Map Reduce- A Simplified Data Processing For Large Clusters

-By
Chen Sun
Shen Cheng
Namratha Bharadwaj

Instructor:
Prof. Ming Hwa Wang
Santa Clara University

Table Of Contents

Abstract..4

1. Introduction……………………………………………………5

1.1 Objective..5

1.2 What is the problem..5

1.3 Approach Study 5

1.4 Area or scope of investigation 5

2.Theoretical basis and literature overview…………………………..6

2.1 Definition of the problem……………………………………...6

2.2 Theoretical background of the problem...6

2.3 Related research to solve the problem 7

2.4 Our Solution to solve this problem 8

3. Project Goals………………………………………………….8

4. Methodologies…………………………………………………...8

4.1. How to generate/Collect input data...9

4.2 How to Solve the Problem 9

4.2.1 Algorithm Design...9

4.2.2 Evaluation Plan 9

4.2.3 Language Used:...10

4.3. How to generate output...10

5. Implementation………………………………………………10

5.1. Code 10

5.2. design document and flowchart 10

6. Data Analysis and Discussion:………………………………………14

6.1. Output Generation: 14

7. Conclusions and recommendations……………………………………15

7.1. Summary and Conclusions 15

7.2. Future Recommendations 16

8. Appendix (Source Codes)……………………………………………………………………………...17

8.1 Sequential Program for word count 17

8.1.1. Word_count.cpp 17

8.1.2. Split_file.hpp 25

8.2. Map-reduce for a word count application 27

8.2.1. Main.cpp..27

8.2.2. Manager_stage3.cpp 28

8.3. Mapper and Reducer code for Histogram…………..……………………………...38

8.3.1. Histogram.cpp………………………………………………………………….38

8.3.2. Main.cpp……………..………………………………………………………..42

Abstract

Map-Reduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key .

Processing a very large set of data on a single computer could be very slow. Therefore, in our project, the goals what we want to achieve are listed as follows.

First of all, we want to prove that Map-Reduce method can greatly increase the speed of processing large volume of data in a cluster environment, even if the cluster only consists of relatively cheap personal computers.

Secondly, we also want to show that load balancing is helpful in a heterogeneous environment. Google's map-reduce implicitly assumes that computer nodes in a cluster are homogeneous in nature. This assumption may not be optimal and would in fact degrade performance in some cases.

1. Introduction

1.1 Objective

MapReduce is a framework used by Google for processing huge amounts of data in a distributed environment and Hadoop is Apache's open source implementation of the MapReduce framework. Due to the simplicity of the programming model and the run-time tolerance for node failures, MapReduce is widely used for not only commercial applications but also scientific computations. Facebook uses a Hadoop cluster composed of hundreds of nodes to process terabytes of user data. The New York Times rents a Hadoop cluster from Amazon EC2 to convert millions of articles. Michael C. Schatz introduced MapReduce to parallelize blast which is a DNA sequence alignment program and achieved 250 times speedup. [1]

Hadoop is a distribute computing platform written in Java. It incorporates features similar to those of the Google File System and of MapReduce[2]. So hadoop is a basic library which should be used in our project to implement one feasible application.

While our project team were thinking about the realistic implementation examples of MapReduce, we came up ideas about the human vascular system's image forming, imaging chip noise detection, and word count in large number of documents.

In this project we determine to split our project into 3 milestones:
1). Implementing an word count application in one node MapReduce Implementation;
2). Enhancing the word count application by distributing the 'slaves' on 3 nodes (Three Computers via SOHO Ethernet)
3). Replacing the application on milestone 2 by imaging chip noise detection.

1.2 What is the problem

As shown before, these three' objectives' implementation is the basic tasks we need to accomplish. In future work, our first problem would be spend on MapReduce's software implementation, central problem may be a C version mapreduce() function; The second one may be the implementation's networking experiment, basically the 3 personal computer network's simulation; The third one is certainly the software implementation of imaging chip noise detection, central problem is its mathematical Modeling to implement map and reduce function.
1.3 Why this is a project related to this class
The general concept of this project is implement true big data MepReduce implementation step by step. Hadoop library should basically deal with the scheduling algorithm and threads' collaboration strategy. This is of course related to master's Operating System course.

1.3 Approach Study

Referring to 2.1 and 2.2.

1.4 Area or scope of investigation

The area or scope of investigation is concentrated on the MapReduce code implementation of word count application and. The major efforts would be paid on the investigation of Map-reduce's efficiently utilization, algorithm of scheduling and algorithm of task transferring between nodes.

2. Theoretical basis and literature overview

2.1 Definition of the problem

Over the past five years, many programmers have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

Furthermore, the performance of a parallel system like MapReduce system closely ties to its task scheduler. Many researchers have shown their interest in the schedule problem. The current scheduler in Hadoop uses a single queue for scheduling jobs with a FCFS (First Come First Serve) method. Using these scheduler, people could assign jobs to queues which could manually guarantee their specific resource share.

As a reaction to the complexity, described in paragraph 1, we are designing a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Further, we concentrate on the problem of how to improve the hardware utilization rate when different kinds of workloads run on the clusters in MapReduce framework. In practice, different kinds of jobs often simultaneously run in the data center. These different jobs make different workloads on the cluster, including the I/O-bound and CPU-bound workloads.

