reside on the same data node, the communication can be over the switch network or through shared memory concept. But when there are large amounts of data that has to be communicated over the network it results in a significant amount of I/O traffic sometimes leading to Network Contention.

### **2.3 MapReduce - An Architecture Overview**

MapReduce [DG08, had] is a programming paradigm and framework developed by Google for simplified, parallel data processing on clusters built of commodity hardware. As its name implies, it was inspired by the Lisp (list-processing) functions map and reduce, also known as accumulate. However, the MapReduce model behaves slightly differently as all of the input, output, and intermediate data are expressed as key/value pairs. The MapReduce [DG04] infrastructure is designed to handle the partitioning of data and distribution of input key/value pair data across the cluster. The Model also employs a namenode and jobtracker to distribute and execute the task across all of the available nodes on the cluster. Each of the mapper function is assigned a input split it operates on and produces the intermediate key/value pairs. The combiner does the process of shuffling and sorting all the intermediate results. Finally these sorted keys are assigned to the reducer, which produces the output key/value pairs. In some cases, the merge-sort [Knu79] [vN63] function may be performed multiple times to get a single output file.

The advantage of MapReduce is the ability to distribute and process map and reduce



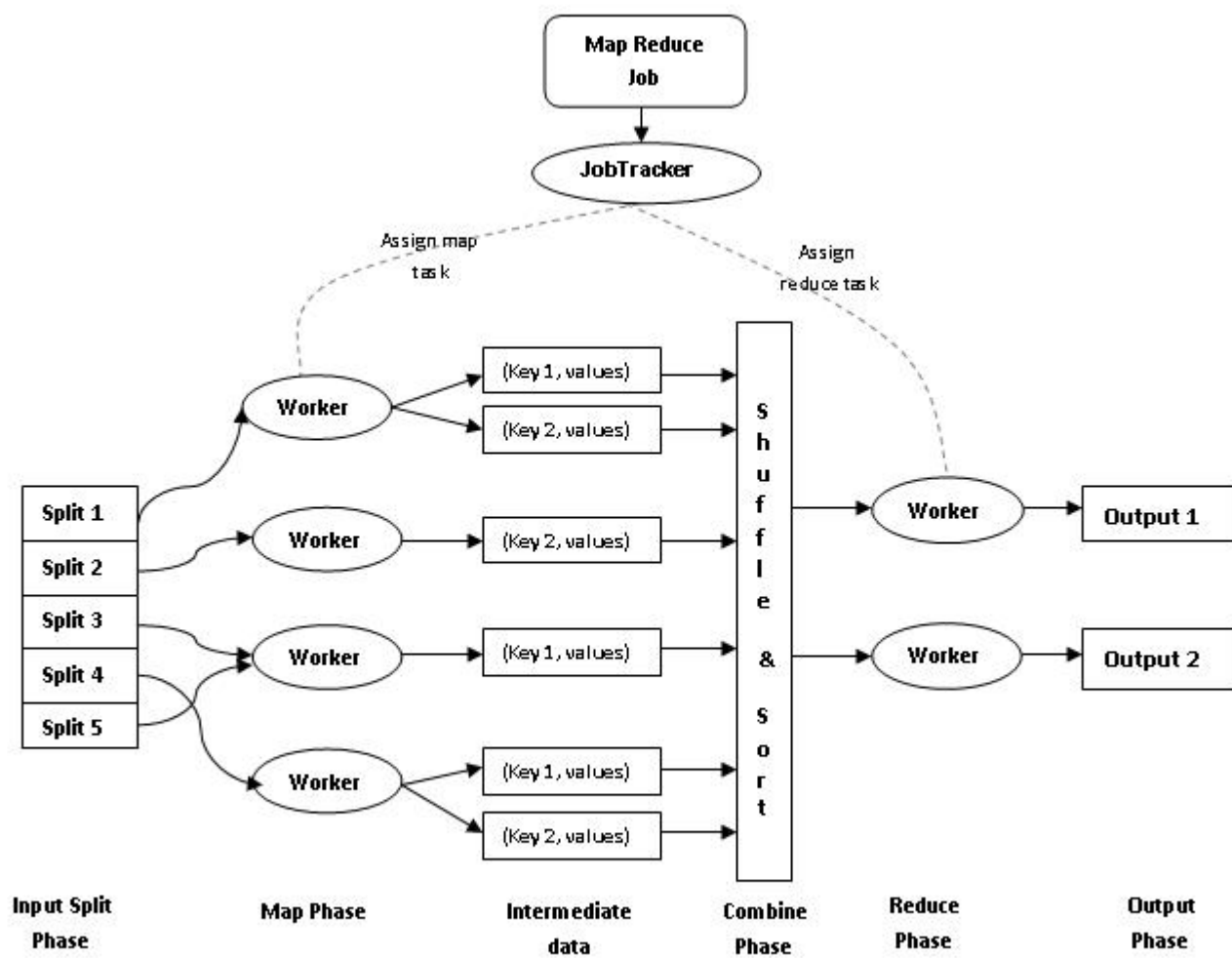


Figure 2.3: MapReduce Execution Overview

operations. The only concern is that the MapReduce job should have the capability of splitting the job and be executing them in parallel. There might be some limitation to the map job arising due to data availability and location of CPU's near the data. Similarly, the reduce job works on the results from the mapper stored in intermediate local disks. For reduce, we need to ensure that records with the same intermediate key end up at the same worker. This technique works well with large data sets. In addition to parallelism it also provides a fairly good failure recovery. Due to the replication of available data, even if a worker node fails, the job could be rescheduled to another node where the replicated data resides. In MapReduce [ZKJ<sup>+</sup>08, RRP<sup>+</sup>07] model, each of the node has local hard drives where the intermediate files are stored. These files can be remotely read by the reducers. Thus MapReduce is a shared-nothing architecture. But all the MapReduce jobs read from and write data to HDFS [hdf] which is shared by all the nodes. HDFS takes care of the replication placing them on compute nodes around the cluster based on the specified replication factor. Due to this shared-nothing architecture scalability can be easily achieved, and the integrated HDFS makes thousands of disks appear like one.

### **Execution Overview**

MapReduce execution begins when a map method has been initiated. The Map invocations are distributed across multiple nodes by automatically partitioning the input data set into a set of M splits. Each of the input split can be processed in parallel on multiple machines. The Reduce invocations are distributed by partitioning the intermediate key/-value pairs using a partitioning function. The number of partitions(R) and the partitioning function for the reduce phase are specified by the user. The MapReduce job for a specific application has all of the functions to deal with input data such as input reading, record reader and input splitting. It also has the modified Map, Combine and Reduce phase tailored to match the application. Finally it provides the specifications for the output writer.

Figure 2.3 illustrates the overall MapReduce Execution Overview. When an user application calls the MapReduce function, the following are the sequence of execution:

- The MapReduce library in the user job for the application first splits the input files into “M” pieces typically of the size 16 MB(megabytes) to 64 MB per input split.
- The input data split are the replicated on the available nodes depending on the replication factor specified by the user.
- It then starts up many copies of the application program on the cluster of machines.
- One such copy is the most important called the JobTracker or the Master. All of the other copies are the workers which are assigned work by the Master. There are “M” map tasks and “R” reduce tasks to be assigned to the workers. The master assigns each of idle workers a map task or a reduce task.
- Each of the worker assigned to a map task has an assigned input split. It reads the content of the input split, converts the input data into key/value pairs. These key/value pairs are then passed to the user-defined Map function. The output of the Map function are intermediate key/value pairs “I”. These intermediate pairs are buffered in memory and passed on to the next phase in the MapReduce execution.
- Periodically, the buffered intermediate key/value pairs are shuffled and sorted so that all occurrences of the same key are grouped together. This is done by the combiner function. If the amount of intermediate key/value pairs is too large to fit in memory, an external sort is used. These shuffled and sorted key/value pairs are written to the local disks. The locations of the shuffled and sorted key/value pairs on the local disk are sent to the JobTracker.

- The shuffled and sorted intermediate key/value pairs from the local disk are partitioned into “R” regions by the partitioning function. The locations of these partitioned intermediate key/value pairs are passed by the JobTracker to the reduce workers.
- When the reduce workers are notified by the JobTracker about the intermediate key/value pair locations, it uses remote procedure calls to read the stored data from the local disks of the map workers.
- When the data assigned to the reduce worker are completely read, it iterates over the sorted intermediate data. And for each unique intermediate key, it passes the key and the corresponding intermediate values to the user-defined Reduce function. The output of the Reduce function are appended to an output file for the corresponding reduce partition.
- When all the map tasks and reduce tasks have been completed, the master returns the the control to the main program from where the user program was called.

After the MapReduce job has been successfully completed, the output of the MapReduce function is available in “R” output files.

The master periodically pings every worker to detect any node failure. When there is no response from a worker for a certain amount of time, the master considers the worker to have failed. If the failed worker has completed a map task, the task is reset back to the initial state, and the map task can now be assigned to any other idle worker. Similarly, any map task or reduce task in progress has to be reset to initial state and the task becomes eligible for rescheduling. Completed map tasks have to be re-scheduled and re-executed on a node failure because the output of the map task will be stored in the local disk. So when a node fails, the output becomes inaccessible. But this is not the case for the Reduce

task completed on a failed node, since the output of the reduce task are always stored on a global file system.

### **Data Flow**

The major blocks involved in the working of MapReduce task is detailed below:

#### **2.3.1 Input Reader**

The Map/Reduce framework relies on the InputFormat of the job to validate them, split the input file(s) into logical InputSplit instances of size 16MB to 64MB, each of which is then assigned to an individual Mapper, and provide RecordReader to collect the input record from logical split and present a record-oriented view to be processed by the Mapper. Typically the input reader reads data from HDFS and generates key/value pairs.

#### **2.3.2 Mapper**

Map function, specified by the user program, takes an input split and produces a set of intermediate key/value pairs. The intermediate key/value pairs are stored on the local disks which are remotely read by the Reducer function. The input and output types of the Map function are often different.

#### **2.3.3 Combiner**

The Combiner function, specified by the user program, remotely reads the intermediate key/value pairs processed by different Map jobs located on the local disks. It then shuffles and sorts the key/value pairs based on the intermediate key.

