5-15-2009

An Efficient Platform for Large-Scale MapReduce Processing

Liqiang Wang
University of New Orleans

MMR: An Efficient Platform for Large-Scale MapReduce Processing

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

by

Liqiang Wang

B.S. Huazhong Normal University, 2004

May, 2009

Acknowledgments

I would like to express my deep gratitude to all the people who have helped and supported my master study during these years.

I give sincere thanks to my adviser, Dr. Vassil Roussev, for his kind guidance and supervision of my study. I have learned a lot from Dr. Roussev, from knowledge and skills for academic research to attitude and strategies for being a wise person. It is always my pleasure to work with him.

I am grateful to all the professors in my thesis committee. Besides my adviser, I want to thank Drs. Golden Richard III and Shengru Tu for their precious suggestions about my master research and advices about my life here.

I especially thank Dr. Mahdi Abdelguerfi who cared much about my life and family during the whole period of my study, and Ms. Jeanne Boudreaux, our department secretary, for her hard work and kind help.

I appreciate the understanding and encouragement of my wife Zhiyu Zhao and my parents. Without their care throughout these years it would be impossible for me to complete this master study.

Contents

Abstract **vi**

1 Introduction **1**

2 Related Work **5**

2.1 Pthreads 5

2.2 Message Passing Interface 6

2.3 MapReduce 6

2.4 Implementations of the MapReduce Model 8

2.4.1 Phoenix 8

2.4.2 Hadoop 8

3 Principal of the MMR Model **10**

3.1 MMR: A MPI Based MapReduce Model 10

3.2 Principal of the MMR Model 11

4 Performance Test and Evaluation **15**

4.1 Test Examples 15

4.1.1 Grep 16

4.1.2 WordCount 17

4.1.3 Pi Estimator 18

4.1.4 Bloom Filter 19

4.2 Test Environment 23
4.3 Benchmark Execution Times 24
4.3.1 Scalability 24
4.3.2 Super-linear and sub-linear speedup 25
4.3.3 Large file processing 28
4.4 Machine Learning Applications of MMR 29
4.4.1 Simple linear regression 30
4.4.2 K-means clustering 31
4.4.3 Performance evaluation 32

5 A MMR Live DVD **34**
5.1 Selection of Software for the MMR Cluster 34
5.2 Building the MMR Live DVD 35
5.3 Booting MMR Compute Nodes 37
5.4 A MMR CUDA Live Cluster 39

6 Conclusions **41**

Bibliography **44**

Appendix **58**

Vita **63**

Abstract

In this thesis we proposed and implemented the MMR, a new and open-source MapReduce model with MPI for parallel and distributed programing. MMR combines Pthreads, MPI and the Google's MapReduce processing model to support multi-threaded as well as distributed parallelism. Experiments show that our model significantly outperforms the leading open-source solution, Hadoop. It demonstrates linear scaling for CPU-intensive processing and even super-linear scaling for indexing-related workloads. In addition, we designed a MMR live DVD which facilitates the automatic installation and configuration of a Linux cluster with integrated MMR library which enables the development and execution of MMR applications.

Keywords MapReduce, Pthreads, MPI, Hadoop and MMR(MPI Based MapReduce).

1

Introduction

Hard disk capacity is growing surprisingly fast. Nowadays 1.5 TB is on the shelf with the price dropping below $200, and 2TB is coming soon. Computing and communication technologies are putting vast amount of disk storage and let user's hard disk content become complicated, bringing programmers headache. With capacity growth outpacing both bandwidth and latency improvements [13], user targets are not only getting bigger in absolute terms, but they are also growing larger relative to our capabilities to process them on time. It is difficult to overstate the urgent need to develop scalable computational solutions that can match the explosive growth in target size with adequate computational resources.

According to the Moore's Law [1], number of transistors and resistors on a chip double every 18 months. However, the race of CPU clock frequency among CPU manufacturers has hit a wall: between 2003 and 2005 clock frequency only increased from 3 to 3.8 GHz. Even though architectures were optimized for high clock speeds, the chip makers have simply come up against the laws of physics. Now Intel is using $2.3 \times 10^{-8}m$ wafers to create their CPU products, but the miniaturization has limitations because atom diameter is $0.5 \backsim 5 \times 10^{-10}m$. Using multi-core CPUs instead of increasing frequency should open a whole new methodology for logical processing. The speed and more importantly the thermal limitations should be drastically reduced. Because of the limitation of heat dissipation and die sizes, processors with more than one core have been produced by CPU manufacturers. For example, Intel Core i7 is Intel's newest processor which has 4 cores with simultaneous multi-threading

(SMT), a technology that had already appeared with the Pentium 4 "Northwood" under the name 'Hyper-Threading' and now comes back as 'SMT'.

The proximity of multiple CPU cores on the same die allows the cache cohere circuitry to operate at a much higher clock rate than is possible if the signals have to travel off-chip. Combining equivalent CPUs on a single die significantly improves the performance of cache snoop. However, the ability of using multi-core processors to increase application performance depends on the use of multiple threads within applications. According to the Amdahl's Law [9], $Speedup_{MAX} = \frac{1}{(1-P)+\frac{P}{S}}$, where P is the proportion of an application that can be improved by multi-threading, and S is the speedup of that improvement. The utilization of multiple threads does not automatically improve the performance of applications. Besides the multi-core processors we need multiple computers working together to process those high performance computing tasks on time.

There are two basic models for sharing data performance tasks: multi-threading and message passing. In a shared memory system,tasks share a global address space, which they read and write asynchronously. Synchronization mechanisms [15] such as locks, semaphores, etc must be used to control access to the shared memory. It is difficult to optimize performance through data locality. Multi-threading is a good model which also shares global address; however, it runs only on a single machine and programmers need to define the control flow of programs. Message passing models such as MPI (Message Passing Interface) can work on multiple machines. Tasks exchange data via messages, while this requires cooperation between functions such as *send()* and *receive()*.

Our goal is to find or develop a parallel model which makes it convenient and efficient to develop parallel applications. A good model in our mind is one that efficiently runs tasks and shares data in parallel on multi-machines and each machine processes multi-threads. It is a general model with which programmers usually do not need to think about complex parallelism issues such as synchronization, memory address, program control flow and

cooperation.

MapReduce [8], a new distributed programming paradigm developed by *Google*, is aimed at simplifying the development of scalable, massively-parallel applications that process terabytes of data on large commodity clusters. The goal is to make the development of such applications easy for programmers with no prior experience in distributed computing. Programs written in this style are automatically executed in parallel on as large a cluster as necessary. All the I/O operations, distribution, replication, synchronization, remote communication, scheduling, and fault-tolerance are done without any further input from the programmer who is free to focus on application logic. (We defer the technical discussion of the MapReduce model to the next section.) Arguably, essential ideas behind MapReduce are recognizable in much earlier work on functional programming, however, the idea of using *map* and *reduce* functions as the sole means of specifying parallel computation, as well as the robust implementation on a very large scale are certainly novel. After the early success of the platform, *Google* moved all of their search-related functions to the MapReduce platform.

As we have mentioned, our goal is to make it easy for developers to write their parallel applications. To achieve this we decided to develop an efficient and convenient parallel model which combines the Google's MapReduce model and the MPI interface of message passing to take full advantage of the performance gain brought by both multi-threading and multi-machines technologies. Our MMR(MPI based MapReduce) model adopts MapReduce's task scheduling strategies and enhances the message passing model by introducing multi-threading which is not originally integrated in the model. On the other hand, since creating a parallel multi-machines environment is not an easy task for everyone, we also considered the automatic establishment of a cluster environment which, although simple, is sufficient for users to run their MMR based application codes. A user turns on a group of machines which are connected via network, after the OS has been running, the cluster environment is automatically configured and the user can easily run his code on it. We have

developed a MMR model live DVD. After booting the operating system from the live DVD, a MMR cluster environment starts up automatically.

In this thesis, we introduce our MMR model which is based on the Google's MapReduce model. We provide several examples to simulate some digital forensic and machine learning tools and use serial, Hadoop, and MMR APIs to implement three versions of the examples and compare the performance of of different versions. The thesis is organized as follows: Chapter 2 first gives background information about the Google's MapReduce model and two implementations based on the model, Hadoop and Phoenix. Chapter 3 proposes our MMR model. It focuses on the conceptual framework of MMR as well as details about its control flow and data flow. Chapter 4 introduces a test environment that we have built to do our MMR experiments with a few test examples, and compares the performance of our MMR version of example codes with the Hadoop version of them. It also shows a few experiments that we have performed for the purpose of testing the machine learning applications of MMR. Chapter 5 describes our design of a MMR live DVD which facilitates the installation of a cluster environment with our MMR APIs integrated. Chapter 6 concludes the completed work and looks ahead the future work, and Appendix is a programming example and an API specification of our MMR implementation.

2

Related Work

Our goal is to find or develop a parallel model which makes it convenient and efficient to develop parallel applications. When an application fits the model very well, we expect to achieve linear to super-linear performance improvement with this model. A good model in our view is one that automatically runs tasks and data in parallel on multiple machines and each machine processes multi-threads. It is a general model with which programmers usually do not need to think about complex parallelism issues such as synchronization, memory address, program control flow and cooperation.

2.1 Pthreads

Pthreads [5] is an implementation of multi-threads model for shared memory multi-processor architectures. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are referred to as POSIX threads, or Pthreads. The primary motivation for using Pthreads is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

2.2 Message Passing Interface

MPI is a library specification for message-passing which is across network, proposed as a standard by a broadly based committee of vendors, implementors, and users. MPI was designed for high performance on both massively parallel machines and on workstation clusters. MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries. There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard. Implementations should be able to exploit native hardware features to optimize performance. A variety of implementations are available, both in vendor and public domains.

2.3 MapReduce

MapReduce [8], a new distributed programming paradigm developed by *Google*, is aimed at simplifying the development of scalable, massively-parallel applications that process terabytes of data on large commodity clusters. The MapReduce computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The developer expresses the computation as two programmer-provided functions–*map* and *reduce*. The former takes an input pair and produces a set of intermediate key/value pairs. The run-time engine automatically groups together all intermediate values associated with the same intermediate key I and passes them on to the *reduce*. The *reduce* function accepts an intermediate key I and a set of values for that key, and merges together these values (however it deems necessary) to form another (possibly smaller) set of values. Typically, just zero or one output value is produced per *reduce* invocation. The I values are supplied to the *reduce* function via an *iterator*, which allows for arbitrarily large lists of values to be passed on.