2.2 Theoretical background of the problem

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. A typical MapReduce computation processes many terabytes of data on thousands of machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

As with other parallel applications, the performance and energy efficiency of MapReduce applications is affected by the degree of parallelism and computational intensity (i.e., the ratio of on-chip CPU computation to off-chip memory and I/O access). Given an application, the optimal efficiency will be achieved when resource allocation matches application characteristics. Specifically, the number of allocated processing cores matches the degree of parallelism of the application; and the processor performance state matches the application's computational

intensity.
Further, as the Internet scale keeps growing up, enormous data needs to be processed by many Internet Service Providers. MapReduce framework is now becoming a leading example solution for this. MapReduce is designed for building large commodity cluster, which consists of thousands of nodes by using commodity hardware. In this environment, many people share the same cluster for different purpose. This situation has led to different kinds of workloads running on the same data center. For example, these clusters could be used for mining data from logs which mostly depends on CPU capability. At the same time, they also could be used for processing web text which mainly depends on I/O bandwidth.

As a reaction to the complexity mentioned in paragraph 1 of this section, In our project we will be designing a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. Most of our computations would involve applying a map operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately.

Furthermore we will be designing a new triple-queue scheduler which consist of a workload predict mechanism MR-Predict and three different queues (CPU-bound queue, I/O-bound queue and wait queue). We classify MapReduce workloads into three types, and our workload predict mechanism automatically predicts the class of a new coming job based on this classification. Jobs in the CPU-bound queue or I/O-bound queue are assigned separately to parallel different type of workloads

2.3 Related research to solve the problem

The following section gives a summary of the papers that have been read and reviewed:

1) "Evaluating MapReduce for Multi-core and Multiprocessor Systems"-Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos KozyrakisComputer Systems Laboratory. Stanford University
2) "A Dynamic MapReduce Scheduler for Heterogeneous Workloads"- Chao Tian, Haojie Zhou, Yongqiang He, Li Zha1. Published during 2009 Eighth International Conference on Grid and Cooperative Computing
3) "Improving MapReduce Energy Efficiency for Computation Intensive Workloads"- Thomas Wirtz and Rong Ge.
4) "MapReduce: Simplified Data Processing on Large Clusters"- Jeffrey Dean and Sanjay Ghemawat, Google, Inc.
5) "Improving MapReduce Performance through Data Placement in
6) Heterogeneous Hadoop Clusters"- Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian,
7) James Majors, Adam Manzanares, and Xiao Qin- April 2010
8) "Improving MapReduce Performance through Complexity and Performance based Data placement in heterogeneous hadoop clusters" - Rajashekhar M. Arasanal, Daanish U. Rumani
9) "MapReduce for Data Intensive Scientific Analyses" - Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox

10)"Map-reduce as a Programming Model for Custom Computing Machines" - Jackson H.C. Yeung1, C.C. Tsang1, K.H. Tsoi1, Bill S.H. Kwan1, Chris C.C. Cheung2, Anthony P.C. Chan2 and Philip H.W. Leong - 2008 IEEE

11)"Evaluating MapReduce for Multi-core and Multiprocessor Systems" – Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis, Computer Systems Laboratory, Standord University

Web Links Referenced:
http://www.ibm.com/developerworks/cloud/library/cl-mapreduce/
http://www.ubuntu.com/content/ubuntu-and-hadoop-perfect-match
http://developer.yahoo.com/hadoop/tutorial/module4.html#pipes
http://pages.cs.wisc.edu/~gibson/mapReduceTutorial.html#REF
https://portal.futuregrid.org/manual/hadoop-wordcount
http://wiki.apache.org/hadoop/C%2B%2BWordCount
http://man7.org/linux/man-pages/man3/CPU_SET.3.html

2.4 Our Solution to solve this problem

We have defined 3 milestones to achieve our goal and exhibit the power of parallel programming in a large distributed environment. As a part of 1st milestone, we used an existing map-reduce library to parallelize the Histogram application and to get familiar with the concept of map-reduce and its usage. As we progressed to the 2nd milestone, implemented a sequential program word the word count application. Finally as a part of the 3rd milestone we implemented the entire map-reduce library for the word count application and compared its run time of it with that of the sequential program and formulated the results.

3. Project Goals

Processing a very large set of data on a single computer could be very slow. Therefore, in our project, the goals that we want to achieve are listed as follows.
First of all, we want to prove that MapReduce method can greatly increase the speed of processing large volume of data in a cluster environment, even if the cluster only consists of relatively cheap personal computers.
Secondly, we also want to show that load balancing is helpful in a heterogeneous environment. Google's Map-reduce implicitly assumes that computer nodes in a cluster are homogeneous in nature. This assumption may not be optimal and would in fact degrade performance in some cases. For example, in a heterogeneous cluster, which is exactly what we will be building for our project, we believe that the default implementation of MapReduce is not the most efficient way in terms of speed and hardware utilization.

4. Methodologies

We designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

4.1. How to generate/Collect input data

We downloaded huge text files of different sizes in order to test our program with various input sizes.

4.2 How to Solve the Problem

4.2.1 Algorithm Design

In a heterogeneous cluster, the computing capacities of nodes may vary significantly. A high-speed node can finish processing data stored in a local disk of the node faster than low-speed counterparts. Compared with low-performance nodes, high performance nodes are excepted to store and process more file fragments.

We have defined 3 milestones in our project. We would first develop a mapper and reducer functions for a histogram application using an existing library
We then developed a sequential program for a word count application
Further we proceeded to implement our own map-reduce library that works for a word count application. The results of this stage was compared with the results of the sequential program and the results were formulated

The program flow is as follows:

Step 1: Preparation stage: During this stage, the master assigns an input file to each of the 3 workers. The input files used are large text documents

Step 2: Map Stage: During this stage, the map- workers process each of the files independent of one another and generate key- value pairs as intermediate result.