### 2.3.4 Partitioner

The Partition function takes the sorted output of all of the maps from the combiner and allocates to particular reducer. The partition function takes the key and the number of reduces as arguments and returns the index of the desired reduce. MapReduce computing system uses a default partition function  $hash(key) \bmod R$ . Typically it is done by hashing the key and modulo the number of reduces.

### 2.3.5 Reducer

The Reduce function, also specified by the user program, accepts an intermediate key and a set of values for that key from the combiner function. It then merges the set of values in to a smaller set of values. A reducer always receives key/value pairs as input and produces key/value pairs as output. For reduce worker, we need to ensure that records with the same intermediate key end up at the same worker. For large datasets, the reducer may perform the reduce action in multiple steps.

### 2.3.6 Output Writer

The Output Writer writes the output of the reduce to stable storage, usually the Hadoop distributed file system.

### 2.3.7 GFS

The Google File System (GFS) [hdf, GGL03] is a distributed file system designed to run on commodity hardware. Figure 2.4 depicts the architecture. There are many similarities with the existing distributes file system whereas the notable differences include high fault tolerance and ability to be built from commodity hardware. GFS is designed to support large datasets, provide high data bandwidth and scale to thousands to nodes.

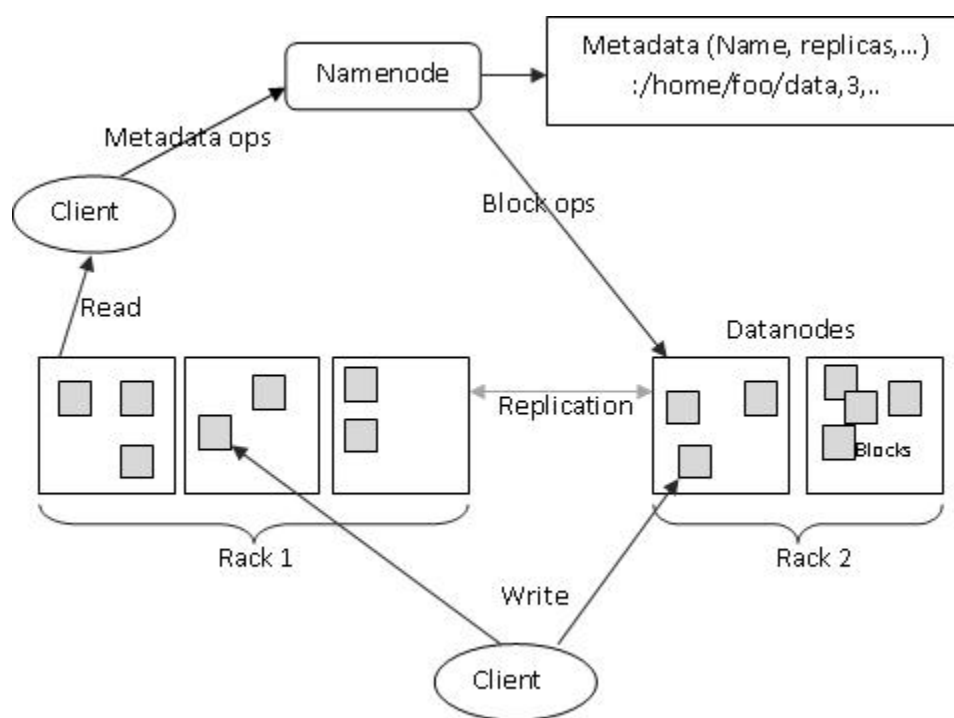


Figure 2.4: GFS Architecture. Adapted from [hdf]

GFS applications are designed to fall under the model of write-once-read-many. So a files that are written once and are accessed multiple times result in high throughput data access. So the parallel computing on large data sets could be one among them. Also when the computation is carried out near data the network congestion could be reduced resulting in increased throughput of the system. GFS provides interface to move application close to data. GFS is designed to be portable on different platforms. GFS has the master/slave architecture. The GFS has a single master, NameNode which keeps track of all the slaves, and manages the namespace. There are many slaves, the DataNode which carry out the process instructed by the master node. The input file(s) is usually split across the available datanodes. The mapping of the blocks on the DataNodes is taken care of by the NameNode. NameNode creates, deletes and renames the file blocks on the DataNode. DataNode also performs these operations when instructed by the NameNode.

## **2.4 Parallel Programming for Data Intensive Computation**

In this section we would discuss in detail the specifications for the applications to be run on the parallel programming techniques.

### **2.4.1 Word Count Specification**

We make use of the open source MapReduce WordCount [DG04, wor] example for our implementation and analysis. This application speedily does the counting of word frequencies in a collection of documents. In general, the input and output types of a Map/Reduce job is specified as follows:

$$(input) < k1, v1 > \rightarrow map \rightarrow < k2, v2 > \rightarrow combine \rightarrow < k2, v2 > \rightarrow reduce \\ \rightarrow < k3, v3 > (output)$$



We had to port the MapReduce implementation of WordCount application on to the mpiJava implementation. This included the implementing of the basic working principle, establish the required communication, distribute and coordinate the data among the various computing nodes. The pseudo-code of the implementation is as follows:

```
class WordCount {
    main(String [] args) throws MPIException {
        MPI.Init(args) ;
        if (MPI.COMM_WORLD.Rank() !=0){
            MPI.COMM_WORLD.Send(filecontent , 0, filecontent.length ,
                                MPI.CHAR,dest , tag );
        }
        else {
            Status s = MPI.COMM_WORLD.Recv(filecontent , 0,
                                            filecontent.length , MPI.CHAR,
                                            Source.rank , tag );
        }
        MPI.Finalize ();
    }
}
```

These are the basic and important communication that has to be executed on mpiJava for any type of application. We could then extend this to suit our needs by adding the specific commands to compute and execute the requirements for the corresponding application.

#### **2.4.2 BLAT Specification**

We make use of the open source BLAT MPI implementation for message passing technique. But had to perform a modification of transferring it from the original C language

implementation to mpiJava implementation. We then developed the MapReduce implementation of the BLAT application by including the following functions to the existing MapReduce framework. The main blocks that has to be added comprise of the Input reader, Map, Reduce and Output Writer. The pseudo-codes below depict the requirements for implementing the BLAT on MapReduce.

**Input Reader:** The user-defined code for the input formatting, reading and converting to key/value pairs follow this framework:

```
read( dir ){
    for each file in dir{
        emit( file , content )
    }
}
```

**Map:** The user-defined code for performing the mapping operation, namely the matching of database file to the contents of the query file and would result in a key/value pairs which are passed to the reduce function should follow this framework:

```
map( String key , String value ){
    query data : read from a file
    // key: database file
    // value: position of query matching in the database
    EmitIntermediate( query , position );
    //The key is ignored.
}
```

**Reduce:** The user-defined code would perform the process of combining the values for a particular key from the results obtained from the preceding Map function and it should

follow this framework:

```

reduce(String key, Iterator values){
    // key: query
    // values: a list of matching positions
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(values);
}

```

**Output Writer:** This is the function which does the operation of putting together the results of the reduce function and publish the result to the global file system. The user-defined code for the Output Writer should follow this framework:

```

write(values, dir){
    // values: a list of matching positions
    file = dir.create(" result")
    for each query, value in result{
        file.write( query + "=" + value)
    }
}

```

These were some of the additions and modifications that has to be incorporated to the existing MapReduce framework. This modifications would result in a parallel programming computing system which suited our applications needs.

## 2.5 Summary

In this chapter, we introduced the two parallel programming techniques, mpiJava and MapReduce. We discussed in detail the architecture and working of the techniques. With the architecture and the handling capabilities of the computing system, we obtain the maturity in choosing the application to match these parallel computing techniques. The changes pertaining to the parallel programming techniques to best suit our data intensive applications were dealt with in detail. Apart from this we learnt how we could benefit from our parallel computing by making use of these changes to fit the two applications. In the following chapters we would delve into the programming benefits and drawbacks faced as a result of the proposed modifications.

## **Chapter 3**

# **Comparison of Message Passing and MapReduce programming**

### **3.1 Introduction**

With data-intensive computing being the daunting challenge in the handling large amount of data to extract valuable information in real-time, parallel computing is the only possible solution. The growth of data is exponential. So we should have appropriate computing technique to acquire the best result. There has been numerous studies involving the parallel programming technique to reduce the total computing time.

Several strategies have to be taken into account before choosing the parallel programming technique. We have chosen to analyze the mpiJava and MapReduce programming techniques. In the previous chapter, we have detailed their architecture and the benefits of using each of the technique. We have also formulated a specification and algorithm required to be implemented using these two techniques to handle our data intensive application.

This chapter analyzes the data intensive applications which we implemented using Hadoop-MapReduce and mpiJava. It also describes the challenges we faced in implementing them.

## **3.2 Characteristics**

The first step in developing parallel programming system is to first understand the problem that we wish to solve in parallel. Very often, manually developing parallel codes are time consuming, complex, error-prone and iterative process. Whereas, in the fully automatic parallelizing, the compiler analyzes the source code and identifies opportunities for parallelism determines if parallelism would actually improve performance. Loops are the most frequent target for automatic parallelization. In this section we would brief, the important characteristics of a distributed parallel computing system.

### **3.2.1 Communication**

The tasks assigned for a partition are intended to execute concurrently but cannot, in general, execute independently. The parallel computation to be performed in one task will typically require some data associated with another task. Such associated data must then be transferred between tasks so as to allow computation to proceed. This information flow in the parallel computing technique design is specified in the communication phase of a design. In the message passing, communication is important for the sending of messages to recipients. Some forms of messages include function invocation, signals, and data packets. The MPI relies on fine-grained communication and are explicit and generally quite visible and under the control of the programmer. Factors to be considered when designing the program's inter-task communications are cost of communications, latency vs. bandwidth, visibility of communications, synchronous vs. asynchronous communications, efficiency of communications, overhead and complexity. In the case of MapReduce, communication is required only between the master and the worker, there is no inter-worker communication and thus, coarse-grained. The run-time system takes care of managing the required inter-

machine communication.