As an illustration, consider the `wordcount` example–given a set of text documents, count

the number of occurrence for each word. In this case, the *map* function simply uses the word as a key to construct pairs of the form (*word*, 1). Thus, if there are n distinct words in the document, the run-time will form n distinct lists, and will feed them to n different instances of the *reduce* function. The *reduce* function needs to simply count the number of elements in its argument list and output that as the result. Below is a pseudo-code solution from [8]:

```
map (String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Without getting into a detailed discussion, we should point out the natural sources of data parallelism in this solution. For the *map* function, we can create as many independent instances (each of which executable in parallel) as we have documents. If that is not enough, we could also split up the documents themselves. The *reduce* computation is also readily parallelization–we can have as many distinct instances as we have words. In general, we would expect that different instances of the functions to take different amounts of time so it is the runtime's job to keep track of partial results and to load balance the cluster. Note that the code written by the programmer is completely free of explicit concerns with regard to the size of the inputs and outputs, and distribution of the computation. Consequently,

the system can transparently spread the computation across all available resources. Further, it can schedule redundant task instances to improve fault-tolerance and reduce bottlenecks.

2.4 Implementations of the MapReduce Model

Although Google has published the MapReduce model, it did not provide its source code, so we cannot use this technology directly. However, there have been over ten different implementations based on this model, among which we are particularly interested in Phoenix and Hadoop.

2.4.1 Phoenix

Phoenix [14] is an open-source research prototype, which demonstrates the viability of the MapReduce model for shared memory multi-processor/ multi-core systems. It has demonstrated close to linear speedup for workloads that we believe are very relevant to many applications that fit the MapReduce model. *Phoenix* is implemented on the basis of Pthreads. As we have mentioned, Pthreads is an efficient way for multi-threading; this is the reason why we select *Phoenix* as our MapReduce implementation. However, *Phoenix* is written with CC and Solaris APIs, while we prefer to make our model runnable on Linux clusters which are used extensively. Therefore we have modified the *Phoenix* code to let it work on Linux. Besides this, we have fixed some bugs to make the code more robust.

2.4.2 Hadoop

Hadoop [2] was developed as an open-source *Java* implementation of the MapReduce programming model and has been adopted by large companies like *Yahoo!*, *Amazon*, and *IBM*. The National Science Foundation has partnered with Google and IBM to create the Cluster Exploratory (CluE), which provides a cluster of 1,600 processors to enable scientists to easily

create new types of scientific applications using the *Hadoop* platform. In other words, there is plenty of evidence that the MapReduce model, while not the answer to all problems of distributed processing, fits quite well with the types of tasks required in large scale applications such as forensic string operations, image processing, statistical analysis, etc.

An obvious question is: Is *Hadoop* an appropriate platform to implement scalable computation and data intensive applications? One potential concern is that a Java-based implementation is inherently not as efficient as Google's C-based one. Another one is that the *Hadoop File System* (HDFS), which is implemented as an abstraction layer on top of the regular file system, is not nearly as efficient as the original *Google File System* [10]. From a cost perspective, it may make perfect sense for a large company to pay performance penalty if that would lead to a simplification of its code base and make it easier to maintain. However, an average company or research lab has very modest compute resources relative to the large data centers available to most companies adopting *Hadoop*.

3

Principal of the MMR Model

3.1 MMR: A MPI Based MapReduce Model

Based on the published results from prior work and our own experience, we can confidently conclude that MapReduce is a very suitable conceptual model for describing many parallel and distributed applications. From a practitioner's perspective, the next relevant question is: Is the performance penalty too big to be practical for use in a realistic company or research lab? If yes, can we build a more suitable implementation? To answer these questions, we developed our own prototype version, called *MPI MapReduce*, or *MMR* for short, and evaluated its performance relative to *Hadoop*.

Our work builds on the *Phoenix* shared-memory implementation of MapReduce and the *MPI* 2 standard [1] for distributed communication. The essential idea is to start out with the shared-memory implementation and extend it with the ability to spread the computation to multiple nodes. MPI has been around for a while and has found wide use on scientific and other high-performance applications. MPI was designed with flexibility in mind and does not prescribe any particular model of distributed computation. While this is certainly an advantage in the general case, it is also a drawback as it requires fairly good understanding of distributed programming on part of the developer who must explicitly manage all communication and synchronization among distributed processes.

Our goal is to hide MPI behind the scenes and to create a middle-ware platform that

[1] http://www.mpi-forum.org/

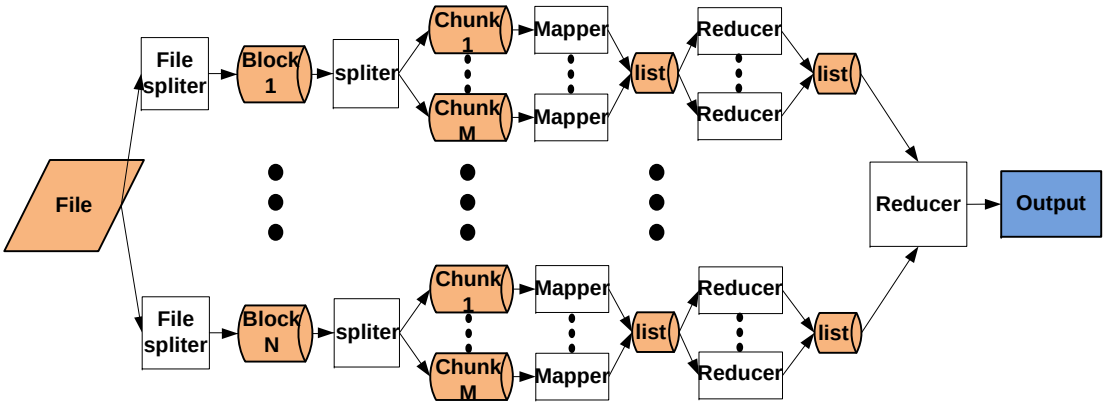

Figure 3.1: MMR data flow

allows programmers to focus entirely on expressing application logic and not worry about scaling up the computation. This becomes possible only because we assume the availability of the MapReduce model, which allows us to automatically manage communication and synchronization across tasks. We should also mention that by choosing MPI, we also simplify cluster management. For example, one could easily (within a day) create a dual-boot setup for a set of workstations in the lab that could be used as a cluster to run long processing jobs overnight. Modern MPI distributions have become quite user-friendly and should not present any major issues for administrators.

3.2Principal of the MMR Model

Figure 3.1 illustrates the overall data flow in a *MMR* application. At present, we use a single text or binary data file as the input. This is done only to simplify the experimental setup and to isolate file system influence on execution time. The system can just as easily use multiple files as input. A file splitter function splits the input into N equal blocks, where N is the number of machines (nodes) available. In our experimental setup the blocks are created of equal size as the machines we use are identical but the block-splitting can be quite different in a heterogeneous setting.

Each node reads its own block of data and splits it up further into M *chunks* according to the number of mapper threads to be created on each node. Normally, that number would be equal to the level of hardware-supported concurrency, i.e., the number of threads that the hardware can execute in parallel. The chunk size is also related to the amount of the on-board CPU cache–the system makes sure that there is enough room in the cache for all the mappers to load their chunks without interference. After the threads have been created, each thread gets a chunk of data, and the programmer-defined *map* function is called to manipulate that data and to produce key/value pairs. If the programmer has specified a *reduce* function, the results are grouped according to keys and, as soon as all the mapper threads complete their tasks, a number of reducer threads are created to complete the computation with each reducer thread invoking the programmer-defined *reduce* function. After the reduction, each node has a reduced key/value pair list. If all the lists should be merged, each node packs its result data into a data buffer and sends content in the buffer to the master node (node 0). When the master receives data from another node, it uses a similar *reduce* function to reduce the received key/value pairs with its own result. This procedure repeats until all the data from other nodes are reduced. Finally, the master outputs a complete key/value list as the final result.

Figure 3.2 illustrates the *MMR* flow of execution on each node and the basic steps a developer needs to perform. All the functions with the "mmr" prefix are *MMR* API functions supplied by the infrastructure. The first step in this process is to invoke the system-provided *mmrInit()* function, which performs a set of initialization steps, including MPI initialization. Next, the application invokes *mmrSetup()* to specify necessary arguments such as file name, unit size, and number of map/reduce threads on each node (optional), as well a list of application-defined functions such as key comparison, map and reduce functions.

Many of the mandatory arguments have default values to simplify routine processing. By default, *mmrSetup()* automatically opens and memory-maps the specified files, although this

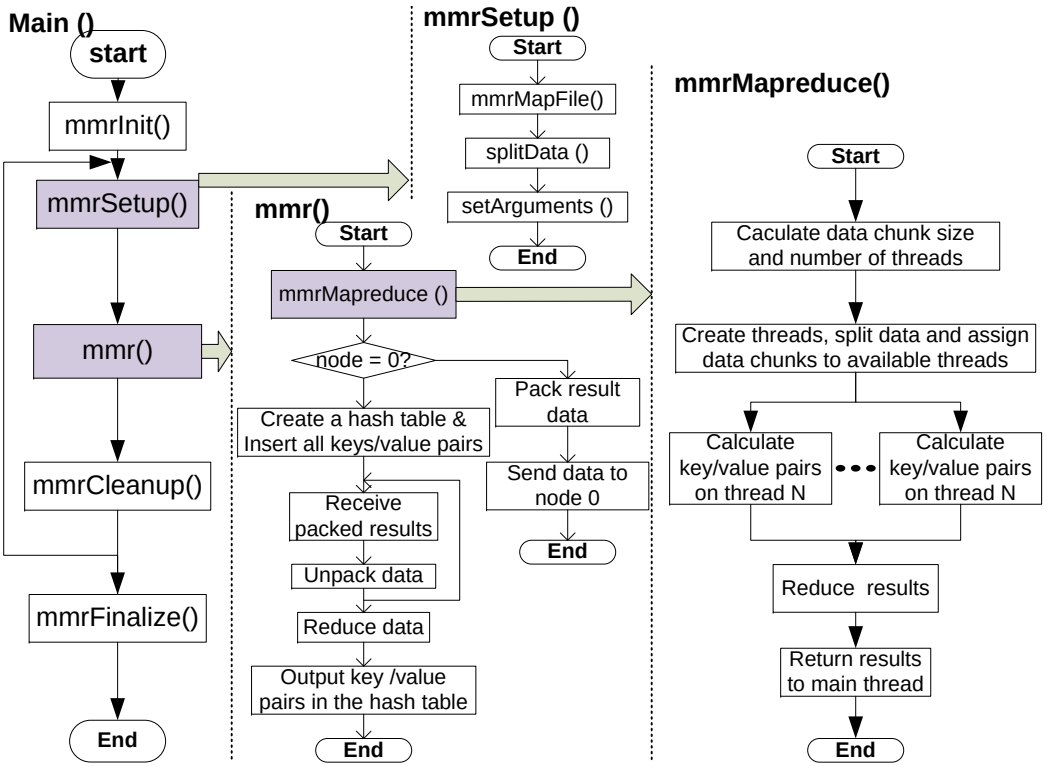

Figure 3.2: MMR Execution Flow

behavior can be overridden if the application needs access to the raw data (e.g., read header information). After the data is mapped into memory, the *splitData()* function calculates the offset and length of the data block on a node, while *setArguments()* sets all the necessary arguments for map, reduce and MPI communication.

After the above initialization steps, the application can call *mmr()* to launch the actual computation. The final result list is generated on the master node and returned to the application. More complicated processing may require multiple map/reduce rounds of computation. If that is necessary, the application invokes *mmrCleanup()* to reset buffers and state information. After that, the application can set up and execute another MapReducing (using *mmrSetup()* and *mmr()*).

The *mmr()* function first calls *mmrMapReduce()* to do map/reduce on a node. If a node is *not* the master, it must pack its result (a list of key/value pairs) into a MPI buffer and sends the content of that buffer to the master node. Since *MMR* does not know the data types of keys and values, the developer must define a function to pack/unpack the data. This is routine code and a number of default functions are provided for the most frequently used data types. If a node *is* the master, after *mmrMapReduce()* it generates a hash table to store hashed keys. The hash table is used to accelerate the later reduction of values on same keys. The master receives byte streams from each node, unpacks them into key-value pairs, aggregates them into a list, and returns them to the nodes for the reduction step. If an application does not need each node to send its partial result to the master, it can short circuit the computation by calling *mmrMapReduce()* rather than *mmr()*. The *mmrMapReduce()* function does map/reduce on each node and returns a partial list of key/value pairs. If there is no need to further reduce and merge the lists, then each node just uses its own partial list.

4

Performance Test and Evaluation

The primary purpose of this chapter is to compare the performance of the *MMR* and *Hadoop*. We also quantify the scalability of *MMR* for a set of applications by comparing it with baseline serial implementations. To provide a fair comparison, we used three *Hadoop* examples as provided by the system developers and wrote functional equivalents in *MMR* .

We did not make any changes to the *Hadoop* examples except to add a time stamp to calculate execution time. In addition, we developed a MMR Bloom filter program with the SHA-1 hashing algorithm for the purpose of testing the impact of network latency to the MMR's performance with massive data exchange between compute nodes.

4.1Test Examples

We have rewritten the following three Hadoop samples for *MMR* : `wordcount`, `pi-estimator`, and `grep` [11], besides which we have written a Bloom filter program. The `wordcount` program calculates the number of occurrence of each word in a text file, the `pi-estimator` program calculates an approximation of PI based on the Monte Carlo estimation method, the `grep` program searches for matched lines in a text file based on the regular expression matching, and the Bloom filter program calculates the false positive rate of a Bloom filter [6]based on the SHA-1 algorithm [4]. `wordcount` and `grep` are text processing programs, `pi-estimator`is computation-intensive with virtually no communication and synchronization overhead, whereas Bloom filter is both computation and communication intensive. The

function of *Hadoop* grep is weaker than the regular Linux command–when searching a specific word in a file, *Hadoop* grep just returns how many times this specific word appears in the file. However in Linux the grep command returns the line numbers and the whole lines. *MMR* grep can return the line numbers and the whole lines back like the Linux grep. However, to produce apples-to-apples comparison, we modified some *MMR* grep functions and made it return just the counts.

4.1.1 Grep

Grep is a command line text search utility originally written for Unix. The name is taken from the first letters in global / regular expression / print, a series of instructions for the text editor. The grep command searches files or standard input globally for lines matching a given regular expression, and prints them to the program's standard output.