Step 3: Shuffle Stage: The intermediate results from the map- stage are returned to the master who, through an efficient hash function distributes these key-value pairs to the reduce- workers for consolidation.

Step 4: Reduce Stage: The output of the shuffle stage serves as input to the reduce workers that merge all intermediate values associated with the same intermediate key

4.2.2 Evaluation Plan

As for our evaluation plan, we will conduct a number of test cases to evaluate our implementations. We choose Word-count as our test application.

Firstly we would implement a sequential program for word count.
Secondly, we generate different size of input files, so we can compare the efficiency of MapReduce that of the sequential program for word count.
Finally, we will draw conclusions based on the data and information obtained from above tests.

4.2.3 Language Used:

We used C- Language to develop the map-reduce framework.

4.4. How to generate output

Once the sequential program was executed, we logged the time taken for the program to run. Similarly we recorded the time taken for the map-reduce to complete its execution. We then compared the performance of both the techniques and plotted the results as a function of time.

5. Implementation

5.1. Code

The entire set of source code pertaining to our project can be found at the appendix section of this document

5.2. Design document and flowchart

Stage 1: Map-reduce for Histogram:

To get started, we first found an open-source map-reduce library which is called Phoenix. Phoenix targets shared-memory systems of MapReduce model for large and data-intensive tasks. The Phoenix library is C-language based and can be used to program multi-core processors as well as shared-memory multiprocessors.

The Map function processes the input data and generates a set of intermediate key/value pairs. The Reduce function properly merges the intermediate pairs which have the same key. Given such a functional specification, the MapReduce runtime automatically parallelizes the computation by running multiple maps and/or reduce tasks in parallel over disjoined portions of the input or intermediate data. Google's MapReduce implementation facilitates processing of terabytes on clusters with thousands of nodes. The Phoenix implementation is based on the same principles but targets shared-memory systems such as multi-core chips and symmetric multiprocessors.

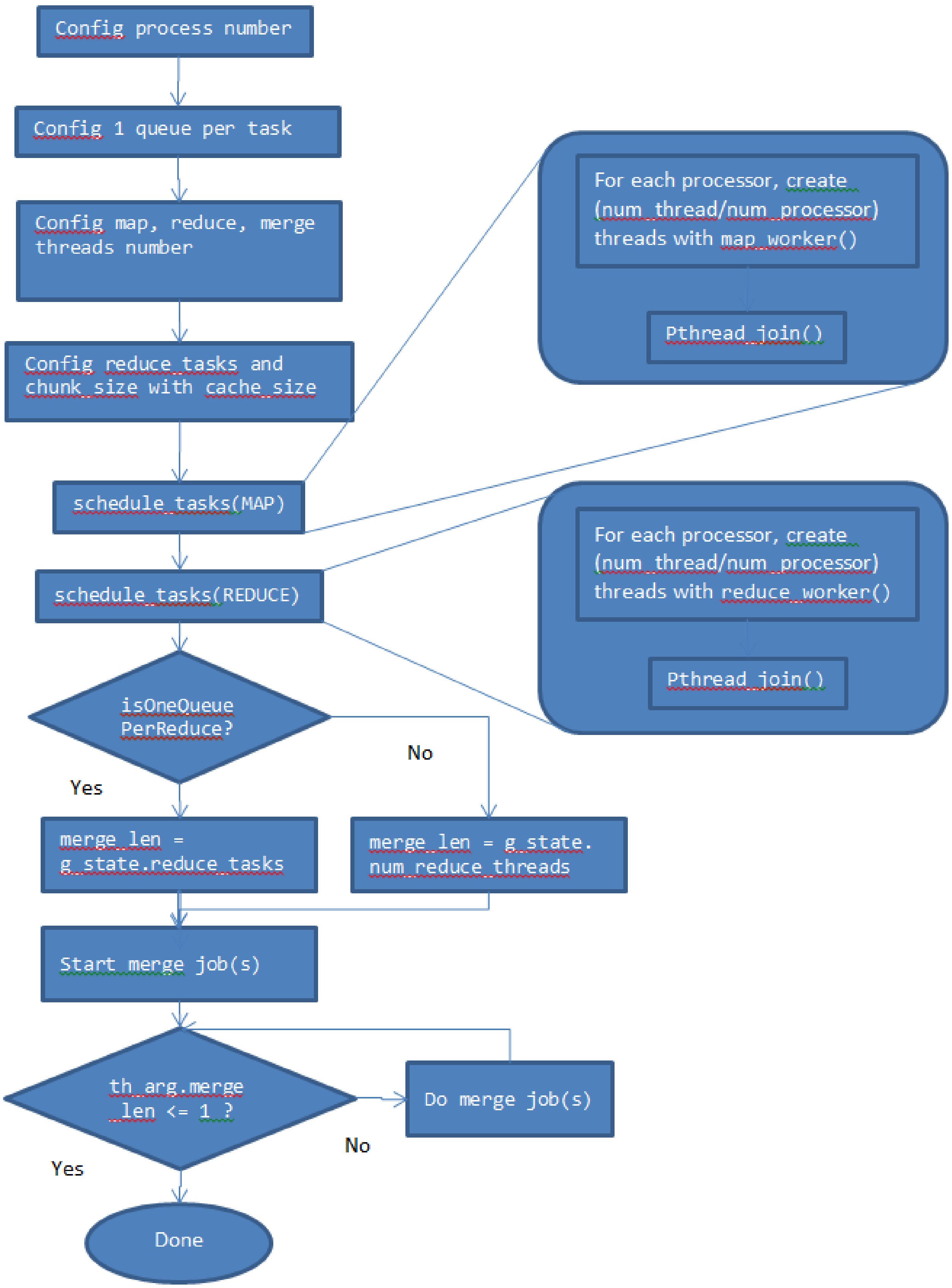
Phoenix uses threads to spawn parallel Map or Reduce tasks. It also uses shared-memory buffers to facilitate communication without excessive data copying. The runtime schedules tasks dynamically across the available processors in order to achieve load balance and maximize task throughput. Locality is managed by adjusting the granularity and assignment of parallel tasks. The runtime automatically recovers from transient and permanent faults during task execution by repeating or re-assigning tasks and properly merging their output with that from the rest of the computation. Overall, the Phoenix runtime handles the complicated concurrency, locality, and fault-tolerance tradeoffs that make parallel programming difficult.