### **3.2.2 Partitioning**

The partitioning of data and tasks in developing a design is intended to expose and extend the opportunities for parallel execution. Hence, the focus when employing a partitioning techniques is on carefully defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a large problem. Message passing doesn't have an automatic partitioning technique. Starting from the assignment of number of process required, ways to partitioning the data, allotment of the partitioned data and task to a particular process is all done by the developer. This may be an advantage when speaking about the control over the program but a major disadvantage when looking at in terms of the workload on the developer. On the other hand, MapReduce automatically parallelizes the computation across large-scale clusters of machines. The run-time system takes care of the details of partitioning the input dataset, and scheduling the program's execution across a set of machines. This allows programmers without any experience with parallel and distributed computing systems to easily utilize the resources and advantages of a large distributed system.

An important step in testing the programming is developing useful and comprehensive test data. Traditionally, we rely on partitioning to get parallelism in I/O, and to reduce network traffic. It can be considered a minimization of task idle time. Partitioning is a way to divide a large data sets into separately divided chunks; the contents of each chunk being determined by the partitioning technique. The multiple pieces of a large data sets give the data administrator a lot of flexibility and manageability in dealing with them. This problem can be compared to problems arising in real world work distribution processes like that of scheduling all tasks needed to construct a large building. Proper partitioning of data will

eventually boost performance. Load balancing is important to parallel computing systems for performance reasons. For example, if all tasks are subject to a barrier synchronization like in the case of message passing, the slowest task will determine the overall performance. Some of the objectives that has to be taken in to consideration when partitioning data are listed below;

- Whole task should be completed as fast as possible.
- Workers are very expensive to maintain and they should be kept busy. Depending on the amount of workload choose the number of workers.
- Distribution of tasks has to be well planned when there is little work. And planning plans a vital role, so can be done ahead of time.
- Work load should be fairly distributed. Each worker should carry out the same amount of work.
- Clever processing order has to be used when there are precedence constraints.

### **3.2.2.1 Load Balancing**

Load balancing for a parallel system is one of the most important problems which has to be solved in order to enable the efficient use of parallel computer systems. Load balancing partitions the data as near to equal size as possible, ensuring an even load across your all the processors. The boundaries are identified by determining the total size of the data and the actual number of the resulting partitions. The computational work of allotted for each processor should be balanced, so that no processor will be waiting for others to complete. This is the most common type of partitioning and is often useful for applications that manage historical data, especially data warehouses. When considering array/matrix



operations where each task performs similar work, even distribution of the data set among the tasks is very important. For loop iterations where the work done in each iteration is similar, we have to evenly distribute the iterations across the tasks. When considering a heterogeneous mix of machines with varying performance characteristics, we have to be sure to use some type of performance analysis tool to detect any load imbalances and adjust work accordingly.

### **3.2.2.2 Intelligent Partitioning**

Certain parallel programming tasks require load imbalances even if data is evenly distributed among tasks. Some of this types include sparse arrays, adaptive grid methods, N-body simulations and genomic matching. When the amount of work each worker will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a scheduler - task pool approach as in the case of MapReduce. As each worker finishes its work, it queues to get a new piece of work. In order to employ such partitioning technique, we have to design an algorithm which detects and handles load imbalances as they occur dynamically within the code. For the WordCount application, Load Balancing would suite to be best for consideration. As each of the worker would have nearly equal sized input file to work on. But when considering the BLAT query computation, this might not be work well. The large data set when partitioned equally may sometime result in splitting the existing query match sequence. When this happens we might not get this existing match in the output due to the improper partitioning employed. So for this type of critical conditions we came up with a solution where we would determine the boundary of the equal split. And from the succeeding partition replicate the beginning of the partition equal to the query length and append to the previous partition. Thus the resulting partitions size would be split length plus the query length.

### 3.2.3 Synchronization

In computer science, synchronization refers to one of two distinct but related concepts: namely synchronization of processes, and synchronization of data. *Process synchronization* refers to the concept of multiple processes joining up or handshaking at a certain point, so as to establish an agreement or commit to a certain sequence of action. It also refers to the coordination of simultaneous threads or processes to complete a task in order to get correct run-time order and avoid unexpected race conditions. Some types of synchronization include barrier, lock/semaphore, thread join, mutex and monitors. *Data synchronization* is the concept of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity view for all the processes. Some types of data synchronization include file synchronization, cluster file systems, cache coherency, RAID, database replication, and journaling. In most cases, process synchronization primitives are commonly used to implement data synchronization. Message passing utilizes barrier synchronization. This implies that usually all tasks are involved, and each task performs its work until it reaches the barrier, it then stops, or "blocks". When the last task reaches the barrier, all tasks are synchronized. Then the tasks are automatically released to continue their work. In MapReduce, the map task are carried out concurrently. When some of the map tasks finish, the partitioner and combiner function shuffles and re-sorts the output of map and sends it to the reduce function. Which then carried out its reduce task. When the last reduce task is done, the final step is to write the final output to the global file system.

### 3.2.4 Fault Tolerance

Fault-tolerance is a property that enables a parallel computing system to continue operating properly even in the event of the failure of (or one or more faults within) some of its components. If at all the operating quality the system decreases, it is proportional to

the severity of the failure, as compared to a natively-designed system in which even a small component failure can cause total breakdown. Fault-tolerance is particularly sought-after in applications requiring high-availability or life-critical. In case of failure, recovery from errors in fault-tolerant systems can be either roll-forward or roll-back. When the system detects that a failure, roll-forward recovery takes the system state at that time and corrects it, to be able to continue with further computation. Roll-back recovery reverts the system state back to some earlier, using checkpointing, and computations start from there. In MapReduce on a node failure occurs, if the failed worker has completed a map task, the task is reset back to the initial state, and the map task can now be assigned to any other idle worker. Similarly, any map task or reduce task in progress has to be reset to initial state and the task becomes eligible for rescheduling. But in message passing when failure occurs, the failed node's state is reset to that of last checkpoint. and all intervening computation are wasted.

### **3.2.5 Programming Languages**

A programming language is a structured mechanism for defining pieces of data and to establish communication to the machine. We can express any type of computations that can be performed by a machine. Programming languages can be used to convert any task to be performed on a machine by specifying the behavior of a machine, to express algorithms precisely, or as a mode of human communication. The message passing interface supports C, C++, Java and Fortran making the programmers from various background utilize the advantages of this technique. On the other hand, MapReduce supports Java, Hadoop pipes supports C++, Hadoop Streaming supports scripting languages like python and perl.

### 3.3 Programming Styles

This section elaborates the programming styles adapted for the applications in detail. We would brief each parallel programming technique for each of the data-intensive application.

***mpiJava implementation for WordCount:*** The technique adopted to implement the mpiJava version was the similar master/slave model. Message passing is well-suited to handling computations where a task is divided up into subtasks, with most of the processes used to compute the subtasks and a few processes (often just one process) managing the tasks. The manager is called the “master” and the others the “workers” or the “slaves”. When implementing using mpiJava programming required careful tuning. There was a considerable time that has to be spent for synchronization, data partitioning and parallelizing the code. But by considering parallel libraries in different parallel threads and avoiding overlapping of communication and computation, we could obtain maximum performance-oriented standard, trading off hardware latency and hardware bandwidth.

***MapReduce implementation for WordCount:*** We make use of the open source MapReduce WordCount [DG04, wor] example for our implementation and analysis. There was not much of programming that had to be done for this application. We only had to do a few modifications in the record reading feature of the input reader function.

***mpiJava implementation for BLAT:*** BLAT was initially a MPI implementation in C, and we had to port it to the mpiJava platform, and we had to spend some time parallelizing the code, on the number of machines chosen for the experimental setup. We had to carefully synchronize the machines and employ a partitioning technique manually to partition the input data. We also had to deal with the distribution of the data to the worker nodes.

**MapReduce implementation for BLAT:** Although MapReduce supports streaming and pipes, currently there is no support for using applications written in C language. BLAT is a bioinformatics genome tool written in C. So in order to use this application under the MapReduce model, the solution we came up with this is to write a SWIG wrapper class [Bea03], [Bea96] utilizing the Java Native Library(JNI) [Gor98] functions for the map and reduce tasks in Java, and use these wrapper functions to execute the BLAT genome searching. The input to the map function, database and the query files were placed in the GFS. The map wrapper function reads the database file, and according to the specified fileinput format creates the input split and stores them in the specified list of datanodes. Each map task will process one of these input splits and the resulting intermediate output will have the query matching done at the InputSplit received by the map function.

Overall, the new Mapper implementations are passed on to the new job configuration(*JobConf*) via the *JobConfigurable.configure(JobConf)* method and overrides it to initialize themselves. The framework then calls *map(WritableComparable, Writable, OutputCollector, Reporter)* method for each key/value pair in the InputSplit for that application. Applications can then override the default *Closeable.close()* method to perform any required cleanup and final chores. Output key/value pairs do not need to be of the same types as input pairs. A given input pair may map to zero or many output key/value pairs. Output key/value pairs are collected with calls to *OutputCollector.collect(WritableComparable, Writable)* and are written to the global file system.

This intermediate output is saved in the GFS. Hadoop's runtime collects these locations and sends them to the appropriate reduce tasks. The reduce tasks now have the partial output. The reduce tasks performs the operation of sorting the received output and merging them. The merged output file is then stored in GFS and the location is passed to the user program, which then performs a similar operation to merge all of them in to a single output

file. When programming for MapReduce the main concentration was on to getting the core application done rather than data partitioning, memory management, communication between jobs, synchronization parallelizing the code. All of these was taken care of by the Hadoop framework.

### **3.4 Summary**

In this chapter, we have presented new modifications to the existing parallel programming techniques. under different workloads and different applications. The key findings are that we have to choose parallel programming technique to suit our application requirements. We had also detailed about the importance of the proper partitioning technique to be used for data partitioning. Current parallel programming applications, partition data based on the database administrators knowledge of the data structure and is partitioned statically by the administrator. The novelty of the intelligent partitioning technique is the use of query length along with equal split length to create an optimal partition. The table 3.4 summarizes the basic characteristics comparison between mpiJava and MapReduce.