```
Serial algorithm of grep:
Grep: Grep <filename> <pattern>
```

Open the file

Search for the pattern specified by the given regular expression

Save all the line numbers with the pattern found in those lines

Output text in the list of lines

Return

```
Parallel algorithm:
Grep: Grep <filename> <pattern>
```

Open the file

Calculate the number of lines in the file

Calculate the chunk size so that each node i is assigned n_i lines ($n_i = \frac{numberoflines}{numberofnodes}$)

Get a data chunk from the file buffer

Search the node's own data chunk for the pattern specified by the given regular expression

Save all the line numbers with the pattern found in those lines

If the node is node 0

Receive a list of line numbers from node i ($0 \leq i \leq numberofnodes - 1$)

Merge all the lists to a single list and sort the line numbers in it

Output text in the list of lines

Return

Else

Send a list of line numbers to node 0

Return

End If

4.1.2 WordCount

The `wordcount` program calculates the number of occurrence of each word in a text file and outputs a list of top 10 words.

```
Serial algorithm of the wordcount:
wordcount: wordcount <filename>
```

Open the file

Calculate the number of occurrence of each word in the file

Sort all the words in the file in a descending order of number of occurrence

Output a list of top 10 words with their numbers of occurrence

Return

`Parallel algorithm:`

`wordcount: wordcount <filename>`

Open the file

Calculate the file length i.e. the number of characters in the file

Calculate the chunk size so that each node i is assigned n_i characters ($n_i = \frac{numberofcharacters}{numberofnodes}$)

Adjust the position of a chunk so that no words are cut by neighbor chunks

Get a data chunk from the file buffer

Calculate the number of occurrence of each word in the chunk

If the node is node 0

Receive a word list with number of occurrence of each word from node i ($0 \leq i \leq numberofnodes - 1$)

Merge all the word lists and sum up the numbers of occurrence of the same words,

so that each word has a total number of occurrence

Sort all the words in the file in a descending order of number of occurrence

Output a list of top 10 words with their numbers of occurrence

Return

Else

Send a word list with number of occurrence of each word to node 0

Return

End If

4.1.3 Pi Estimator

The `pi-estimator` program calculates an approximation of PI based on the Monte Carlo estimation method.

`Serial algorithm of PiEstimator:`

Generate n (n = number of samples) random 2D points where each point is in a square with side length s

Calculate m, the number of points in a circle with radius $r = \frac{s}{2}$

Calculate the estimated $Pi = 4 \times \frac{m}{n}$

Output Pi

Return

`Parallel algorithm:`

Calculate n_i, the number of random points to be generated ($n_i = \frac{totalnumberofpoints}{numberofnodes}$)

Generate n_i random 2D points where each point is in a square of side length s

Calculate m_i, the number of points in a circle with radius $r = \frac{s}{2}$

If the node is node 0

Receive n_i and m_i from node i ($0 \leq i \leq numberofnodes - 1$)

Calculate $n_i = \sum_i n_i$ and $m_i = \sum_i m_i$

Calculate the estimated $Pi = 4 \times \frac{m}{n}$

Output Pi

Return

Else

Send n_i and m_i to node 0

Return

End If

4.1.4 Bloom Filter

The Bloom filter [3], conceived by Burton H. Bloom in 1970, is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives

are possible, but false negatives are not. Elements can be added to the set, but not removed (though this can be addressed with a counting filter). The more elements that are added to the set, the larger the probability of false positives.

`Serial algorithm of the Bloom filter false positive rate:`

Open the reference file
Use the SHA-1 algorithm to calculate hash keys for the file, so that each 4KB data has a 20-byte hash key
Save all the hash keys in a reference list
Create a buffer for the Bloom filter and set the buffer to all $0s$
For each hash key in the reference list
    Calculate its position in the Bloom filter
    Set the value of that position to 1
End For
Open the query file
Use the SHA-1 algorithm to calculate hash keys for the file, so that each 4KB data has a 20-byte hash key
Save all the hash keys in a query list
Save nq, the number of keys in the query list
Set nfp to 0, where nfp is the number of false-positives
For each hash key in the query list
    Calculate its position in the Bloom filter
    If the value of that position is 1
        $nfp = nfp + 1$
    End If
End For

Calculate $fpr = \frac{nfp}{nq}$, where fpr is the false-positive rate of the loom filter

Output fpr

Return

`Parallel algorithm:`

Open the reference file

Calculate the file length i.e. the number of bytes in the file

Calculate the chunk size so that each node i is assigned n_i bytes ($n_i = \frac{numberofbytes}{numberofnodes}$)

Get a data chunk from the file buffer

Use the SHA-1 algorithm to calculate hash keys for the chunk, so that each 4KB data has a 20-byte hash key

Save all the hash keys in a reference list

Create a buffer for the Bloom filter and set the buffer to all $0s$

For each hash key in the reference list

Calculate its position in the Bloom filter

Set the value of that position to 1

End For

If the node is node 0

Receive a Bloom filter from node i ($0 \leq i \leq numberofnodes - 1$)

Merge all the Bloom filters to a single filter, so that if any position in any Bloom filter is 1, then that position is set to 1 in the merged filter

Broadcast the merged Bloom filter to all the nodes

Else

Send the Bloom filter to node 0

End If

Open the query file

Calculate the file length i.e. the number of bytes in the file

Calculate the chunk size so that each node i is assigned n_i bytes ($n_i = \frac{numberofbytes}{numberofnodes}$)

Get a data chunk from the file buffer

Use the SHA-1 algorithm to calculate hash keys for the file, so that each 4KB data has a 20-byte hash key

Save all the hash keys in a query list

Save nq_i, the number of keys in the query list

Set nfp_i to 0, where nfp_i is the number of false-positives

For each hash key in the query list

Calculate its position in the Bloom filter received from node 0

If the value of that position is 1

$nfp_i = nfp_i + 1$

End If

End For

If the node is node 0

Receive nfp_i and nq_i from node i ($0 \leq i \leq numberofnodes - 1$)

Calculate $nfp = \sum_{i=1}^{nfp_i}$ and $nq = \sum_{i=1}^{nq_i}$

Calculate $fpr = \frac{nfp}{nq}$, where fpr is the false-positive rate of the Bloom filter

Output fpr

Return

Else

Send nfp_i and nq_i to node 0

Return

End If

Evidently, for I/O-bound programs, launching more map processes would not increase the performance of either *MMR* or *Hadoop*. However, for CPU-bound code, launching more map

	Cluster #1	Cluster #2
CPU	Intel Core 2 Duo E6400	Core 2 Extreme X6800
Clock (GHz)	2.13	3.0
Number of cores	2	4
CPU Cache (KB)	2048	2 x 4096 KB
RAM (MB)	2048	2048

Table 4.1: Hardware Configuration

processes than the number of processing units tends to improve the *MMR* performance; for *Hadoop*, that is not the case. We recommend that, when performing computation programs, setting the number of map processes to 3-4 times of the computate nodes; however, when performing file processing programs, setting the number of map processes to the computation nodes.

4.2 Test Environment

MMR has fairly modest hardware requirements: a cluster of networked Linux machines, a central user account management tool, GCC 4.2, ssh, gunmake, OpenMPI, and a file sharing system. Certainly, better machines provide more benefits but, as our experiments show, even more modest hardware provides tangible speedup. For our experiments we used two ad-hoc clusters made up of lab workstations. The first one, Cluster #1, consists of 3 Dell dual-core boxes, whereas Cluster #2 has 3 brand new Dell quad-core machines. Table 4.1 gives the configuration of machines in the clusters. (Note: the quad-core machines were run on a 2GB RAM configuration to make the results comparable–normally they have 4GB.) All machines were configured with Ubuntu Linux 8.04 kernel version 2.6.24-19, *Hadoop*, and *MMR* . The network setup included the standard built-in Giga-bit Ethernet NICs and a CISCO 3750 switch.