To familiarize us with Phoenix library, we designed a sample image processing application – a histogram generator. The application is used to generate histograms for image files. The input file format is standard BMP format. The outputs are the three histograms for each of the red, green and blue colors.

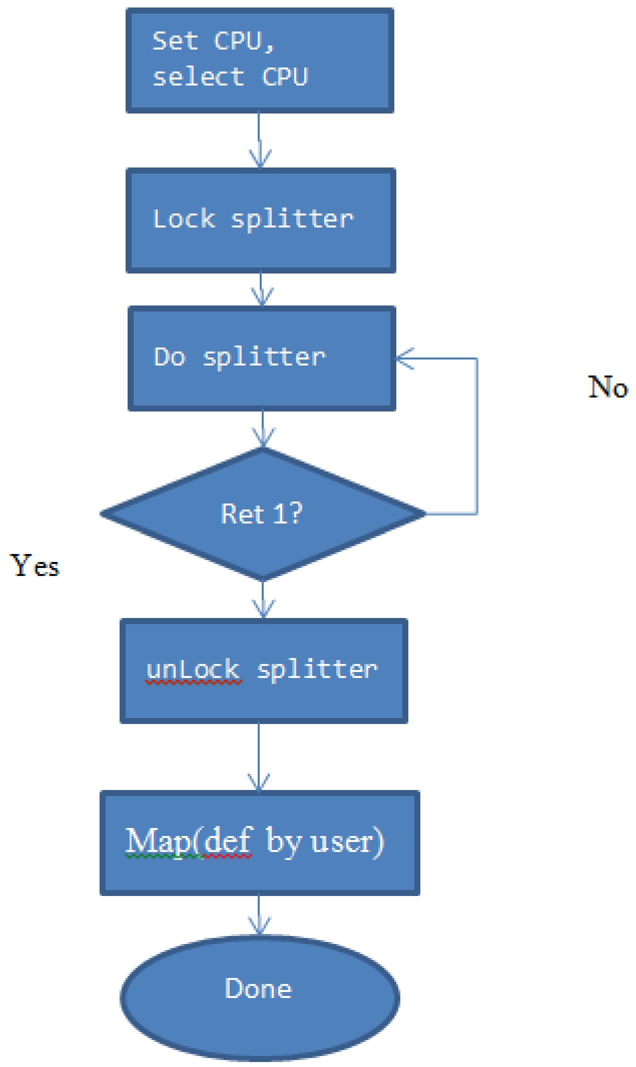
The flow charts representing the program flow for stage 1 are as follows:

map_reduce_scheduler:
the map_reduce_scheduler(scheduler_args_t * args) related to below important procedure:
1. scheduler_init
2. schedule_tasks
3. map_worker
4. reduce_worker
5. merge_worker

The map_reduce_scheduler function is the most important part of this library frame work. At the very beginning, the user should decide how many processors would be used to do the map, reduce, merge tasks. So the first step is configurations of processor numbers. Basically, the program set 1 task per queues ask default. Next is to set threads numbers and the reduce tasks is closely related to cache_size cause this frame work supposed that after map is down, things in cache would facilitate the task obviously.
After configurations, it comes to schedule_tasks, for map and reduce, the function all creates several threads for each processor, the threads creating job works fairly, but there is some difference between map and reduce_worker function which are the actual jobs in each thread.
There is one loop in the merge procedure. If the merge job is more than 1, just loop and do merge job. Otherwise the whole task comes to an end.


Config process number
Config 1 queue per task
Config map, reduce, merge threads number
Config reduce tasks and chunk size with cache size
schedule tasks(MAP)
schedule tasks(REDUCE)
For each processor, create (num thread/num processor) threads with map worker()
Pthread join()
For each processor, create (num thread/num processor) threads with reduce worker()
Pthread join()
isOneQueue PerReduce?
Yes
No
merge len = g state.reduce tasks
merge len = g state. num reduce threads
Start merge job(s)
th arg.merge len <= 1 ?
Do merge job(s)
Yes
No
Done