Characteristics	mpiJava	MapReduce
Communication	Fine-Grained.	Coarse-grained.
Partitioning	Done by the developer.	Handled by GFS and specified partitioning technique.
Synchronization	Message passing and barrier options.	Map jobs finish followed by Reduce jobs.
Fault Tolerance	Sensitive to failing components and relies on everything working correctly.	Reschedules the job on a node failure.
Programming Languages	Supports C, C++, Java and Fortran.	Supports Java, Pipes supports C++, Hadoop Streaming supports scripting languages like python and perl.

Table 3.1: Comparison of mpiJava and MapReduce characteristics

# **Chapter 4**

## **Quantitative Analysis**

### **4.1 Introduction**

Parallel computing techniques have been designed to handle large amounts of data to perform data-intensive computing in order to extract valuable information in real-time. As data growth has been exponential, the parallel computing environment has to be chosen with utter care. All of this computation become possible with the parallel computing technologies discussed in the previous chapter. Once the distributed parallel programming technique, data-intensive application and partitioning techniques has been chosen the next step is to evaluate the performance of the parallel computing for the specific application.

The experiments were performed with the parallel programming technique and data-intensive applications discussed in the previous chapter. This section details the experimental results for the data intensive applications run on the two parallel programming setup. The results have been discussed in two sections. The first section details the result from Word Count application executed on the message passing and MapReduce framework. The following section elaborates the execution results of BLAT application on message passing and MapReduce framework.

The experimental setup we used to test out our application is a 16 node cluster with



Fedora 9 Linux operating system version 2.6.25-14. Each of the node had Dual Core AMD Opteron(tm) Processor, with a CPU speed of 1800 MHz. The Mpi version used for message passing programs was openmpi-1.2.9. And the MapReduce programs were written in the Yahoo's open source version of Hadoop-0.18.3. Several experiments were conducted, with varying number of nodes and varying data set sizes. Some of our experimental results are tabulated and discussed in the following sections.

## **4.2 Timing Analysis**

### **4.2.1 WordCount**

We make use of the open source MapReduce WordCount [DG04, wor] example for our implementation and analysis. This application speedily does the counting of word frequencies in a collection of documents. The Word count application was ported on to message passing interface. The first step we had to perform was to decide on how many nodes will be executing the application. Then we had to do the distribution and managing of the data, synchronization and proper communication by manual programming. Finally, we incorporated the function of Word count as a class in the program. The synchronization and proper communication between the computing nodes required careful tuning in order to get the proper results.

For the WordCount application, we evaluate the overall performance of MapReduce and mpiJava implementations by measuring the total execution time as the number of computing nodes increases. For the WordCount application, the input file sizes were chosen to be 90MB and 5.4GB. The result is tabulated in Table 4.2.1.

From Figure 4.1 we observe that MapReduce performs fairly well for large input data sets, but when we evaluate the performance of MapReduce and mpiJava for the smaller data

No. of Compute Nodes	90MB Filesize		5.4GB Filesize	
	MapReduce (sec)	mpiJava (sec)	MapReduce (sec)	mpiJava (sec)
1	62.80	52.10	436.90	431.20
2	33.70	27.60	219.50	218.50
4	15.80	13.20	109.20	108.70
8	7.91	7.21	54.60	54.10

Table 4.1: WordCount - Execution time for MapReduce and mpiJava for different input file sizes

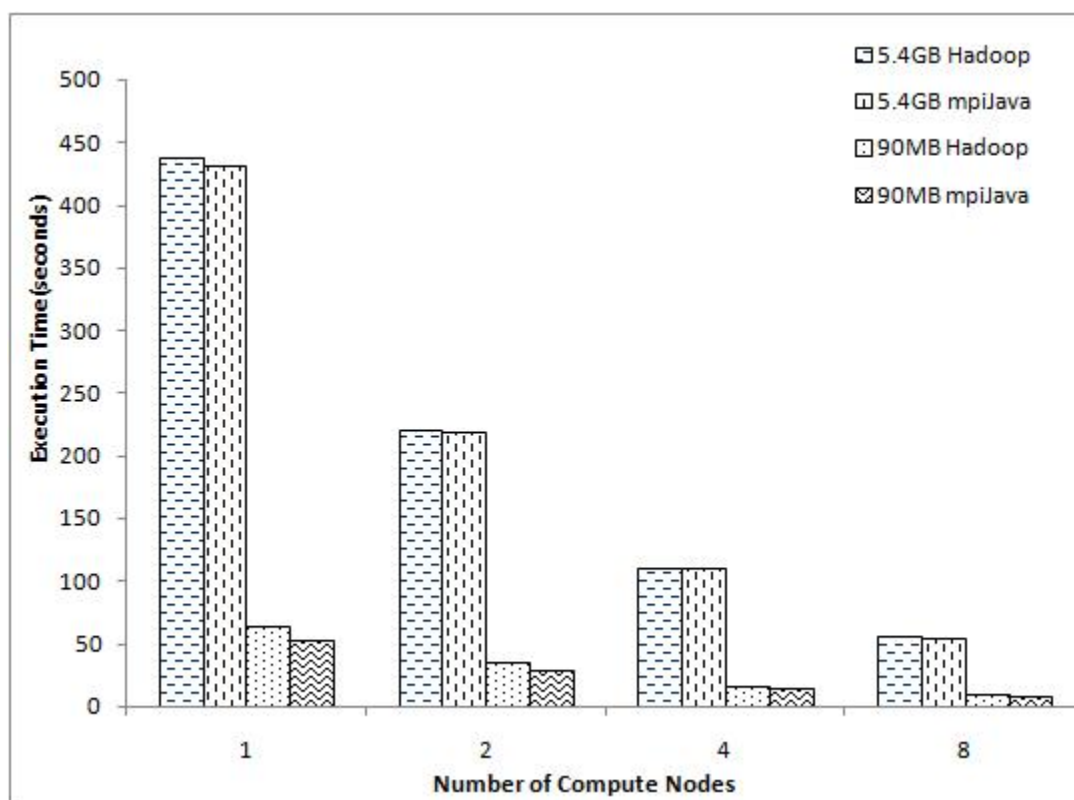


Figure 4.1: WordCount - Execution time vs Number of compute nodes - 90MB and 5.4GB input file

file we observe that there is a considerable overhead in MapReduce. Startup overhead is caused by the propagation of program to all worker nodes. This also involves the formatting of namespace in the NameNode, copying of the input data file to the GFS and to notify the map nodes about the job assigned. When we analyzed how the actual time was spent during a WordCount application run on a single compute node using MapReduce technique, it turned out that the startup overhead took about 10 sec of the total computation time. The actual Map job took about 49 sec and the shuffling, sorting and reducing took 2.9 sec. But when the same job was carried out on increased compute nodes, the overhead startup time was dominated by the reduced MapReduce computation, thus resulting in a performance similar to that of the mpiJava. Thus the startup overhead was nearly eliminated when we had 8 compute nodes.

#### **4.2.2 BLAT**

The second data intensive application to be considered is BLAT (BLAST Like Alignment Tool) [Ken, Ken02]. BLAT is one of the most widely used bioinformatics tool. This is commonly used to look up the location of a sequence in the genome or determine the exon structure of an mRNA and also performs a rapid mRNA/DNA and cross-species protein alignments. The BLAT was originally an MPI implementation in C language. We had to first port it on to mpiJava. We had used SWIG in order to get the conversion done from C language to mpiJava. we had to spend some time parallelizing the code, on the number of machines chosen for the experimental setup. We had to carefully synchronize the machines and employ a partitioning technique manually to partition the input data. We also had to deal with the distribution of the data to the worker nodes. The next step was to port the MPI implementation to the MapReduce framework.

For this MapReduce implementation of the BLAT application we had to include the

No. of Compute Nodes	938MB Filesize		1.9GB File size	
	MapReduce (sec)	mpiJava (sec)	MapReduce (sec)	mpiJava (sec)
1	102.31	100.30	844.4	842.7
2	53.80	49.30	329.2	326.1
4	26.08	26.01	214.1	213.6
8	13.10	12.98	106.3	106.1

Table 4.2: BLAT - Execution time for MapReduce and mpiJava for different input file sizes

following functions to the existing MapReduce framework. The main blocks that has to be added comprise of the Input reader, Map, Reduce and Output Writer. To evaluate the MapReduce and mpiJava technique for BLAT we first measured the total execution time it takes to process data by increasing the amount of data. The number of compute nodes were fixed at 8. This shows that as the data grows the total execution time it takes with MapReduce seems to be getting better than the mpiJava. Figure 4.2 depicts the result.

We performed another benchmark to see how MapReduce and mpiJava implementations scale as the number of computing nodes increases. For the BLAT application, the input file sizes were fixed as 938 Megabytes(MB) and 1.9 Gigabytes(GB). This was run on a cluster of 8 computing nodes, increasing in order of 1, 2, 4 and 8. The results are tabulated in Table 4.2.2. As we increase the nodes the number of map tasks were also increased so that each of the map task processes almost equal amount of data in every run.

MapReduce and mpiJava both show similar performance. The overhead induced by the MapReduce implementations has negligible effect on the overall computation as the amount of data accessed in each analysis is large. Figure 4.3 highlights the scalability of the MapReduce technique. This also indicates how the speedup is achieved with the increase in number of compute nodes.