Note that the *Hadoop* installation uses the *Hadoop Distributed File System* (HDFS), whereas *MMR* just uses NFS. HDFS separates a file into many chunks and distributes them

into many nodes. When a file is requested, each node sends its chunks to the destination node. However the NFS file system uses just one file server; when a file is requested, the server has to send the whole file to the destination node. Therefore, in general, the HDFS performs better than NFS, which means the *MMR* implementation spends more time on opening a file than the comparable *Hadoop* tool. However, for the purposes of our tests, we virtually eliminated such effect by forcing the caching of the files before the timed execution.

4.3 Benchmark Execution Times

We tested `wordcount` and `grep` with 10MB, 100MB, 1,000MB and 2,000MB files, and `pi-estimator`with 12,000 to 1,200,000,000 points. All our testing results are averages over 10 runs at the optimal settings (determined empirically), first and last runs are ignored. All experiments are performed on Cluster #2. As shown by 4.1, for `wordcount`, *MMR* is about 15 times faster than *Hadoop* for large (1GB+) files and about 23 times for small ones. For `grep`, in the worst case MMR is about 63 times faster than *Hadoop*. For `pi-estimator`, the purely computational workload, *MMR* is just over 3 times faster for the large point set. Overall, it is evident that *Hadoop* has significantly higher startup costs so the times from the longer runs should be considered more representative.

4.3.1 Scalability

Let us now consider the efficiency with which the two implementations scale up and use available CPU resources. Figure 4.2 compares the execution times for each of the three applications under *MMR* and *Hadoop* on each of the two clusters. `pi-estimator`is a trivially parallelized application which we would expect to scale up proportionately to hardware improvements of the CPU. We do not expect caching or faster memory to make any difference. Since the two processors are successive designs with virtually the same architecture, it all boils down to number of cores and clock rates. Thus, given twice the number of cores and

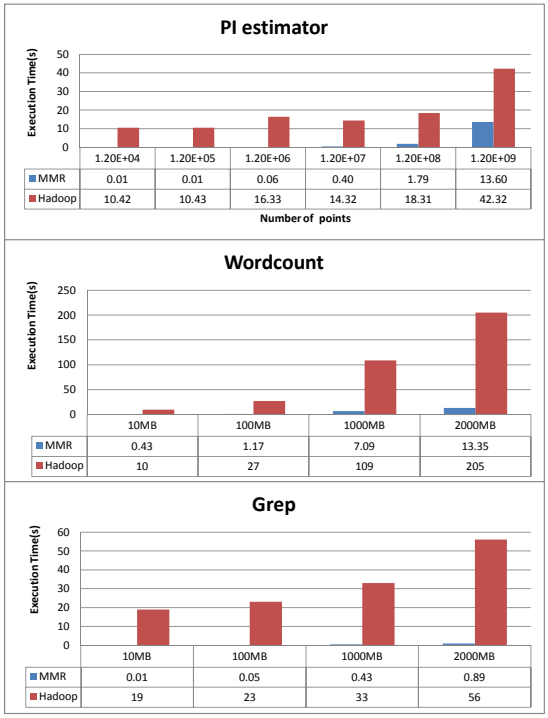

Figure 4.1: MMR vs Hadoop: execution times for three example applications

50% faster clock, one would expect speedup in the order of 3 times.

Indeed, the actual execution times do show the expected 3 times improvement. By contrast, the *Hadoop* version shows only 50% improvement, which we have no ready explanation for, especially given the 3 times improvement on the `wordcount` benchmark. In `wordcount`, our implementation achieves 4.2x speedup, which exceeds the pure CPU speed up of 3 times due to faster memory access and larger caches. For `grep`, which is a memory-bound application, the speedup is dominated by faster memory access, with minor contribution from the CPU speedup. We could not measure any speedup for the *Hadoop* version.

4.3.2 Super-linear and sub-linear speedup

So far we have seen that, for CPU-bound applications, *MMR* is able to efficiently take advantage of available CPU cycles and deliver speedup close to the raw hardware improve-

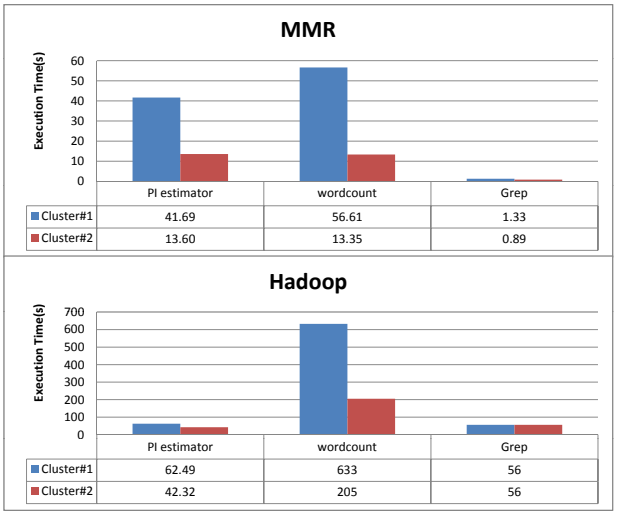

Figure 4.2: MMR vs Hadoop: benchmark times on Cluster #1 and Cluster#2

ments. In a realistic setting, we would expect a significant number of applications not to adhere to this model and we now consider the use of *MMR* in such scenarios. Specifically, we compare the speedup of two *MMR* applications–`wordcount` and `bloomfilter`–relative to their serial version, using Cluster #2. In other words, we compare the execution on a single core versus execution on 12 cores.

We consider two applications–`wordcount` (same as before) and `bloomfilter`. The `bloomfilter` application hashes a file in 4KB blocks using SHA-1 and inserts them into a Bloom filter [3]. Then, a query file of 1GB is hashed the same way and the filter is queried for matches. If such maches are found, the hashes triggering them are returned as the result. In the test case, the returned result would be about 16MB. To isolate and understand the effects of networking latency, we created two versions: *MMR* and *MMR Send*. The former completes the computation only and sends a total count back, whereas the latter sends the actual hash matches back to the master.

Figures 4.3 and 4.4 show the performance results. As before, we observe that `wordcount` scales in a super-linear fashion with 15.65x speedup. In fact, the relative speedup factor (speedup/concurrency) is about 1.3, which is very close to the 1.4 number obtained earlier

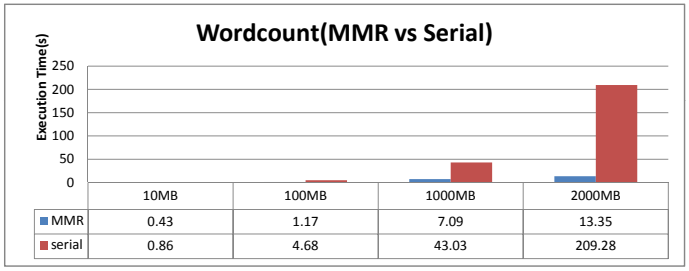

Figure 4.3: Wordcount: MMR vs serial execution

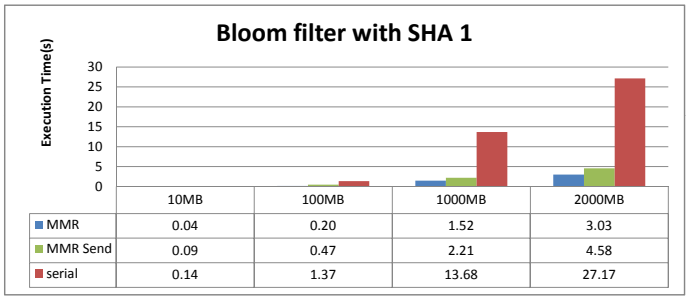

Figure 4.4: Bloom filter: MMR vs serial execution

across the two clusters. Again, this is likely due to beneficial caching effects and is great news because the core `wordcount` computation is closely related to the computations performed during indexing.

In the `bloomfilter` case, because the computation is relatively simple and the memory access pattern is random (no cache benefits), memory and I/O latency become the bottleneck factor to speedup. Looking at the 1 and 2GB cases, we see that the *MMR* version gets consistently a speedup factor of 9, whereas the *MMR Send* version achieves only 6. For longer computations, overlapping network communication with computation would help hide some of the latency but, overall, this type of workload cannot be expected to scale as well as the previous one.

	Compute node
CPU	Intel(R) Xeon(R) CPU E5345
Clock (GHz)	2.33
Number of CPU	2
Number of cores	4
CPU Cache (KB)	2 x 4096 KB
RAM (MB)	4096
Network Interface	10 GB/sec Infiniband
Network Interface	1 GB/sec Ethernet
Operating System	Red Hat Enterprise Linux 4 X64

Table 4.2: Hardware configuration of Poseidon

4.3.3 Large file processing

Due to the 32-bit file length limitation, neither Hadoop nor Phoenix is able to process files larger than 4 GB. We have modified the Phoenix code used in our MMR implementation for the purpose of large file processing. With this capability, users just need to complie their MMR based application codes with *-D_FILE_OFFSET_BITS=64*.

We have tested the large file processing performance of our Bloom filter on Poseidon, one of the clusters of the Louisiana Optical Network Initiative (LONI) [1]. Poseidon is a 4.77 tflops peak performance, 128 compute node cluster running the Red Hat Enterprise Linux 4 operating system. Each node contains two dual-core Xeon 64-bit processors operating at a core frequency of 2.33 GHz. Poseidon is a LONI's Dell Linux cluster housed at the University of New Orleans. Table 4.2 shows the hardware configuration of Poseidon.

Figure 4.5 shows the performance results of the Bloom filter with reference and query file sizes being 1 GB, 5 GB and 10 GB, respectively, and the number of compute nodes varies from 2 to 16. The experiment of 10 GB files with 2 nodes could not be completed because of an "out of memory" error, so we just put a 0.00 in the figure.

[1]Portions of this research were conducted with high performance computational resources provided by the Louisiana Optical Network Initiative (http://www.loni.org).

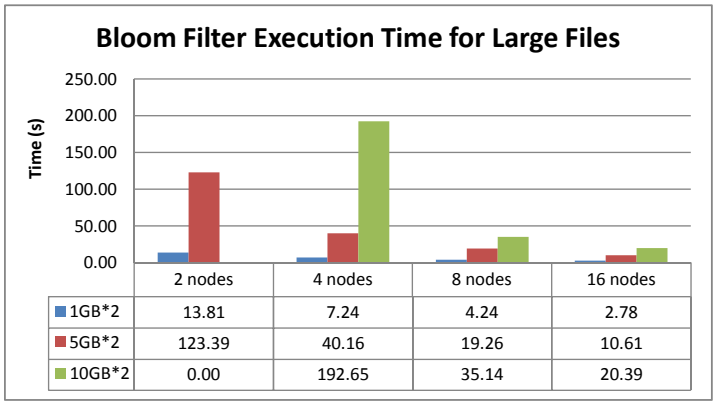

Figure 4.5: Bloom Filter Execution Time for Large Files

According to the figure, when the number of nodes and the file sizes match well, execution time for processing large files is proportional to that of smaller files. Examples in the figure include 1 GB, 5 GB and 10 GB with 8 and 16 nodes, respectively. For some cases where file sizes are large while number of nodes is relatively small, the execution time of a large file is not proportional to that of a smaller file. For example, the execution time of 5 GB/2 nodes is about 10 times of that of 1 GB/2 nodes. This is because a large data chunk on a node will have to be put in the swap partition instead of the memory of that node, and the former is obviously much slower than the latter.