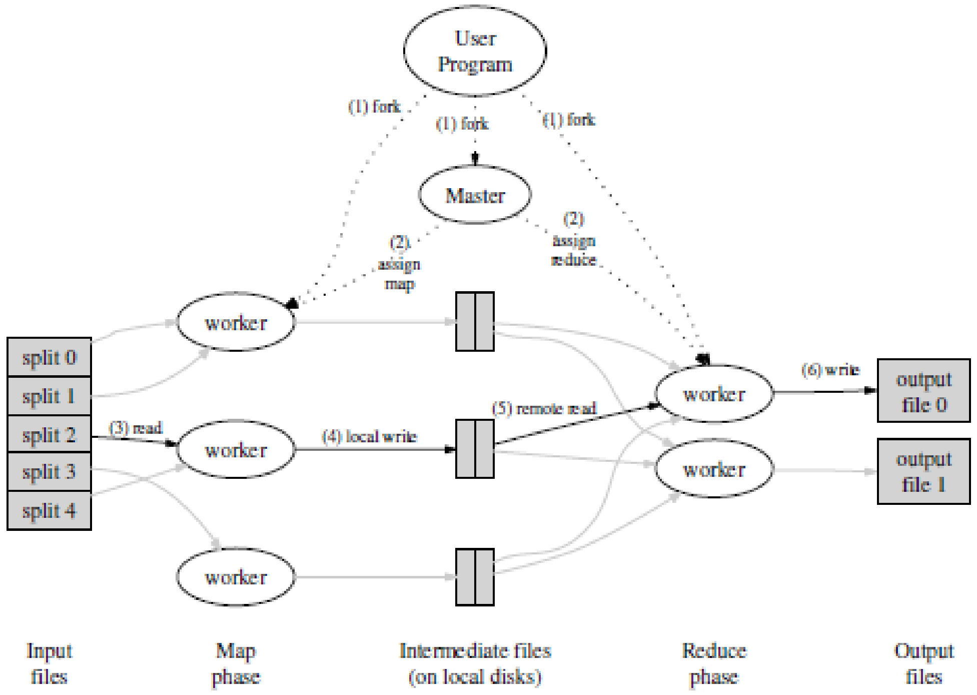
Here we come to the map_worker function, which works in each thread. First lock the splitter, and do split, if it's done, unlock splitter and start map function which is defined by the user; Or just loop and continue to split inputs.



Stage 2: A sequential program for word count

Stage 3: Our implementation of map-reduce for word count

Process flow diagram:



Execution Overview

Preparation stage: During this stage, the master assigns an input file to each of the 3 workers. The input files used are large text documents

Map Stage: During this stage, the map- workers process each of the files independent of one another and generate key- value pairs as intermediate result.

Shuffle Stage: The intermediate results from the map- stage are returned to the master who, through an efficient hash function distributes these key-value pairs to the reduce- workers for consolidation.

Reduce Stage: The output of the shuffle stage serves as input to the reduce workers that merge all intermediate values associated with the same intermediate key

6. Data Analysis and Discussion:

6.1. Output Generation:

Stage 1: We ran the map-reduce frame work for a histogram application. When using a 100MB BMP file, the phoenix program took 0.25 s, while the sequential program took 0.31 s to finish.

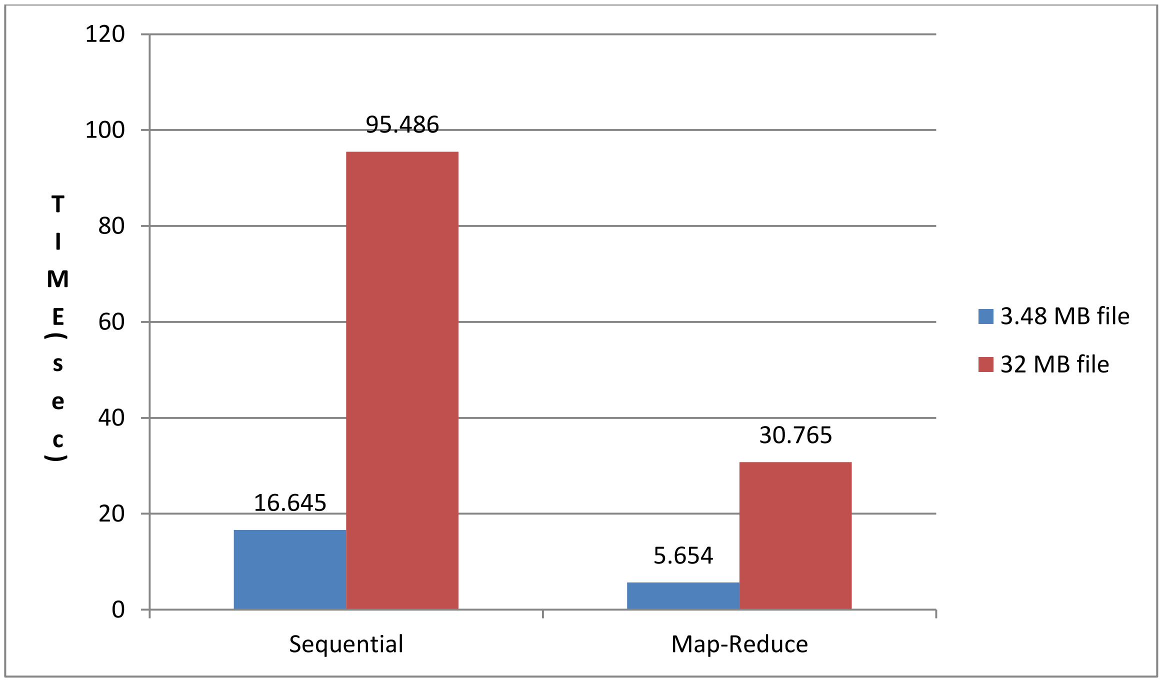
Stage 2: We ran the sequential program for a word count application for various sets of input files and following were the results.

File Size in MB	Time Taken in sec
3.48 MB	16.645 sec
32 MB	95.486 sec

Stage 3: We ran the map-reduce program for the same set of input files as that of sequential program and following were the results.