### **4.3 Programming Comparison**

The Message passing and MapReduce frameworks were had their own advantages and disadvantages when used for executing an application. The Message passing required the programmer to explicitly establish communication between the master-worker and worker-worker. The partitioning of the data and the allocation of task to a worker has to be explicitly specified by the programmer. The synchronization task and need to tolerate failure was a great challenge. Other than implementing the actual function, all of these had to

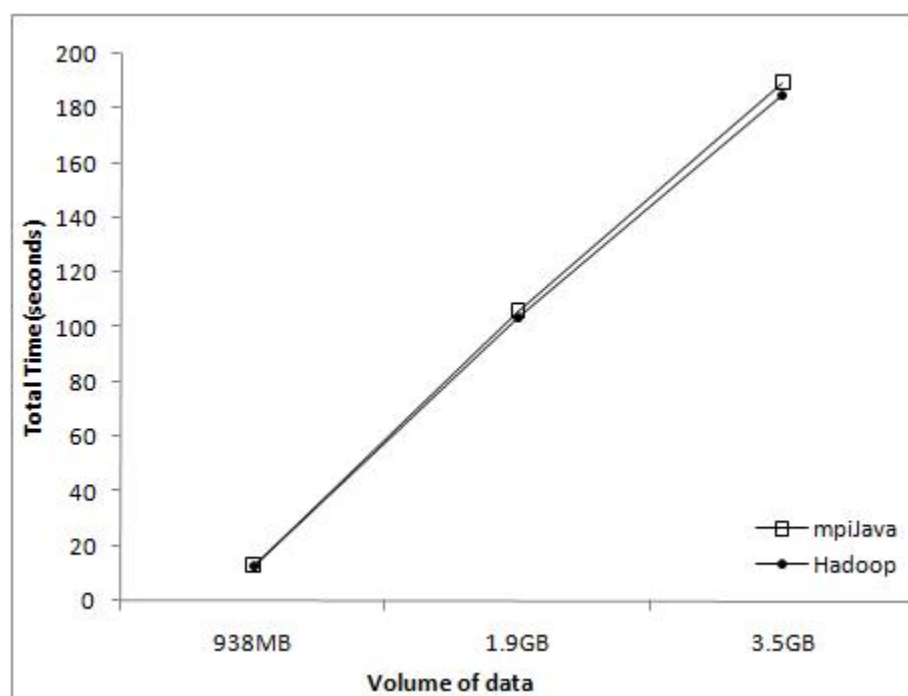


Figure 4.2: BLAT - Execution time vs Volume of data on 8 compute nodes

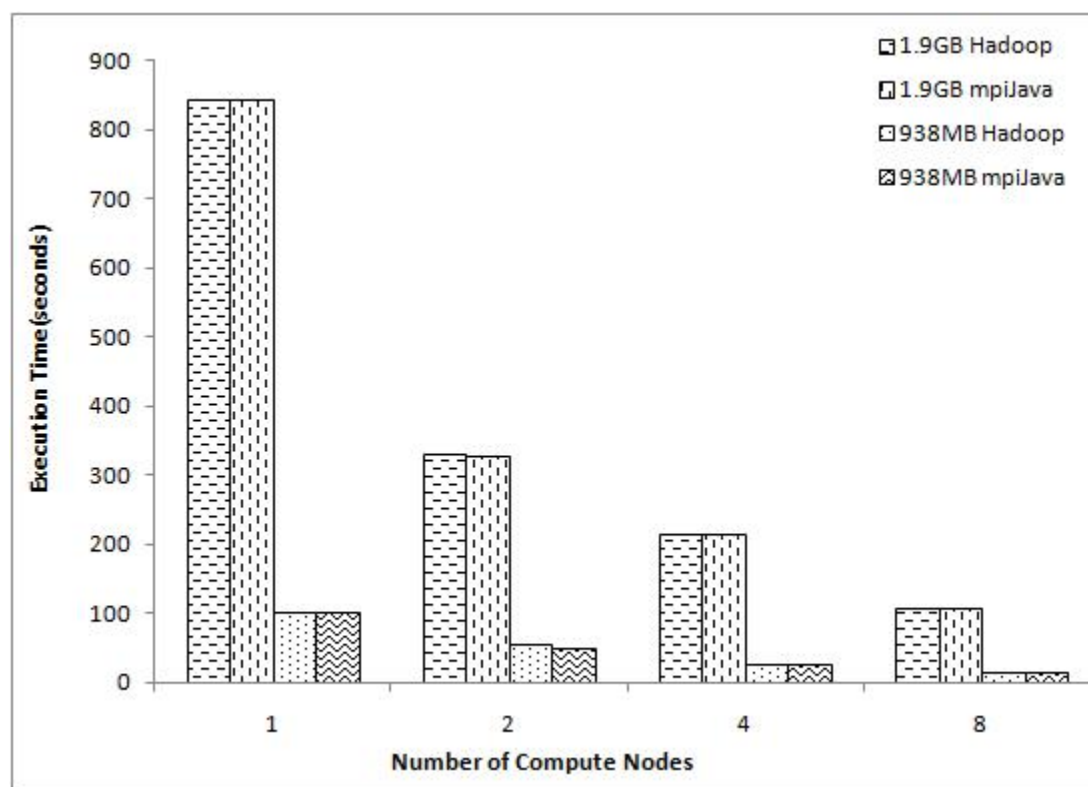


Figure 4.3: BLAT - Execution time vs number of compute nodes - 1.9GB and 938MB input file



be taken care of to establish a perfect working message passing environment for a specific application. The advantage of using message passing is that it can be used for most of the applications and does support numerous programming languages as described in the previous sections.

When programming with the MapReduce framework, the communication, synchronization, partitioning and fault tolerance was automatically done by the MapReduce framework. So the main concentration of the programmer was to implement the actual functions of the application. These have reduced the loads on the programmer and result in focus. Although, they provide advantages, there happen to be drawbacks. Since the MapReduce framework employs implicit communication, it cannot be employed to applications requesting inter-worker communication. It also can't be used for applications that don't employ data partitioning. The following Table 4.3 gives in a glimpse the comparison of message passing and MapReduce in terms of the total number of lines of code, the length of common areas and percentage reduction in the code in MapReduce compared to message passing.

No. of lines of code	<b>mpiJava</b>	<b>MapReduce</b>	<b>Common Code</b>	<b>% reduction</b>
WordCount	82	46	15	53.7%
BLAT	1870	1582	1100	37.4%

Table 4.3: Programming comparison of MapReduce and mpiJava

## 4.4 Failure analysis

To evaluate the MapReduce and mpiJava performance when a node fails during the execution of BLAT application, we first measured the total execution time it takes to process data with the number of compute nodes fixed at 8. Then the experiment was repeated with one of the nodes failing during runtime. In case of mpiJava, the system didn't finish the job. We had to wait for the failed node to resume and had to reschedule and restart the job again. When the experiment was run on a MapReduce system, the master, when it found a failed node, waited for one of the workers to finish and rescheduled the failed job to this worker. The results are tabulated in Table 4.4.

## 4.5 Summary

The parallel programming techniques, mpiJava and MapReduce designed for the data-intensive computations, WordCount and BLAT seemed to fair well. With the mentioned modifications from the previous chapter, to the existing parallel programming techniques, experiments were performed under different workloads and different conditions. The WordCount did fair well for larger files than for the smaller data sets with the MapReduce framework. Whereas the BLAT application which was a matching algorithm seemed to perform well with increasing data set and fixed compute nodes. The experiment conducted with a failing node was the highlight of all this analysis. The mpiJava didn't complete the assigned job when a node failure occurred. But MapReduce did fair well even in the case of failing nodes. But it did require some extra compute time in order to complete the rescheduled job.

Filesize	Time without node failure (sec)	Time with node failure(sec)
100MB	3.43	5.27
938MB	13.10	21.43
1.9GB	106.30	133.11
3.5GB	189.51	206.30

Table 4.4: BLAT - Execution time for MapReduce with and without a node failure for different input file sizes on 8 compute nodes

## **Chapter 5**

# **Active storage in Message Passing and MapReduce**

### **5.1 Introduction**

An important trend in the design of a storage subsystem is to provide an environment for executing applications directly at disk drives. It also provides the capability of increasing the intelligence at storage devices. Active disks happen to be the future of the disk drives. By utilizing network-attached storage, we have an opportunity of executing many requests directly at the storage devices without the intervention of the server. By partitioning the processing at the storage disks, makes possible efficient scheduling and reduce the load of interconnection network.

### **5.2 Active Storage with Object Based Devices**

In this section we detail an object oriented framework for supporting applications that benefit from an Active Storage system [JRC08], such as the Active Disks mentioned previously. This framework can support a variety of data-intensive applications. In the following sections, we describe the basic working of object based devices(OSD). Some of the earlier work on Active Disks follows from network-attached secure disks (NASD), in which computational power at the storage level for parallel and network file system functions are ex-

ploited, as well as traditional storage optimizations [GNA<sup>+</sup>97, GNA<sup>+</sup>98]. The NASD work experimented on several types of applications that can benefit from Active Disks - including filters, multimedia, batching, and storage management - and enumerated the challenges to providing an execution environment on commodity disk drives [Rie99, RGF98].

### **5.2.1 Object-based Storage Devices**

The Object-based Storage Devices (OSD) specification [ANS08] has introduced a new set of device-type specific commands into the SCSI standards family. The specification defines the OSD model and their required commands and command behavior.

Figure 5.1 illustrates the basic model of OSD in comparison to a traditional block-based device model for a file system. The traditional functionality of file systems is divided mainly to achieve increased intelligence in such storage devices. Object-based Storage devices have the capability of managing their storage capacity and exposes a file-like storage interface, “objects”, to their hosts. These objects behave exactly like files.