4.4Machine Learning Applications of MMR

In this section we present some additional examples for further evaluating the performance of machine learning applications of MMR.

There are two ways to implement machine learning with the MMR model. First is data parallelism which does the same operations on different data chunks. We can separate training data and/or testing data into small chunks, then each thread on each node runs the same operations. Another way is task parallelism which allows different threads and nodes to run different operations with different data. However, it requires high data independency

to the operations.

We use two machine learning methods, the simple linear regression [16] and the K-means clustering [12]to explain how to use the MMR model to improve the time performance of machine learning algorithms.

4.4.1 Simple linear regression

A simple linear regression is a linear regression in which there is only one covariate (predictor variable). Simple linear regression is a form of multiple regression.

Simple linear regression is used in situations to evaluate the linear relationship between two variables. One example could be the relationship between muscle strength and lean body mass. Another way to put it is that simple linear regression is used to develop an equation by which we can predict or estimate a dependent variable given an independent variable.

Given a sample (Y_i, X_i), $i = 1, \ldots, n$,

the regression model is given by

$Y_i = a + bX_i + \varepsilon_i$

Where Y_i is the dependent variable, a is the y intercept, b is the gradient or slope of the line, X_i is the independent variable and ε_i is a random term associated with each observation.

The minimization problem can be solved using calculus, producing the following formulas for the estimates of the regression parameters:

$\hat{b} = \frac{\sum_{i=1}^{N}(x_i y_i) - N\bar{x}\bar{y}}{\sum_{i=1}^{N}(x_i)^2 - N\bar{x}^2}$

$\hat{a} = \bar{y} - \hat{b}\bar{x}$

A parallel implementation is that some values can be calculated in parallel, such as

$SXX = \sum_{i=1}^{N} x_i x_i$

$SYY = \sum_{i=1}^{N} y_i y_i$

$SXY = \sum_{i=1}^{N} x_i y_i$

$SX = \sum_{i=1}^{N} x_i$

$SY = \sum_{i=1}^{N} y_i$

Each node sends its partial result, which is calculated in parallel, to the node 0. Node 0 sums up the result values and calculates

$\hat{b} = \frac{N \times SXY - SX \times SY}{N \times SXX - SX \times SX}$

$\bar{x} = \frac{SX}{N}$

$\bar{y} = \frac{SY}{N}$

$\hat{a} = \bar{y} - \hat{b}\bar{x}$

$\epsilon^2 = \frac{(N \times SXY - SX \times SY)(N \times SXY - SX \times SY)}{(N \times SXX - SX \times SX)(N \times SYY - SY \times SY)}$

Finally, node 0 reports the result.

4.4.2 K-means clustering

K-means is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. These centroids should be placed in a tricky way because different locations result in different clusters. So, a better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenter of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more. Finally, this algorithm aims at minimizing an objective function, in this case a squared error function.

	Cluster
CPU	Intel Core 2 Extreme QX6850
Clock (GHz)	3.0
Number of cores	4
CPU Cache (KB)	2 x 4096 KB
RAM (MB)	2048

Table 4.3: Hardware configuration for machine learning experiments

The algorithm is composed of the following steps:

1. Place K points into the space represented by the objects that are being clustered. These points represent initial group of centroids.

2. Assign each object to the group that has the closest centroid.

3. When all objects have been assigned, recalculate the positions of the K centroids.

4. Repeat steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.

Parallel implementation of K-means clustering is very simple. Node 0 sends K initial centroids to each node. After that, each thread clusters a small number of points based on the K centroids and sends its clustering result to the node 0. Node 0 recalculates the K centroids. Repeat these steps until the centroids do not move.

4.4.3 Performance evaluation

The primary purpose of this section is to compare the performance of the *MMR* and serial implementations. For the machine learning experiments we used an ad-hoc cluster made up of lab workstations. Table 4.3 gives the configuration of the machines in the cluster. All machines were configured with Ubuntu Linux 8.04 kernel version 2.6.24-19 and *MMR* . The network setup included standard built-in Gbit Ethernet NICs and a CISCO 3750 switch.

We tested the linear regression with a 1.5GB file and the K-means with 100,000 3D points clustered into 100 groups. All our testing results are averages over 10 runs at the optimal

	Linear regression(s)
serial	3.379729
MMR	0.382139

Table 4.4: Execution times of linear regression(serial vs MMR)

	k-means(s)
serial	6.959
MMR	0.712

Table 4.5: Execution times of K-means(serial vs MMR)

settings (determined empirically), first and last runs are ignored. Table 4.4 compares the execution times of serial and MMR versions of the linear regression program. Table 4.5 compares the execution times of serial and MMR versions of our K-means program. For the linear regression, the MMR version is about 8 times faster than the serial version. For the K-means, the MMR is about 9 times faster. Based on both tables, the performance gain of using the MMR is close to 12, the theoretical limit that can be achieved with a cluster of 3 nodes where each node is a quad-core machine.

5

A MMR Live DVD

Creating a parallel multi-machines environment is not an easy task for everybody. To facilitate the installation and configuration of a cluster, we have designed a MMR live DVD. We considered the automatic establishment of a cluster environment that is simple but sufficient for users to run their MMR based application codes. A user turns on a set of machines which are connected via network and boots the DVD, then after booting the OS from the DVD, a cluster environment is automatically configured and the user can easily run his code on it.

5.1 Selection of Software for the MMR Cluster

The GNU/Linux world supports various cluster software: for application clustering, there are Beowulf, distcc, and MPICH; for Linux Virtual Server, there are Linux-HA - director-based clusters that allow incoming requests for services to be distributed across multiple cluster nodes; MOSIX, openMosix, Kerrighed and OpenSSI are full-blown clusters integrated into the kernel that provide for automatic process migration among homogeneous nodes; OpenSSI, openMosix and Kerrighed are Single-System Image implementations. Microsoft Windows Compute Cluster Server 2003 based on the Windows Server platform provides pieces for high performance computing like the Job Scheduler, MSMPI library and management tools. Among them, Beowulf clusters are performance-scalable clusters based on commodity hardware, on a private system network, and with open source software (Linux) infrastructure. The designer can improve performance proportionally with added machines.

The commodity hardware can be any of a number of mass-market, stand-alone compute nodes as simple as two networked computers each running Linux and sharing a file system.

Beowulf [17]is a multi-computer architecture which can be used for parallel computations. It is a system which usually consists of one server node and one or more client nodes connected together via Ethernet or some other network. It is a system built using commodity hardware components, like any PC capable of running a Unix-like operating system, with standard Ethernet adapters, and switches. It does not contain any custom hardware components and is trivially reproducible. Beowulf also uses commodity software like the Linux or Solaris operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). The server node controls the whole cluster and serves files to the client nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf machines might have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases client nodes in a Beowulf system are dumb, the dumber the better. Nodes are configured and controlled by the server node, and do only what they are told to do. In a disk-less client configuration, client nodes do not even know their IP address or name until the server tells them what it is.