File Size in MB	Time Taken in sec
3.48 MB	5.654
32 MB	30.765

The comparison of stage 2 and 3 are as follows:



7. Conclusions and recommendations

7.1. Summary and Conclusions

The Map-Reduce programming model has been successfully used at Google, yahoo and several other companies for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as Map-Reduce computations. For example, Map-Reduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of Map-Reduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems

7.2. Future Recommendations

Our implementation of map-reduce is a simulation of the one in a distributed environment. This could be further enhanced to work on a cluster of commodity machines with huge sets of input data in the range of terabytes to witness the actual power of map-reduce

8. Appendix (Source Codes)

8.1 Sequential Program for word count

8.1.1. Word_count.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <pthread.h>
#include <unistd.h>

#include <iostream>
#include <fstream>

#include <string>
#include <cstring>

#include <boost/lexical_cast.hpp>
#include <boost/regex.hpp>
#include <boost/config.hpp>
#include <boost/algorithm/string/split.hpp>
#include <boost/algorithm/string.hpp>

#include <sys/stat.h>
#include <sys/time.h>
#include <ctime>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <dirent.h>
#include <split_file.hpp>

#define FALSE 0
#define TRUE  1
#define FOUND 1
#define NOT_FOUND 0

using namespace std;
using namespace boost;
using namespace boost::algorithm;

typedef vector< string > split_vector_type;
struct timeval start_t, end_t;
```

```
typedef struct node{
	string s_file_name;
	int i_slave_id;
	int i_is_done;
	node *p_next;
}node;
typedef struct res_node{
	string s_word;
	int i_frequency;
	res_node *p_next;
}res_node;
pthread_mutex_t counter_mutex_job_list = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t counter_mutex_count = PTHREAD_MUTEX_INITIALIZER;

node *p_glob_head;
res_node *p_glob_A_result_head = new res_node;
res_node *p_glob_B_result_head = new res_node;

int is_empty_sentence(string);
void print_linked_list(node *);
void print_result_linked_list(res_node *);
int find_next_file(string);
node *create_jobs_list(int);
int find_job(node *, int);
res_node *mapper(int, node *);
void *slave1_map(void *);
void *slave2_map(void *);
void delete_the_linked_list(node *);
void delete_the_result_linked_list(res_node *);
void mark_job(int, node *);
res_node *reduce(res_node *);

vector<string> cast_to_lower_case(vector<string>);
res_node *cumulate_wordlist(res_node *, split_vector_type);
// reducer()
// generate_result()

int main(int argc, char **argv)
{
	int i_file_num = split_file("Word_Count_input.txt");
	int sec, usec;
	// split the whole input file into several tasks
	p_glob_head = create_jobs_list(i_file_num);
	// for debug use
	// print_linked_list(p_head);
	gettimeofday(&start_t, NULL);
	pthread_t tidA;
	pthread_t tidB;
	pthread_create(&tidA, NULL, &slave1_map, NULL);
	pthread_create(&tidB, NULL, &slave2_map, NULL);

	pthread_join(tidA, NULL);
	pthread_join(tidB, NULL);
	delete_the_linked_list(p_glob_head);
	// generate_result();
	// print_result_linked_list(p_glob_A_result_head);
	res_node *p_reduce_result_head = new res_node;
	p_reduce_result_head->i_frequency = -1;
	p_reduce_result_head->s_word = "";
	p_reduce_result_head->p_next = NULL;
	p_reduce_result_head = reduce(p_reduce_result_head);
	print_result_linked_list(p_reduce_result_head);
	gettimeofday(&end_t, NULL);
	sec = (int)end_t.tv_sec - (int)start_t.tv_sec;
	if (end_t.tv_usec >= start_t.tv_usec)
	{
		usec = (int)end_t.tv_usec - (int)start_t.tv_usec;
	}

	cout <<endl<<"Time costs:"<<sec<<". "<<usec<<endl;
	delete_the_result_linked_list(p_glob_A_result_head);
	delete_the_result_linked_list(p_glob_B_result_head);
	delete_the_result_linked_list(p_reduce_result_head);
}

int is_empty_sentence(string source_str)
{
	if ( source_str == "")
		return TRUE;
	regex r_pattern("\\s*");
	smatch sm;
	if(regex_match (source_str, sm, r_pattern))
		return TRUE;
	else
		return FALSE;
}

void print_linked_list(node *p_head)
{
	node *p_itr = p_head->p_next;
	while(p_itr != NULL)
	{
		cout<<p_itr->s_file_name<<"'s slaver is: "<<p_itr->i_slave_id<<endl;
		p_itr = p_itr->p_next;
	}
	cout<<"+++++++print over+++++++++"<<endl;
```

```
}
void print_result_linked_list(res_node *p_head)
{
	res_node *p_itr = p_head->p_next;
	while(p_itr != NULL)
	{
		cout<<" "<<p_itr->s_word<<" :  "<<p_itr->i_frequency<<endl;
		p_itr = p_itr->p_next;
	}
	cout<<"++++++++print over+++++++++"<<endl;
}
int find_next_file(string name)
{
	// size_t len = name.length();
	DIR *dirp = opendir(INPUT_OUTPUT_DIR);
	struct dirent *dp;
	while ((dp = readdir(dirp)) != NULL)
	{
		int compare_result = strncmp(dp->d_name, name.c_str(), 1);
		if ( !compare_result )
		{
			(void)closedir(dirp);
			return FOUND;
		}
	}
	(void)closedir(dirp);
	return NOT_FOUND;
}