#### **5.2.1.1 OSD Objects**

Storage objects that are used to store regular data are called user objects in the OSD specification. There are three other kinds of objects defined within the specification, namely, root objects, partition objects and collection objects. There is one root object for every OSD logical unit [ANS04]. The root object forms the top most module of an OSD logical unit. User objects are put into partitions that are represented by partition objects. The maximum number of partitions permissible within a logical unit is given by a quota defined for the logical unit in the root object. Every user object belongs to one and only one partition. Collections are represented by collection objects. Each collection object belongs to one and only one partition and may contain zero or more user objects belonging to the same par-

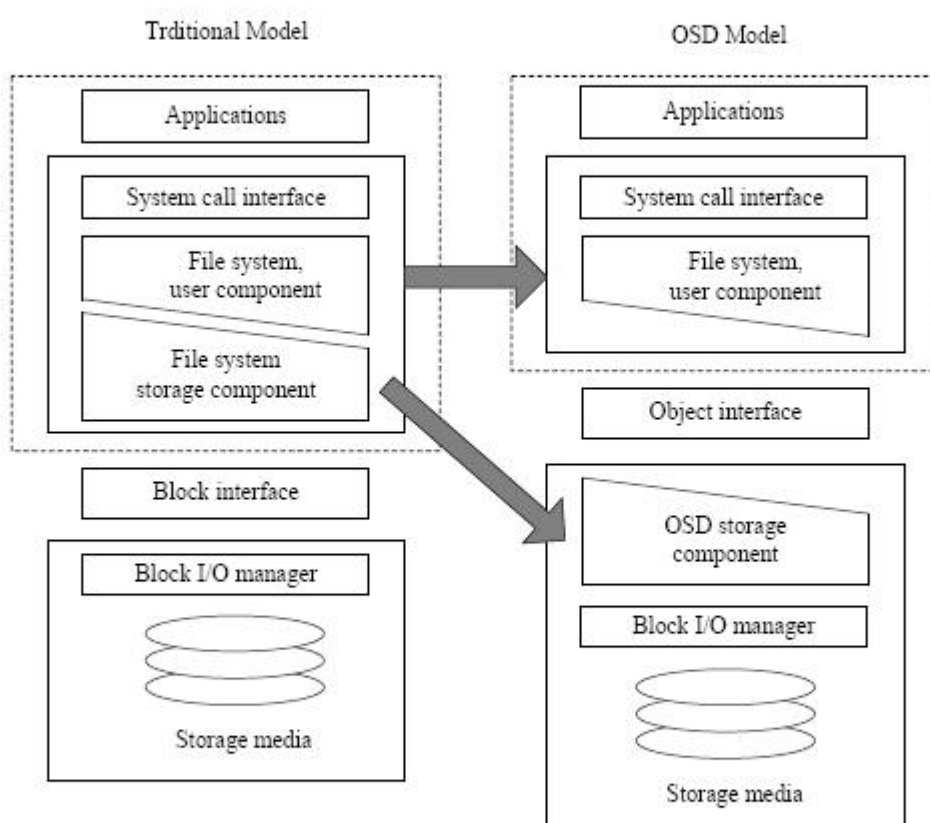


Figure 5.1: Comparison of traditional and OSD storage models [DHH<sup>+</sup>06]

Partition_ID	User_Object_ID	Object type
0	0	root object
$2^{20}$ to $2^{64} - 1$	0	partition object
$2^{20}$ to $2^{64} - 1$	$2^{20}$ to $2^{64} - 1$	collection/user object

Table 5.1: Object identification numbers

tion. The unique combination of the Partition ID and the User Object ID identifies each OSD object as illustrated in Table 5.1. The ranges not specified in the table are reserved.

#### 5.2.1.2 Object Attributes and Attribute Pages

Object attributes are used to associate meta data with any OSD object, i.e., root, partition, collection or user. Attributes are organized in pages for identification and reference. Attribute pages associated with an object is uniquely identified by their attribute page numbers ranging from 0 to  $2^{32} - 1$ . This page number space is divided into several segments so that page numbers in one segment can only be associated with certain type of object. For instance, the first segment from 0x0 to 0x2FFFFFFF can only be associated with user objects.

Attributes within an attribute page have similar sources or uses. Each of them has an attribute number between 0x0 and 0xFFFFFFFFFE that is unique within the attribute page. The 0xFFFFFFFFF is used to represent all attributes within the page when retrieving attributes.

#### 5.2.2 OSD commands

The OSD commands execution follows a request-response model as defined in SCSI Architecture Model (SAM-3) [ANS04]. This model can be represented as a procedure call as following: The OSD commands use the variable length Command Descriptor Block(CDB) format defined in SPC-3 but have a fixed length of 200 bytes. An opcode 0x7F in CDB dif-

ferentiates the OSD command from commands of other command sets. The service action field in the CDB specifies one of the twentythree OSD service requests defined in the OSD specification. Every CDB has a Partition ID and a User Object ID, to identify the requested object in a logical unit. Any OSD command may retrieve attributes and any OSD command may store attributes. Twenty-eight bytes in CDB are used to define the attributes to be set and retrieved. Two other common fields in CDB are capability and security parameters.

When the execution of an OSD command results in errors, it returns a status of CHECK CONDITION, along with sense data to report errors generated in OSD logical units. The sense data contain information that allows initiators to identify the OSD object in which the reported error was detected. Any error can be reported by including the appropriate sense key and additional sense code to identify the condition. The OSD specification uses descriptor format sense data so that several sense data descriptors can be returned together.

## **5.3 Basic Approach**

To capitalize on the parallelism in an active storage system, we designed and developed a Record Storage Application. The objective of this application development was to exemplify a framework that supports the parallel execution of any kind of application, and to analyze the performance gains of object oriented Active Disk computation over traditional single server systems.

### **5.3.1 Design Methodology**

The basic design principles of developing an application to run in an Active Disk environment are to exploit parallelism in the data-intensive portion of the data processing, separate the code to be executed at the disks as much as possible, from the code that controls the execution flow, so as to form self-contained and manageable units, and take advantage



of variations in available resources during execution, by using primitives. These three principles allow for a great deal of performance and flexibility while placing and executing the application code. The Active Disk application code is divided such that the core piece of code will be made to run at each of the drives, while the high-level synchronization, control, or merging code is made to run at the host. This forms a client-server parallel programming model as illustrated in Figure 5.2.

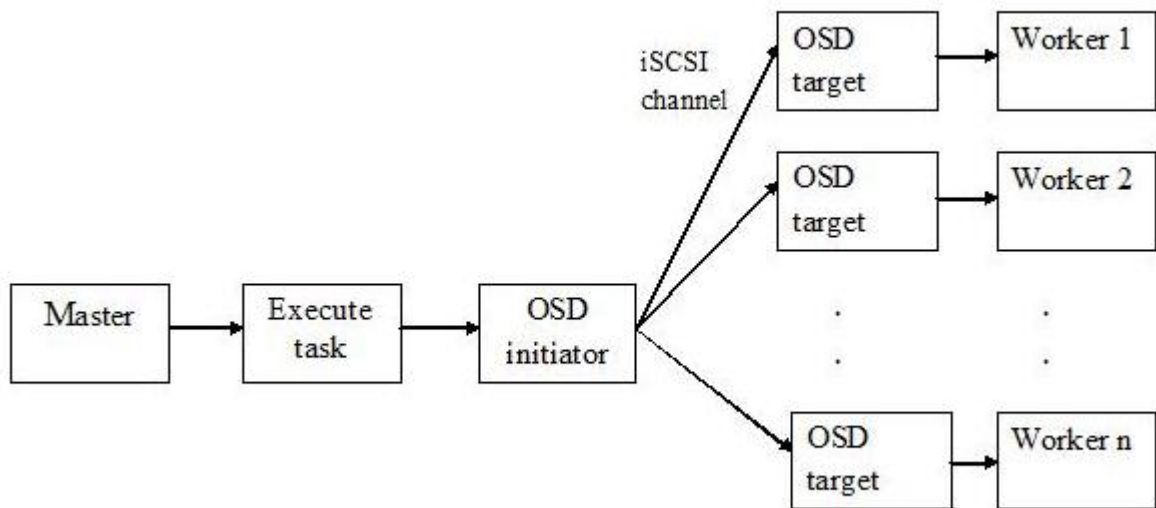


Figure 5.2: Basic structure of Active Disk computation via OSD

Initialization of input parameters takes place at the host and is sent across to all of the disks. Each of the disks operates on its own local data and produces its portion of the final result. The host collects the results from all the disks and merges them in an appropriate manner to form the final result. One of the major objectives of this work was to be able to facilitate the transfer of program files, related to an object type under consideration, to a OSD target which has no prior knowledge about that object type. To perform this operation, we chose to implement our application in Java, allowing methods pertaining to each object

be encapsulated within its own class. For the OSD target to execute methods, this Java code must be downloaded to the target and then executed. This class file transfer is achieved by setting an attribute on the object where the attribute data is the JAR file associated with the class.

The class file, associated with the object as an attribute, defines the methods for that object. A Java server runs on each OSD target, which takes in the requests coming from the client and first determines the type of object that the client request is meant for. Then the corresponding objects methods are executed. In the case of a class file transfer, the Java server first downloads the transferred class file to an appropriate location and then uses the newly downloaded class to invoke the objects methods. The transfer of the OSD attribute page was performed as a part of work. This determines the type of object to be executed.

### **5.3.2 Sample APIs Developed**

Two main functionalities that have been implemented in the application are the record sorting and matching record search functions, which are expected to benefit the most from such active computations. We use at the client a variant of Merge Sort algorithm, which is the most common out-of-core sorting methodology. Merge Sort is based on the assumptions that: 1) a small list will take fewer steps to sort than a large list , and 2) fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. The algorithm performs a sequence of in-memory sorts on small subsets of the data, followed by a series of merge phases that combine the sorted subsets until the entire dataset is in order.

### 5.3.3 Implementation

This section provides an outline of the structure of the application described above, that execute on Active Disks, including the design philosophy, the structure of the on-drive code and the host code. Our implementation is based on the iSCSI OSD reference implementation developed by the University of Minnesota [DHH<sup>+</sup>06]. The reference implementation is built using Linux kernel modules and user-level libraries. The kernel modules are used to implement a primitive OSD file system and the user-level code provides the communication channel and the implementation of the OSD target semantics. Since our framework does not require the use of the OSD file system, our implementation is entirely user based.

#### 5.3.3.1 Client Design

The client side of our application was designed to have APIs for multiple operations on an object, that can exploit the parallelism offered by such a framework. Continuing with the above discussed examples of record sorting and searching, the splitting of the application code is done as follows. Each target sorts the data set it contains for the particular database object, in parallel, and transfers the sorted result back to the client. The client receives the sorted data sets from each of the targets and merges them together to produce the final sorted data set. This scheme of computation totally eliminates the need to transfer the data to the client for the sorting phase, as it is done directly at the target. In the case of record search, each target searches for records matching the given search criteria and sends back just the resulting records back to the client. The client, just as in the case of sorting, collects the partial result records from all the OSD targets and combines them together to produce the final search result. Since filtering of the data occurs at each of the targets, the data that gets transferred across the network is just a fraction of the initial data set, thereby reducing network bottleneck.