5.2Building the MMR Live DVD

Our MMR cluster is based on Beowulf cluster so the commodity hardware can be any of a number of mass-market, stand-alone compute nodes as simple as two networked computers each running Linux and sharing a file system. The MMR live DVD is based on the Ubuntu live CD, but we have removed X-windows and man-db (help documents) because we need the live DVD to be as small as possible. In addition, we have written a script file which is executed after the Ubuntu live CD has booted, and the script automatically runs DHCP service which provides IP addresses, TFTP service which provides Linux kernel for PXE clients, NFS service which provides system files and MMR libraries, and NIS (Network

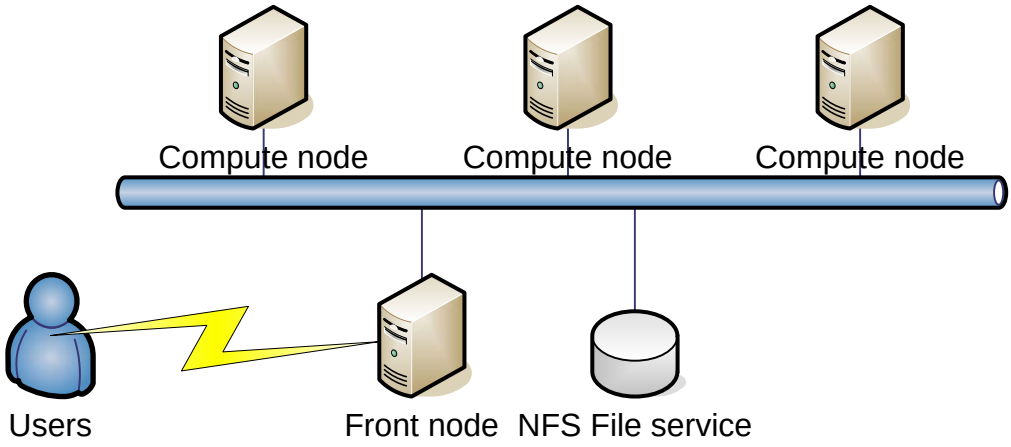

Figure 5.1: The typical configuration of a MMR cluster

Information Service) which provides central user management. Because the system files and MMR libraries (include GCC, open-MPI, CUDA, and MMR) for compute nodes need more than 3 GB disk space, we use a DVD to save all of them. All the MMR library files are located in the /usr/lib/mmr directory and system environment variables such as PATH and LIB are saved in the /etc directory with file name *environment*. The users need to set the compute nodes' BIOS to boot from network PXE. The IP address range for the cluster is from 192.168.10.1 to 192.168.10.254 and 192.168.10.1 is the front node (the machine running the live DVD). We have also compiled a new kernel for the compute nodes to load system files from the NFS server. The key point for the change is to redirect the kernel booting path to the NFS server whose IP is 192.168.10.1. The front node also supports NAT (Network address translation) service, therefore users can conveniently login the compute nodes from the Internet. Figure 5.1 illustrates the typical configuration of a MMR cluster.

The MMR compute nodes are hard-disk-free machines which boot their OS from the front node using the PXE protocol [7]. PXE is defined on a foundation of industry-standard Internet protocols and services that are widely deployed in the industry, namely TCP/IP, DHCP, and TFTP. These standardize the form of the interactions between clients and servers. To

ensure that the meaning of the client-server interaction is standardized as well, certain vendor option fields in DHCP protocol are used, which are allowed by the DHCP standard. The operations of standard DHCP and/or BOOTP servers (that serve up IP addresses and/or NBPs) will not be disrupted by the use of the extended protocol. Clients and servers that are aware of these extensions will recognize and use this information, and those that do not recognize the extensions will ignore them.

In brief, the PXE protocol operates as follows. The client initiates the protocol by broadcasting a DHCPDISCOVER containing an extension that identifies the request as coming from a client that implements the PXE protocol. Assuming that a DHCP server or a Proxy DHCP server implementing this extended protocol is available, after several intermediate steps, the server sends the client a list of appropriate boot servers. The client then discovers a boot server of the type selected and receives the name of an executable file on the chosen boot server. The client uses TFTP to download the executable from the boot server. Finally, the client initiates execution of the downloaded image. At this point, the client's state must meet certain requirements that provide a predictable execution environment for the image. Important aspects of this environment include the availability of certain areas of the client's main memory, and the availability of basic network I/O services.

5.3 Booting MMR Compute Nodes

Figure 5.2 illustrates the steps of booting a MMR compute node. All nodes have the same booting procedure, and below a step-by-step synopsis for the booting is given. Although the whole procedure is complex, users do not need to take much care because the setups are completed automatically as we have modified and recompiled the Linux kernel on the live DVD to run them after the front node has booted from the DVD.

Step 1. The front node boots itself from the MMR live DVD.

Step 2. A client(compute node) broadcasts a DHCPDISCOVER message to the standard

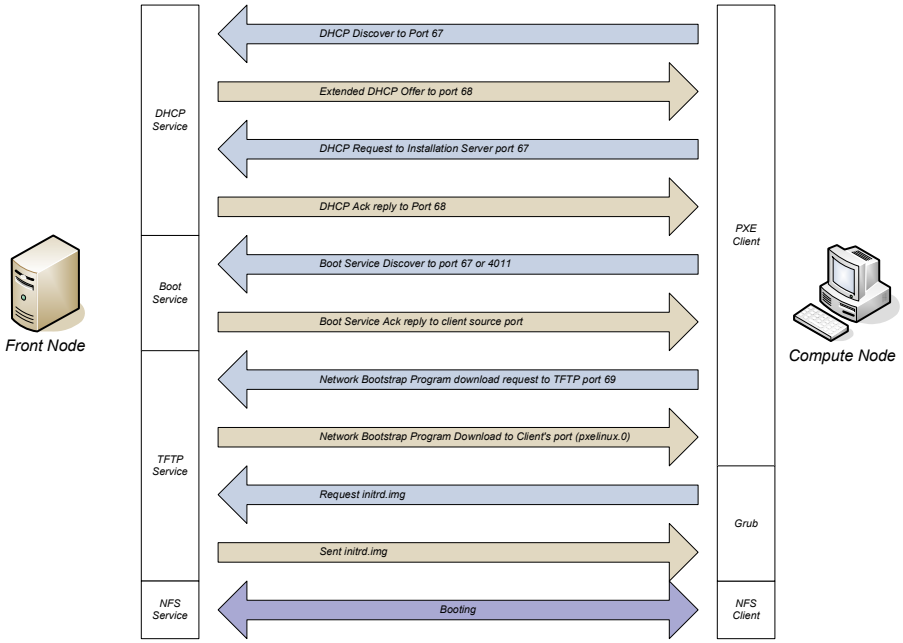

Figure 5.2: MMR live DVD

DHCP port (67).

Step 3. The DHCP or Proxy DHCP server responds by sending a DHCPOFFER message to the client on the standard DHCP reply port (68).

Step 4. From the DHCPOFFER(s) that it receives, the client records the following:

The Clients IP address offered by a standard DHCP.

The boot server list from the boot server field in the PXE tags from the DHCPOFFER.

The Discovery Control Options (if provided).

The Multicast Discovery IP address (if provided).

Step 5. If the client selects an IP address offered by a DHCP server, then it must complete the standard DHCP protocol by sending a request for the address back to the server and then waiting for an acknowledgment from the server. If the client selects an IP address from a BOOTP reply, it can simply use the address.

Step 6. The client selects and discovers a boot server and sends a packet to the DHCP server. This packet may be sent broadcast (port 67), multicast (port 4011), or unicast

(port 4011) depending on discovery control options included in the previous DHCPOFFER containing the PXE service extension tags.

Step 7. The boot server unicasts a DHCPACK packet back to the client on the client source port. This reply packet contains the boot file name *pxelinux.o*.

Step 8. The client downloads the executable file *pxelinux.o* using standard TFTP (port69).

Step 9. The PXE client determines whether an authenticity test on the downloaded file is required. If the test is required, the client sends another DHCPREQUEST message to the boot server requesting a credentials file for the previously downloaded boot file, downloads the credentials file via TFTP, and performs the authenticity test.

Step 10. If the authenticity test is successful, then the PXE client initiates execution of the downloaded pxelinux.o file.

Step 11. After the PXE client has executed a the pxelinux.o file, the client sends another REQUEST message to the TFTP server to request an image file (*initrd.img*) of the Linux kernel.

Step 12. Finally, If the initrd.img is downloaded successfully, the client sends another REQUEST message to the NFS boot server to request system files.

5.4A MMR CUDA Live Cluster

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA. Simply put, CUDA is the compute engine in NVIDIA graphics processing units or GPUs, that is accessible to software developers through industry standard programming languages. Programmers use C for CUDA, compiled through a PathScale Open64 C compiler, to code algorithms for execution on the GPU. CUDA is architected to support all computational interfaces, including C and new open standards like OpenGL and DX11 Compute.

Our MMR cluster also includes a CUDA library, so it also supports a CUDA cluster with the combination of MPI and CUDA. The easiest way to combine MPI and CUDA is to use nvcc to compile MPI-enabled CUDA codes with a nvcc compiler wrapper. The nvcc compiler wrapper is somewhat more complex than the typical mpicc compiler wrapper, however, for the users it is easier to make MPI code into CUDA source files and compile with nvcc than the other way around. Blow is what a makefile for a MPI-enabled CUDA code would look like:

```
MPICC      := nvcc -Xptxas -v
MPI_INCLUDES    := /usr/include/mpi
MPI_LIBS   := /usr/lib
%.o : %.cu
  $(MPICC) -I$(MPI_INCLUDES) -o $@ -c $<

program_name : program_name.o
  $(MPICC) -L$(MPI_LIBS) -o $@ *.o

clean :
  rm program_name.o

all : program_name
```

6

Conclusions

Scalable computational developers are in critical need of scalable development platforms that can automatically take advantage of distributed compute resources. The MapReduce model has recently gained popularity and the backing of major industry players as a viable option to quickly and efficiently build scalable distributed applications and has been adopted as a core technology by the top two web search companies-Google and Yahoo!.

In this thesis, we introduced a new open-source implementation–MPI MapReduce (MMR)–which significantly outperforms the leading open-source solution Hadoop. We showed that, unlike Hadoop, we can efficiently and predictably scale up a MapReduce computation. Specifically, for CPU-bound processing, our platform provides linear scaling with respect to the number and speed of CPUs provided. For the wordcount application, which is closely related to common indexing tasks, we demonstrated super-linear speedup, whereas for I/O-bound and memory-constrained tasks the speedup is substantial but sub-linear.

Based on this initial evaluation, and our development experience, we can confidently conclude that the proposed MMR platform provides a promising basis for the development of large-scale computational processing tools. Our next short term goals are to scale up the experiments to tens/hundreds of cores, and develop specific tools that can deal with actual terabyte-size targets in real-time using a GPU MMR MapReduce model.

In addition to the MMR programming model, we have also designed a MMR live DVD which facilitates the automatic installation and configuration of a Linux cluster with inte-

grated MMR libraries. The cluster is simple but the software environment that it provides is sufficient for developing and running MMR applications, and there is no need to manually install the software on each node. Moreover, users can easily build such a cluster with various types of machines, from a desktop PC to a Mac laptop, with Intel or compatible processors. This provides a convenient and economic MMR cluster solution to companies and institutions who would like to build such a cluster with no extra hardware costs.

Bibliography

[1] http://en.wikipedia.org/wiki/Moore%27s_law.

[2] http://hadoop.apache.org/core/.

[3] M. Mitzenmatcher A. Broder. Network applications of bloom filters: a survey. In *In Annual Allerton Conference on Communication, Control, and Computing, Urbana-Champaign, Illinois, USA*, 2002.

[4] P. van Oorschot A. Menezes and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc, 1997.

[5] Maarten van Steen Andrew S. Tanenbaum. *Distributed Systems Principles and Paradigms*. Pearson Education, 2002.

[6] R. Rubinfeld A. Tal. B. Chazelle, J. Kilian. The bloomier filter an efficient data structure for static support lookup tables. In *In Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 30–39, 2004.

[7] Intel Corporation. *Preboot Execution Environment Specification*. SystemSoft, 1999.

[8] Ghemawat S. Dean, J. Mapreduce: Simplified data processing on large clusters. In *n Proceedings of the Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, 2004.

[9] Legnago (VR) G. Argentini, R. Group. A generalization of amdahl's law and relative conditions of parallelism. In *Distributed, Parallel, and Cluster Computing*, 2002.

[10] Gobioff H. Ghemawat, S. and S. Leung. The google file system. In *In 19th ACM Symposium on Operating Systems Principles, Proceedings, pages 2943. ACM Press*, 2003.

[11] Joseph A. Goguen and Jose Meseguer. *Unifying functional, object-oriented and relational programming with logical semantics*. MIT Press, 1987.

[12] Roger K. Blashfield Mark S. Aldenderfer. *Cluster Analysis*. Sage, 1984.

[13] D. Patterson. Latency lags bandwidth. In *I Communications of the ACM*, volume 47, 2004.

[14] Raghuraman R. Penmetsa A. Bradski G. Ranger, C. and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture, Phoenix, AZ,*, 2007.

[15] William Stallings. *Operating Systems Internals and Design Principles*. Pearson Prentice Hall, 2004.

[16] Chris Tofallis. Investment volatility: A critique of standard beta estimation and a simple way forward. In *European Journal of Operational Research*, volume 187, pages 1358–1367, 2008.

[17] Ewing Lusk William Gropp and Thomas Sterling. *Beowulf Cluster Computing with Linux*. Jon maddog Hall, 2003.

Appendix

A C Programming Example

```
//Include necessary header files here.
/*********************************************************/
#include "mmr.h" //A header file for the MMR functions.
#include "common_macros.h" //A header file for some common macros.
/*********************************************************/
//Following functions must be implemented by the user.
/*********************************************************/
//Key comparison function.
int keyComp(const void *key1, const void *key2)
{
        //Return 0 only when two keys are identical.
        /*Following is a word count example which compares two keys
           as two words.
  return strcmp((const char *)key1, (const char *)key2);*/
}
/*********************************************************/
//Map function.
void map(map_args_t *args)
{
        assert(args);
```

```
        assert(args->length>0);
        char *data = (char *)args->data;
        assert(data);
        //Find or calculate keys and values from data and emit a key
           /value pair for each key.
        //Call emit_intermediate() (see MapReduceScheduler.h). It
           can be called multiple times depending on the number of
           key/value pairs.
        /*Following is a word count example which emits a word as a
           key and 1 i.e. the count as a value.
        emit_intermediate(wordPos, (void *)1, wordLen);*/
}
/*************************************************************/
//Following functions are optional.
/*************************************************************/
//Specify a mmr splitter function if the default mmr splitter is
   insufficient. By default data is splitted by user defined unit
   size.
//If splitter() is defined, mmrSplitter() must be defined in a
   similar way.
int mmrSplitter(mmr_data_t* mmrData){
        assert(mmrData);
        assert(mmrData->fdata);
        assert(mmrData->fpos>=0);
        assert(mmrData->chunkSize>0);
        long endPos = mmrData->fpos+mmrData->chunkSize-1;
        //Recalculate fpos based on the value calculated by a
           default MMR splitter function.
```

```
        /*Following is a word count example which adjusts fpos so
           that no words are cut in the middle.
        if (mmrData->fpos > 0) {
                while (mmrData->fdata[mmrData->fpos] != ' ' &&
                   mmrData->fdata[mmrData->fpos] != '\t'
                                && mmrData->fdata[mmrData->fpos] !=
                                   '\r' && mmrData->fdata[mmrData->
                                   fpos] != '\n') {
                        mmrData->fpos++;
                }
        }*/
        //Recalculate endPos based on the value calculated by a
           default MMR splitter function.
        /*Following is a word count example which adjusts endPos so
           that no words are cut in the middle.
        if (endPos < mmrData->fDataLen-1) {
                while (mmrData->fdata[endPos+1] != ' ' && mmrData->
                   fdata[endPos+1] != '\t'
                                && mmrData->fdata[endPos+1] != '\r'
                                   && mmrData->fdata[endPos+1] != '\
                                   n') {
                        endPos++;
                }
        }*/
        //Recalculate chunkSize.
        mmrData->chunkSize = endPos-mmrData->fpos+1;
        return 0;
}
```

```
/*************************************************************/
//Specify the key/value packing method for MPI communication. It is
   called if result from each node should be sent to node 0 for
   reduction.
int mmrPackTypes(const void *key, const void *val, int *keyValLen,
   void **keyVal){
        /*Following is a word count example in which key type is
           string and value type is int.
        char *keyStr = *(char**)key;
        keyValLen[0] = strlen(keyStr)+1;
        keyValLen[1] = 1;
        keyVal[0] = keyStr;
        keyVal[1] = (int*)val;*/
        return 0;
}
/*************************************************************/
//Specify the key/value unpacking method for MPI communication. It
   is called if node 0 should receive result from each node for
   reduction.
int mmrUnpackTypes(void **key, void **val, const int *keyValLen,
   void **keyVal){
        /*Following is a word count example in which key type is
           string and value type is int.
        *key = (char*)keyVal[0];
        *(int*)val = *(int*)keyVal[1];*/
        return 0;
}
/*************************************************************/
```

```
//Specify a mmr reduce function only when identity reduce is
   insufficient.
//If reduce() is defined, mmrReduce() must be defined in a similar
   way.
int mmrReduce(const void *val1, const void *val2, void **val){
        /*Following is a word count example in which values i.e.
           counts are summed up for a same key i.e. word.
        *(int*)val = (int)val1 + (int)val2;*/
        return 0;
}
/*********************************************************/
//Specify a key/value compare function if the final key/value pairs
   should be sorted. Sorting will be done automatically.
int mmrKeyValCmp(const void *v1, const void *v2)
{
        /*Following is a word count example in which keys i.e. words
            are sorted in alphabetic order. Same keys are sorted by
           values.
        keyval_t* kv1 = (keyval_t*)v1;
        keyval_t* kv2 = (keyval_t*)v2;
        int i1 = (int)kv1->val;
        int i2 = (int)kv2->val;
        int ret = strcmp((char *)kv1->key, (char *)kv2->key);
        if (ret)
                return ret;
        else{
                if (i1 < i2)
                        return 1;
```

```
                else if (i1 > i2)
                        return -1;
                else
                        return 0;
        }*/
        return 0;
}
/*************************************************************/
//Specify a splitter function if the default is insufficient. By
   default data is splitted by user defined unit size.
//If splitter() is defined, mmrSplitter() must be defined in a
   similar way.
int splitter(void *data_in, int req_units, map_args_t *out)
{
        mmr_data_t * data = (mmr_data_t *)data_in;
        assert(data);
        assert(out);
        assert(data->fDataLen >= 0);
        assert(data->fdata);
        assert(req_units);
        assert(data->fpos >= 0);

        //End of file reached. Return 0 for no more data
        if (data->fpos >= data->fDataLen) return 0;
        //Set the start of the next data
        out->data = (void *)&data->fdata[data->fpos];
        //Determine the nominal length
        out->length = req_units * data->unit_size;
```

```
        if (data->fpos + out->length > data->fDataLen)
                out->length = data->fDataLen - data->fpos;

        //Recalculate fpos based on the value calculated by a
           default splitter function.
        /*Following is a word count example which adjusts fpos so
           that no words are cut in the middle.
        for (data->fpos += (long)out->length;
                data->fpos < data->fDataLen &&
                data->fdata[data->fpos] != ' ' && data->fdata[data->
                   fpos] != '\t' &&
                data->fdata[data->fpos] != '\r' && data->fdata[data
                   ->fpos] != '\n';
                data->fpos++, out->length++);*/

        return 1;
}
/*********************************************************/
//Specify a reduce function if not identity reduce.
//If reduce() is defined, mmrReduce() must be defined in a similar
   way.
void reduce(void *key_in, void **vals_in, int vals_len)
{
        /*Following is a word count example in which values i.e.
           counts are summed up for a same key i.e. word.
        char * key = (char *)key_in;
        int  * vals = (int*)vals_in;
        int i, sum = 0;
```

```
        assert(vals);

        for (i = 0; i < vals_len; i++)
        sum += vals[i];
        if (key){
        //Call emit() (see MapReduceScheduler.h) to emit a key with
           its reduced value.
        emit(key, (void *)sum);
        }
        else{
        vals[0] = sum;
        }*/
}
/*****************************************************************/
//Specify a partition function if the default is insufficient. By
   default a key/value pair is assigned to a reduce thread based on
   a hash of that key.
//Also, specify it if key is a single value, not an array.
int partition(int reduce_tasks, void* key, int key_size)
{
/*Following is an example for key being a single value, not an array
   .
        //Call default_partition() (see MapReduceScheduler.h), or
           define your own partition function.
  return default_partition(reduce_tasks, (void *)&key, key_size);*/
}
/*****************************************************************/
```

```
//Following is the main function.
int main(int argc, char *argv[]) {
        //MMR initialization. Call it only once before anything is
           done.
        mmrInitialize(argc, argv);

        //Optional: MMR file mapping. Call it only when information
           about the file should be processed before calling
           mmrSetup().
        char *fileName = argv[1]; //Assume that argv[1] is the file
           name.
        mmrMapFile(fileName);
        //Optional: get a buffer pointer to the mapped file. Call it
            after mmrMapFile() or mmrSetup().
        char *fdata = mmrGetMappedFile();
        //Optional: process file head and tail info and calculate
           their length (in bytes). Head/tail means any content
           before/after the data block.
        ...