// make linked list of all the files's name and their completing-state
node *create_jobs_list(int i_file_num)
{
	node *p_prev = NULL;
	node *p_new = new node;
	node *p_head = p_new;
	p_head->s_file_name = "0";
	p_head->i_is_done = TRUE;
	p_prev = p_head;
	int name = 1;
	while(TRUE)
	{
		if(name >= i_file_num)
			break;
		string s_name = boost::lexical_cast<string>(name);
		if (find_next_file(s_name))
		{
			node *p_new = new node;
			p_prev->p_next = p_new;
```

```
				p_new->s_file_name = s_name;
				p_new->i_is_done = FALSE;
				p_new->i_slave_id =  (name % 2 == 0) ? 2:1 ;
				name++;
				p_prev = p_new;
			}
		}
		p_prev->p_next = NULL;
		return p_head;
}
int find_job(node *p_head, int i_slave_id)
{
		node *p_itr = p_head;
		while (p_itr->p_next != NULL)
		{
			if((p_itr->i_slave_id == i_slave_id) && !(p_itr->i_is_done))
				return (boost::lexical_cast<int>(p_itr->s_file_name));
			p_itr = p_itr->p_next;
		}
		return 0;
}
res_node *cumulate_wordlist(res_node *p_head, split_vector_type SplitVec)
{
		p_head->i_frequency = -1;
		p_head->s_word = "";
		int i_this_word_dealt = FALSE;
		res_node *p_itr = p_head;
		for( size_t i = 0; i < SplitVec.size(); i++)
		{
			i_this_word_dealt = FALSE;
			p_itr = p_head;
			while(p_itr->p_next != NULL)
			{
				if(	!(SplitVec[i].compare(p_itr->s_word))&&(p_itr->i_frequency	!=	-1)
&&(SplitVec[i].length() != 0))
				{
					p_itr->i_frequency++;
					i_this_word_dealt = TRUE;
					break;
				}
				p_itr = p_itr->p_next;
			}
			if((i_this_word_dealt == FALSE)&&(SplitVec[i].length() != 0))
			{
				res_node *p_new = new res_node;
				p_new->s_word = SplitVec[i];
				p_new->i_frequency = 1;
				p_itr->p_next = p_new;
```

```
                        p_new->p_next = NULL;
                }
        }
        return p_head;
}

res_node *mapper(int job_id, res_node *p_result_head)
{
        ifstream myfile;
        res_node *p_return_head;
        string s_job_id = boost::lexical_cast<string>(job_id);
        string s_first_line_from_source_file = "ERIC";
        split_vector_type SplitVec;
        string s_open_file_name = INPUT_OUTPUT_DIR + s_job_id;
        myfile.open (s_open_file_name.c_str());
        while ( myfile.good())
        {
                getline (myfile, s_first_line_from_source_file);
                if( is_empty_sentence(s_first_line_from_source_file) )
                        continue;
                split(SplitVec, s_first_line_from_source_file,
                         is_any_of(" \t\"\',.-!?:;\\\n\r"), token_compress_on );
                SplitVec = cast_to_lower_case( SplitVec );
                p_return_head = cumulate_wordlist(p_result_head, SplitVec);
        }
        myfile.close();
        return p_return_head;
}
vector<string> cast_to_lower_case(vector<string> SplitVec)
{
        for (size_t i = 0; i < SplitVec.size(); i++)
        {
                string s_result = SplitVec[i];
                std::transform(s_result.begin(), s_result.end(),s_result.begin(), ::tolower);
                SplitVec[i] = s_result;
        }
        return SplitVec;
}
void *slave1_map(void *vptr)
{
        int i_slave_id = 1;
        int job_id = 0;
        while(TRUE)
        {
                pthread_mutex_trylock(&counter_mutex_job_list);
                pthread_mutex_unlock(&counter_mutex_job_list);
                job_id = find_job(p_glob_head, i_slave_id);
                if( !job_id )
```

```
				{
					// for debug use
					// print_result_linked_list(p_glob_A_result_head);
					pthread_exit(NULL);
				}
				p_glob_A_result_head = mapper(job_id, p_glob_A_result_head);
				mark_job(job_id, p_glob_head);
		}
		pthread_exit(NULL);
}
void *slave2_map(void *vptr)
{
		int i_slave_id = 2;
		int job_id = 0;
		while(TRUE)
		{
				pthread_mutex_trylock(&counter_mutex_job_list);
				pthread_mutex_unlock(&counter_mutex_job_list);
				job_id = find_job(p_glob_head, i_slave_id);
				if( !job_id )
					pthread_exit(NULL);
				p_glob_B_result_head = mapper(job_id, p_glob_B_result_head);
				mark_job(job_id, p_glob_head);
		}
		pthread_exit(NULL);
}
res_node *reduce(res_node *p_reduce_result_head)
{
		res_node *p_itr = p_glob_A_result_head->p_next;
		res_node *p_res_itr = p_reduce_result_head;
		res_node *p_res_itr_prev = p_reduce_result_head;
		int i_slaver = 1;
		for(;i_slaver<=2; i_slaver++)
		{
				while (p_itr != NULL)
				{
					p_res_itr = p_reduce_result_head;
					p_res_itr_prev = p_reduce_result_head;
					while (p_res_itr != NULL)
					{
						if( !((p_res_itr->s_word).compare(p_itr->s_word)) )
						{
							p_res_itr->i_frequency += p_itr->i_frequency;
							break;
						}
						p_res_itr_prev = p_res_itr;
						p_res_itr = p_res_itr->p_next;
					}
```