### **5.3.3.2 OSD Target Server Design**

The target server is designed to have a dedicated process, constantly listening to requests being sent from the client. In order to ensure the parallelism inherent with Active Disk applications, the client spawns a thread for each server, while sending requests. The individual server threads, send the requests to the client, and waits to receive the response from their respective targets. The `ActiveServer` class receives the client requests and calls the appropriate function in the `RequestHandler` which was downloaded from the client. The `RequestHandler` class has the methods for the object as described in the previous section. In order to extend this to support multiple object types, a separate `RequestHandler` class has to be created for each object type. The appropriate `RequestHandler` class is then called by the `ActiveServer` class depending in the object type. This extensible architecture makes it very easy to add new objects and their functions to the framework. And as described above , the class file transfer capability allows the newly created `RequestHandler` classes of the objects to be transferred to the targets, along with the OSD commands. Thus extending the framework is made to become a trivial task.

The protocol used here for the client server communication is iSCSI. Configuration files have also been made use of, for specifying server parameters such as server names, port numbers, and directory paths for the object creation and so on.

### **5.3.4 OSD Design Changes**

The OSD implementation used had to be tailored to fit the requirements of the application, in quite a few areas. The main idea adopted here was to relate the API of the object to an OSD command, such that issuing the OSD command from the OSD initiator at the master to an OSD target at the worker, would be equivalent to sending a request from the client to a server via a TCP/IP socket connection. we could implement by adding a new OSD

EXECUTE METHOD command for these APIs, in order to keep the method execution code self contained.

The interface between the Java application program client and the OSD client can be implemented as data pipes. The same holds for the interface between the OSD target and the Java server program. When the application client issues a request, the “PARAEXECUTE” module receives the request and packages it into an appropriate EXECUTE METHOD command CDB. Application level parameters, if any, are placed as object attribute values in the CDB, and so is the class file data. Once the CDB reaches the OSD target, it is decapsulated and the parameters are passed to the java server, which takes up the execution of the requested method.

### 5.3.5 Results

The experimental setup we used to test out our application is a 16 node Linux cluster. One node was set up as the client machine and up to 8 nodes were used as OSD targets. Several test runs were conducted, with varying number of targets and varying data set sizes. Some of our experimental results are tabulated as below.

Table 5.2 show the results for a data set of 1 million records. The total sort time is the time required for the client to issue sort execute methods to the targets and then retrieve the data and do a merge sort locally. The average sort time is the average time required on each target node to do the read from disk and then sort. The results show that the targets sort times decrease linearly. The total sort time shows improvement but is not linear with respect to the number of target nodes. The reason for this is that the Java virtual machine begins to run against its heap size limit and as a result the merge time takes longer than expected.

We also measured the effect of using iSCSI as an RPC transport mechanism rather than

No. of OSD Targets	Total Sort time (ms)	Avg. Target Sort time (ms)
1	19931	16689
2	12645	8716
4	8761	4227
8	7437	2351

Table 5.2: Results for 1M Records

a purely TCP/IP socket based implementation. We constructed a socket-based implementation that was hard coded to perform application-specific RPCs. Through experiments, we found that the iSCSI OSD implementation is comparable in run-time to the socket-based implementation with no more than 10% degradation in performance for the iSCSI implementation.

## 5.4 Active storage in message passing and MapReduce

Active Storage moves I/O-intensive tasks from the compute nodes to the storage nodes to profit their resources. The main benefits include local I/O operations, very low network traffic, and better overall system performance. Active Storage enables scientific applications to exploit underutilized resources of storage nodes for computations involving data located in secondary storage. As we have learnt in the earlier sections, the message passing and MapReduce framework fairs well for standard clusters. The message passing and MapReduce perform a similar operation of data computation designed to be carried out closer to data, than transferring data over the network. In this section we would see how active storage would benefit message passing and MapReduce frameworks.

Instead of executing all application processing at the host, and forcing all the raw data to move from the disk drives, through the storage interconnect, and to the host before pro-

cessing, Active Disk applications execute concurrently on both the host and the disks. The “core portion of the applications data-intensive computing is separated and executed in parallel across all the storage disks in the system. This has been the most promising aspect of Active Disks, which enabled to leverage the parallelism in highly concurrent workloads by striping across a large number of drives. This ability to perform on-drive computations should be localized to small amounts of data, essentially performing a little processing as data “ streams past” from the disk back to the network and the host. Some of the applications that may benefit from Active disk concepts are filtering (search, sort, matching), storage management (backup, organization and optimization), specialized support (locks , transactions) and real-time (audio streaming, video server).

#### 5.4.1 Architecture Overview

**Message passing** The basic structure for active storage implementation on message passing is shown in Figure 5.3. The Master side of the message passing should be designed to have APIs for multiple operations on an object, that can exploit the parallelism offered by such a framework. The MPI\_Send and MPI\_Recv command in the message passing should be used for hand-shaking. The attributes of which method has to be executed at the target are sent from the master. The worker on receiving the attributes and method to be executed, performs the task and send the result back to the master. The Message passing framework follows SIMD model. So it can benefit from the active storage setup. But the major disadvantage of active storage is the absence of Inter-worker communication. As the message passing requires frequent inter-process communication we have to implement extra modules in order to achieve this when using active storage for message passing.

**MapReduce** The basic structure for active storage implementation on message passing is shown in Figure 5.4. In MapReduce, the map task that need to be executed at the

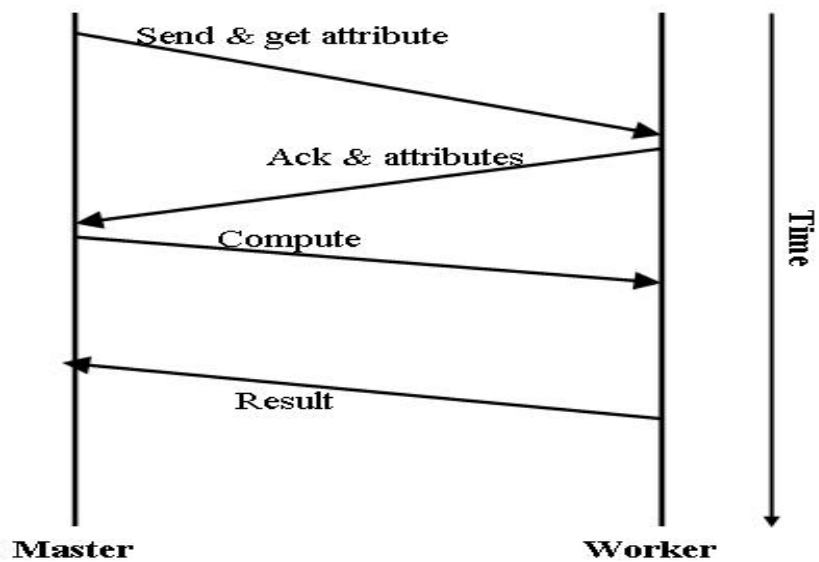


Figure 5.3: Active storage programming model for message passing

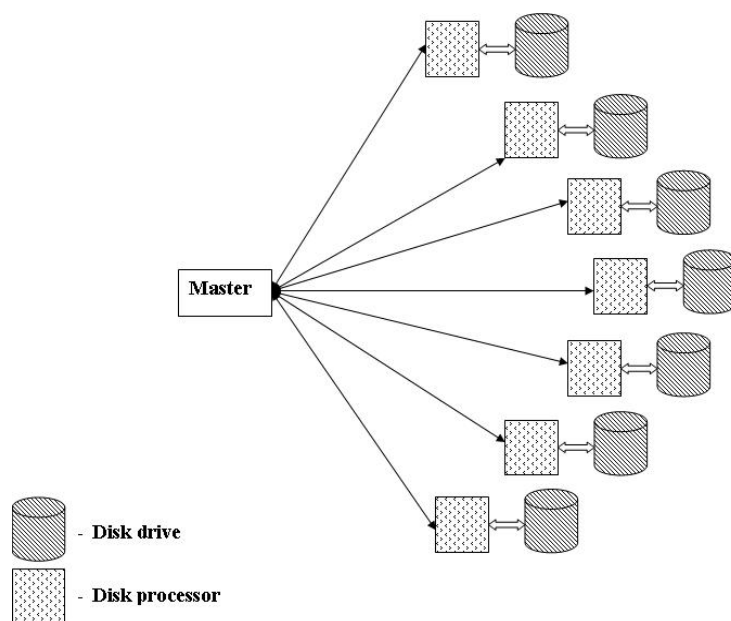


Figure 5.4: Active storage programming model for MapReduce



target could be encapsulated in to an archive and transferred over to the target. Then we could use method and attributes from the archive and execute the task allotted for the worker. It receives the Map code from the host and does the map task with the data portion on the local disk and then return the output back to the Master via the channel. As MapReduce doesnt require inter-worker communication, so it can benefit greatly from active storage concept. One of the major disadvantage when using MapReduce framework is the running of a reduce task. This could be solved in either of the ways. We could either perform a reduce task in two steps or a single step. For the two step case, we could assign a reduce task at each of the worker nodes which ran a map task, which means the reduce task could again work on local data. Then the result from each of the reduce job is transfered back to the master node and run a overall Reduce task at the master. In the single step reduce case, we could transfer all the result from each of the map task to the master and perform the reduce task in one step at the master.

#### **5.4.2 Application**

The climate model prediction seems to be a good application for using active storage on message passing and MapReduce framework. The data formats of the climate model is in the form of striped and netCDF files formats. The striped files are broadly used in active storage for performance. Active Storage efficiently deals with both types of files. The storage nodes with relevant data only run processing components. Processing components only compute on local chunk of available data. Active Storage efficiently deals with both striped and netCDF files, eliminating  $> 95\%$  of the network traffic.