        /*Set MPI datatypes for key/value pairs. Both a key and a
           value can be:
        (1) a single value of a MPI datatype such as MPI_CHAR,
           MPI_INT, MPI_DOUBLE, and so on;
        (2) an array of a MPI datatype;
        (3) a set of values/arrays of MPI datatypes, e.g. a date
           value can have a MPI_CHAR array for month, a MPI_INT for
           day and a MPI_INT for year.
```

```
        E.g.: if your key is an array of MPI_CHAR and your value is
           a strucuture of date, do the following:
        MPI_Datatype keyValType[4] = {MPI_CHAR, MPI_CHAR, MPI_INT,
           MPI_INT};
        */
        MPI_Datatype keyValType[m+n] = {typeK1, ..., typeKm, typeV1
           ..., typeVn}; //Assume that your key has m MPI datatypes
           and your value has n.

        //Specify MMR arguments.
        mmr_args_t mmrArgs;
        memset(&mmrArgs, 0, sizeof(mmr_args_t));        //Make sure
           all default arguments are set to 0 or NULL.
        //Following arguments are necessary.
        mmrArgs.keyValTypes = m+n; //Specify the total number of key
           /value elements. Default: 2
        mmrArgs.keyTypes = m; //Specify the number of key elements.
           Default: 1
        mmrArgs.keyValType = keyValType; //Specify an array for MPI
           datatypes of key/value elements.
        mmrArgs.map = map; //Map function pointer. Function map()
           must be defined by the user.
        mmrArgs.keyComp = keyComp; //Key comparison function pointer
           . Function keyComp() must be defined by the user.
        //Following arguments are optional. Set them only when
           necessary, otherwise just delete corresponding lines or
           set them to 0 or NULL.
mmrArgs.fileName = fileName;
```

```
mmrArgs.fHeadLen = fHeadLen; //Assume that fHeadLen is the
    length (in bytes) of file head info.
mmrArgs.fTailLen = fTailLen; //Assume that fTailLen is the
    length (in bytes) of file tail info.
mmrArgs.fDataLen = fDataLen; //Specify the data length if no
     file is used, otherwise there is no need to do it.
mmrArgs.unit_size = unitSize; //Number of bytes for one data
     unit. Default: 1.
mmrArgs.hashTableSize = hashTableSize; //Specify size of
    hash table used by mmrReduce(). Default: 1024.
mmrArgs.mmrSplitter = mmrSplitter; //MMR splitter function
    pointer. If NULL, default splitter is used.
mmrArgs.mmrPackTypes = mmrPackTypes; //MMR result packing
    function pointer.
mmrArgs.mmrUnpackTypes = mmrUnpackTypes; //MMR result
    unpacking function pointer.
mmrArgs.mmrReduce = mmrReduce; //MMR reduce function point.
    If NULL, identity reduce function is used, which does
    nothing.
mmrArgs.mmrKeyValCmp = mmrKeyValCmp; //Key-value comparison
    function pointer.
mmrArgs.reduce = reduce; //If NULL, identity reduce function
     is used, which emits a key/val pair for each val.
mmrArgs.splitter = splitter; //If NULL, the array splitter
    is used, which splits the data evenly.
mmrArgs.partition = partition; //If NULL, default partition
    function is a hash function.
mmrArgs.use one queue per task = 1; //Creates one emit queue
```

```
    for each reduce task, instead of per reduce thread. This
    improves time to emit if data is emitted in order, but
    can increase merge time. Default: 0
mmrArgs.L1_cache_size = L1_cache_size; //Size of L1 cache in
    bytes. If NULL, default value is used.
mmrArgs.num map threads = num map threads; //Number of
    threads to run map tasks on. If NULL, default value is
    used.
mmrArgs.num reduce threads = num reduce threads; //Number of
    threads to run reduce tasks on. If NULL, default value
    is used.
mmrArgs.num merge threads = num merge threads; //Number of
    threads to run merge tasks on. If NULL, default value is
    used.
mmrArgs.num procs = num procs; //Maximum number of
    processors to use. If NULL, default value is used.
mmrArgs.key match factor = key match factor; //Magic number
    that describes the ratio of the input data size to the
    output data size. This is used as a hint. If NULL,
    default value is used.

//Set mmr arguments. mmrMapFile() will be called here if not
    called previously.
mmrSetup(&mmrArgs);

//Call either mmr() i.e. MapReduce with MPI or mapReduce() i
    .e. MapReduce without MPI, but not both.
//Call it after mmrSetup().
```

```
final_data_t mmrResult;
mmr(&mmrResult);
//Alternative: final_data_t mrRsult = mapReduce();

//Process the result.
...

//Clean up here if you need to do a second round of mmr().
mmrCleanup();

//Specify MMR arguments for the second round.
memset(&mmrArgs, 0, sizeof(mmr_args_t));        //Make sure
   all default arguments are set to 0 or NULL.
...

//Set mmr arguments for the second round. mmrMapFile() will
   be called here if not called previously.
mmrSetup(&mmrArgs);

//Call either mmr() i.e. MapReduce with MPI or mapReduce() i
   .e. MapReduce without MPI, but not both.
//Call it after mmrSetup().
final_data_t mmrResult;
mmr(&mmrResult);
//Alternative: final_data_t mrRsult = mapReduce();

//Process the result for the second round.
...
```

```
        //Call it only once after everything is done. mmrCleanup()
           will be called here.
        mmrFinalize();


        return 0;
}
```

MMR API Specification

API	Description
int mmrInitialize(int argc, char **argv)	mmr Initialize; Call it only once before anything is done.
int mmrSetup(mmr_args_t *args)	Set mmr arguments. mmrMapFile() will be called if not called previously.
int mmr(final_dat_t *mmrResult)	MapReduce with MPI_Datatype. Call it after mmrSetup().
int mmrCleanup()	Call it after mmr().
int mmrFinalize()	Call it only once after everything is done.

Table 6.1: MMR API functions that must be called

API	Description
int mmrMapFile(char *fileName)	Call it only when information about the file should be processed before calling mmrSetup().
char *mmrGetMappedFile()	Call it after mmrMapFile() or mmrSetup().
long mmrGetFileLen()	Call it after mmrMapFile() or mmrSetup().
char *mmrGetRawData()	Call it after mmrSetup().
long mmrGetRawDataLen()	Call it after mmrSetup().
double mmrGetDuration(struct timeval start, struct timeval end)	Call it anytime.
final_data_t mapReduce()	Call it only when you want to do a manual map_reduce. Don't call mmr() if you call this.

Table 6.2: Optional MMR API function calls

Function name	Description
int keyComp(const void *key1, const void *key2)	Specify a key compare function.
void map(map_args_t *args)	Specify a map function.

Table 6.3: Functions that must be implemented by the user

Function name	Description
int partition(int reduce_tasks, void* key, int key_size)	Specify a partition function if the default is insufficient. By default a key/value pair is assigned to a reduce thread based on a hash of that key.
void reduce(void *key_in, void **vals_in, int vals_len)	Specify a reduce function if not identity reduce.
int splitter(void *data_in, int req_units, map_args_t *out)	Specify a splitter function if the default is insufficient. By default data is splitted by user defined unit size.
int mmrSplitter(mmr_data_t* mmrData)	Specify a mmr splitter function if the default mmr splitter is insufficient. By default data is splitted by user defined unit size.
int mmrPackTypes(const void *key, const void *val, int *keyValLen, void **keyVal)	Specify the key/value packing method. Call it if result from each node will be sent to node 0.
int mmrUnpackTypes(void **key, void **val, const int *keyValLen, void **keyVal)	Specify the key/value unpacking method. Call it if node 0 will receive result from each node.
int mmrReduce(const void *val1, const void *val2, void **val)	Specify a mmr reduce function only when identity reduce is insufficient.
int mmrKeyValCmp(const void *v1, const void *v2)	Specify a key/value compare function if the final key/value pairs will be sorted. Sorting will be done automatically.

Table 6.4: Optional user-defined functions.

int keyValTypes	Specify the total number of key/value elements. For example, if a key is a structure with a int and a string, and value is a double array, keyValTypes should be 2+1=3.
int keyTypes	Specify the number of key elements. For example, if a key is a structure with a int and a string, keyTypes should be 2.
MPI_Datatype *keyValType	Specify an array for types of key/value elements. For example, if a key is a structure with a int and a string, and value is a double array, keyValType should be {MPI_INT, MPI_CHAR, MPI_DOUBLE}.
map_t map	Map function pointer, must be user defined.
key_cmp_t keyComp	Key comparison function, must be user defined.

Table 6.5: Arguments that must be specified

char *fileName	Input file name.
long fHeadLen	Specify the head length if the file has a head.
long fTailLen	Specify the tail length if the file has a tail.
long fDataLen	Specify the data length if no file is used. If a file will be used, no need to specify it.
int unit_size	# of bytes for one unit. Default value is 1.
int hashTableSize	Specify size of hash table used by mmrReduce(). Default is 1024.
mmr_splitter_t mmrSplitter	mmr splitter function pointer. If NULL, default splitter is used.
mmr_pack_types_t mmrPackTypes	mmr result packing function pointer.
mmr_unpack_types_t mmrUnpackTypes	mmr result unpacking function pointer.
mmr_reduce_t mmrReduce	mmr reduce function point. If NULL, identity reduce function is used, which does nothing.
mmr_keyvalcmp_t mmrKeyValCmp	Key-value comparison function pointer.
reduce_t reduce	If NULL, identity reduce function is used, which emits a key/val pair for each val.
splitter_t splitter	If NULL, the array splitter is used.
partition_t partition	Default partition function is a hash function.
int use_one_queue_per_task	Creates one emit queue for each reduce task, instead of per reduce thread. This improves time to emit if data is emitted in order, but can increase merge time.
int L1_cache_size	Size of L1 cache in bytes. If NULL, default value is used.
int num_map_threads	# of threads to run map tasks on. If NULL, default value is used.
int num_reduce_threads	# of threads to run reduce tasks on. If NULL, default value is used.
int num_merge_threads	# of threads to run merge tasks on. If NULL, default value is used.
int num_procs	Maximum number of processors to use. If NULL, default value is used.
float key_match_factor	Magic number that describes the ratio of the input data size to the output data size. This is used as a hint. If NULL, default value is used.

Table 6.6: Optional arguments (must be zero if not used)

Vita

Liqiang Wang received his B.S. (2004) degree in Computer Networks from the Huazhong Normal University, Wuhan, China. From the year 1999 to 2002, he worked as a network engineer in the Ganko Information System Engineering Co. Ltd, and from 2003 to 2006 he was a senior design engineer and technical architect in the Speednet Network & Systems Integration Co. Ltd., Wuhan, China. In the year 2007, he started his master study in the Department of Computer Science at the University of New Orleans, where he has been working with Dr. Vassil Roussev as his research assistant.