The pseudocode for the execution of the netCDF format files from the climate prediction application on message passing and MapReduce framework is as illustrated below:

```
<?xml version="1.0"?>
```

```

<rule>
  <stdfiles>
    <stdout>/path.out-${NODENAME}</stdout>
  </stdfiles>
  <match>
    <pattern>/home/query.txt</pattern>
  </match>
  <program>
    <path arch="any">/home/climate.py</path>
    <arguments>/path arg </arguments>
  </program>
  <mapper>
    <path arch="any">/home/climatemapper.py</path>
    <arguments>/path arg ${CHUNKNUM} ${CHUNKSIZE}</arguments>
  </mapper>
</rule>

```

**Key findings** The Active Disks can be tailored to fit the requirements of any application. The main idea that has to be adopted here was to port the raw data computation on to the disks rather than doing it at the network end. For in the case of MapReduce, When a map task is assigned to a node by the master, we could port the map function to be executed to that assigned host. In that case the remote reading of data over the network could be avoided. Thus the resulting network contention when employing large cluster of nodes could be eliminated. One main place where this would not work is the reduce phase, the reason is that the data for a reduce phase resides on multiple machines and we have to perform a remote read in order to execute the reduce function.

The same concept would work well when employed for message passing. But the major disadvantage of the active storage concept being the absence of inter-node communication. There is a lot of communication between the Master which issues the task and the Worker which executes the command. But there is negligible communication between the workers. This would work well for the MapReduce framework where there is no communication within the workers. But this is not favorable for the message passing. There has to be additional programming that has to be employed for using the active storage concept in message passing.

Similar to the message passing and MapReduce framework, the Active Disk application code can be divided such that the core piece of code will be made to run at each of the drives, while the high-level synchronization, control, or merging code is made to run at the host.

## **5.5 Summary**

Remote execution of any application at the storage allows filter operations to be performed closer to the data, allows exploitation of cycles at storage. This also supports management functions customized and updates without requiring the firmware upgrades. All these features contribute to an improved individual application performance with an increased scalable system. They also seem to match well with the MapReduce and message passing framework defined in the previous chapters.

## **Chapter 6**

### **Conclusion**

In this paper, we have presented our analysis of distributed parallel programming techniques MapReduce and mpiJava for data intensive computing applications. BLAT represents a data intensive computation that can be implemented in MapReduce to gain scalability and efficiency. We have used our implementations to analyze up to 2GB of data. WordCount illustrates that MapReduce can be slower due to the introduced startup overhead for smaller datasets. However, this overhead becomes negligible as the input data size grows. We analyzed the distributed programming techniques with 5.4GB of data.

Comparing the performance of the two data intensive applications using MapReduce and mpiJava led to the following observations:

- Most data-intensive computation, can benefit from MapReduce technique to achieve speedup and scalability.
- As the amount of data and computation increases the startup overhead induced by MapReduce diminishes.
- MapReduce provides valuable fault tolerance. Even if a datanode fails due to the replication feature, the computation could be successfully completed. But in the mpiJava when a link is broken the whole computation will have to be rerun again.

- MapReduce Provides Coarse-Grained Parallelism. So the I/O or network access does not create a bottleneck.
- mpiJava is very sensitive to number of failing components thus programming requires careful tuning.

Our experience shows that mpiJava could be used directly for data intensive computations. However, we strongly believe that some features such as fault tolerance, ease of parallelization, usage of known programming languages, easier scalability, great flexibility in placement, and scheduling exhibited by MapReduce could be effectively used by most of data intensive computations which fall under the MapReduce model.

In this thesis, we have also analyzed how the active disks can be well incorporated into the message passing and MapReduce framework. In addition, we have explored Active Disks and in particular, the Object-based Storage Device (OSD) standard which provides a major step towards enabling explicit application awareness in storage systems behind a standard, fully interoperable interface.

# Bibliography

- [ANS04] ANSI. **Information Technology - SCSI Architecture Model - 3 (SAM-3)**, September 2004.
- [ANS08] ANSI. **Information Technology - SCSI Object Based Storage Device Commands -2 (OSD-2)**, January 2008.
- [Bar] Blaise Barney. Message passing interface tutorial. <https://computing.llnl.gov/tutorials/mpi/>.
- [BCKL98] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. mpijava: A java interface to mpi. In **In Proc. First UK Workshop on Java for High Performance Network Computing**, 1998.
- [Bea96] David M. Beazley. Swig: an easy to use tool for integrating scripting languages with c and c++. In **TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996**, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.
- [Bea03] D. M. Beazley. Automated scientific software scripting with swig. **Future Generation Computer Systems**, 19(5):599–609, 2003.
- [BFG<sup>+</sup>95] Chaitanya Baru, Gilles Fecteau, A Goyal, H Hsiao, A Jhingran, S. Padmanabhan, G. P. Copeland, and W. G. Wilson. Db2 parallel edition. **IBM Systems**, 1995.
- [Ble96] G. E. Blelloch. Programming parallel algorithms. **Communications of the ACM**, 39:85–97, 1996.
- [Bry] Randy E Bryant. Data-intensive scalable computing. <http://research.yahoo.com/files/BryantDISC.pdf>.

- [CFKL] Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko, and Sang Lim. mpi-java 1.2: Api specification. <http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html>.
- [CFZL97] Bryan Carpenter, Geoffrey Fox, Guansong Zhang, and Xinying Li. A draft Java binding for MPI., November 1997. <http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html>.
- [CGJ<sup>+</sup>98] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPI for Java: Position document and draft API specification. Technical Report JGF-TR-3, Java Grande Forum, November 1998. <http://www.javagrande.org/>.
- [CGJS00] B. Carpenter, V. Getov, G. Judd, and A. Skjellum. Mpi: Mpi-like message passing for java. In **In Proc. Concurrency, Practice and Experience**, volume 12(11), pages 1019–1038, 2000.
- [DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In **Proceedings of the Sixth Symposium on Operating System Design and Implementation**, pages 137–150, 2004.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In **COMMUNICATIONS OF THE ACM**, volume 51, pages 107–113, January 2008.
- [DHH<sup>+</sup>06] D. Du, D. He, C. Hong, J. Jeong, V. Kher, and Y. Kim. Experiences in building an object-based storage system based on the osd t-10 standard. Technical Report DTC-13-006, Digital Technology Center, University of Minnesota, 2006.
- [dic] Data intensive computing. <http://dicomputing.pnl.gov/>.
- [EPF08] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In **4th IEEE International Conference on e-Science**, pages 277–284, 2008.
- [Fer98] Adam J. Ferrari. Jpvm: Network parallel computing in Java. In **ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998**, Concurrency: Practice and Experience, 1998. To appear.
- [FK98] I. Foster and N. Karonis. A grid-enabled mpi: Message passing in heterogeneous distributed computing systems. In **Proceedings, 1998 SC Conference**, November 1998.
- [Fox97] Geoffrey C. Fox. Java for computational science and engineering - simulation and modelling ii. In **In Proc. Concurrency, Practice and Experience**, volume 9(11), November 1997.

- [Fox98] Geoffrey C. Fox, editor. **ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998**, Concurrency: Practice and Experience, 1998. To appear.  
<http://www.cs.ucsb.edu/conferences/java98>.
- [GFB<sup>+</sup>04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In **Proceedings, 11th European PVM/MPI Users' Group Meeting**, pages 97–104, Budapest, Hungary, September 2004.
- [GFHM98] Vladimir Getov, Susan Flynn-Hummel, and Sava Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In **ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998**, Concurrency: Practice and Experience, 1998. To appear.
- [GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In **Proceedings of the 19th ACM Symposium on Operating System Principles**, pages 29–43, 2003.
- [GNA<sup>+</sup>97] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. June 1997.
- [GNA<sup>+</sup>98] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Harding, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. October 1998.
- [Gor98] Rob Gordon. **Essential JNI: Java Native Interface**. Prentice Hall, 1998.
- [had] Apache hadoop. <http://hadoop.apache.org/core/>.
- [hdf] HDFS architecture. [http://hadoop.apache.org/core/docs/current/hdfs\\_design.html](http://hadoop.apache.org/core/docs/current/hdfs_design.html).
- [jav96] Hpcc and java - a report by the parallel compiler runtime consortium. <http://www.npac.syr.edu/users/gcf/hpjava3.html>, May 1996.
- [JRC08] Tina Miriam John, Anuradharthi Thiruvengkata Ramani, and John A. Chandy. Active storage using object-based devices. In **IEEE International Conference on Cluster Computing 2008**, September-october 2008.



- [Ken] W. James Kent. Blat source code. <http://www.soe.ucsc.edu/~kent/src/>.
- [Ken02] W. James Kent. Blat the blast-like alignment tool. In **Genome Research 2002**, volume 12, pages 656–664, March 2002.
- [Knu79] Donald E Knuth. The art of computer programming. 3, 1979.
- [mpia] mpijava homepage. <http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html>.
- [MPIb] MPICH—a portable implementation of MPI. <http://www.mcs.anl.gov/mpi/mpich/>.
- [mpi95] Message passing interface forum. mpi: A message passing interface standard. <http://www-unix.mcs.anl.gov/mpi/>, June 1995.
- [mpi97] Message passing interface forum. mpi-2: Extensions to the message passing interface. <http://www.mpi-forum.org>, July 1997.
- [MSW<sup>+</sup>05] X. Martorell, N. Smeds, R. Walkup, J. R. Brunheroto, G. Almsi, J. A. Gunnels, L. De Rose, J. Labarta, F. Escal, J. Gimenez, H. Servat, and J. E. Moreira. Blue gene/l performance tools. In **IBM Journal of Research and Development**, volume 49(2–3), pages 407–424, March – May 2005.
- [RGF98] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. August 1998.
- [Rie99] Erik Riedel. **Active Disks - Remote Execution for Network-Attached Storage**. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 1999. Tech. Report no. CMU-CS-99-177.
- [RRP<sup>+</sup>07] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In **Proceedings, International Symposium on High-Performance Computer Architecture (HPCA)**, pages 13–24, 2007.
- [vN63] John von Neumann. Collected works. 6, 1963.
- [wor] Word count tutorial. [http://hadoop.apache.org/core/docs/current/mapred\\_tutorial.html](http://hadoop.apache.org/core/docs/current/mapred_tutorial.html).
- [YC98] Narendar Yalamanchilli and William Cohen. Communication performance of Java based Parallel Virtual Machines. In **ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998**, Concurrency: Practice and Experience, 1998. To appear.

- [ZKJ<sup>+</sup>08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In **Proceedings of the Tenth Symposium on Operating System Design and Implementation**, 2008.