```
                if(p_res_itr_prev->p_next == NULL)
                {
                        res_node *p_new = new res_node;
                        p_new->s_word = p_itr->s_word;
                        p_new->i_frequency = p_itr->i_frequency;
                        p_new->p_next = NULL;
                        p_res_itr_prev->p_next = p_new;
                }
                p_itr = p_itr->p_next;
        }
        p_itr = p_glob_B_result_head->p_next;
    }
    return p_reduce_result_head;
}

void mark_job(int job_id, node *p_head)
{
    node *p_itr = p_head;
    string s_job_id = boost::lexical_cast<string>(job_id);
    while (p_itr->p_next != NULL)
    {
        if(p_itr->s_file_name == s_job_id)
        {
            p_itr->i_is_done = TRUE;
            break;
        }
        p_itr = p_itr->p_next;
    }
}
void delete_the_linked_list(node *p_head)
{
    node *p_iterator = NULL;
    node *p_prev_iterator = NULL;
    p_iterator = p_head;
    p_prev_iterator = p_iterator;
    while (p_iterator->p_next != NULL)
    {
        p_iterator = p_iterator->p_next;
        delete p_prev_iterator;
        p_prev_iterator = p_iterator;
    }
    p_prev_iterator = NULL;
    delete p_iterator;
}
void delete_the_result_linked_list(res_node *p_head)
{
    res_node *p_iterator = NULL;
    res_node *p_prev_iterator = NULL;
```

```
        p_iterator = p_head;
        p_prev_iterator = p_iterator;
        while (p_iterator->p_next != NULL)
        {
                p_iterator = p_iterator->p_next;
                delete p_prev_iterator;
                p_prev_iterator = p_iterator;
        }
        p_prev_iterator = NULL;
        delete p_iterator;
}
```

8.1.2. Split_file.hpp

```
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <pthread.h>
#include <unistd.h>

#include <iostream>
#include <fstream>

#include <string>
#include <cstring>

#include <boost/lexical_cast.hpp>
#include <boost/regex.hpp>
#include <boost/config.hpp>
#include <boost/algorithm/string/split.hpp>
#include <boost/algorithm/string.hpp>

#include <sys/stat.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <dirent.h>

#define FALSE 0
#define TRUE  1
#define FOUND 1
#define NOT_FOUND 0
#define U_TAR_FILE_AVERAGE_SIZE unsigned(1000)
#define INPUT_OUTPUT_DIR ("./files/")

using namespace std;
using namespace boost;
using namespace boost::algorithm;

unsigned count_file_lines(string s_file_name)
{
        string s_open_file_name = s_file_name;
        string s_first_line_from_source_file = "";
        unsigned u_count = 0;
        ifstream myfile;
        myfile.open (s_open_file_name.c_str());
        while ( myfile.good())
        {
                getline (myfile, s_first_line_from_source_file);
                u_count++;
        }
        myfile.close();
        return u_count;
}
// split a big file into several files
int split_file(string s_file_name)
{
        int i_file_name = 1;
        s_file_name = INPUT_OUTPUT_DIR + s_file_name;
        unsigned u_line_num = 0;
        string string_file_name;
        string s_current_line;

        ifstream source_File;
  source_File.open(s_file_name.c_str());

  ofstream new_file;
  string_file_name = boost::lexical_cast<string>(i_file_name);
  string_file_name = INPUT_OUTPUT_DIR + string_file_name;
  new_file.open(string_file_name.c_str());

  while(source_File.good())
  {
                getline(source_File, s_current_line);
                new_file << s_current_line;
                u_line_num++;
                if(u_line_num % U_TAR_FILE_AVERAGE_SIZE == 0)
                {
                        new_file.close();
                        string_file_name = boost::lexical_cast<string>(++i_file_name);
                        string_file_name = INPUT_OUTPUT_DIR + string_file_name;
                        new_file.open(string_file_name.c_str());
                }
        }
        source_File.close();
```

```
	new_file.close();
	return i_file_name;
}
```

8.2. Map-reduce for a word count application

8.2.1. Main.cpp

```
/* FILE: main.cpp
*/

#include<stdio.h>
#include<fstream>
#include<iostream>
#include<string.h>
#include<string>
#include<sstream>
#include<stdlib.h>
#include<unistd.h>
#include<netdb.h>
#include<dirent.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<math.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<sys/wait.h>
#include<arpa/inet.h>
#include<assert.h>
#include<errno.h>
#include<cstdlib>
#include <sys/time.h>
#include<ctime>
#include<pthread.h>
#include <signal.h>
#define bzero(dst, len) (void)memset(dst, 0, len)
#define BACKLOG 200 // how many pending connections queue will hold
#include "manager_stage3.cpp"

using namespace std;

int main(int argc,char **argv)
{
	ifstream infile;
	infile.open(argv[1]);
	char name[256];
	char stage[10];
```

```
        char person[50];
    int i=0;
    int count=0;
    char *pch;

        while(!infile.eof())
        {
                infile.getline(name,256);

                if((*name=='#')   ||   (strlen(name)==0)   ||   (strlen(name)==1))   //check   for
                commented lines and empty spaces.
                {
                        continue;
                }

                else
                {
                        count++;

                        pch = strtok (name," ");
                        if(count==1)
                        {
                                strcpy(stage,name);
                                continue;
                        }

                        if(count==2)
                        {
                                strcpy(person,name);
                                continue;
                        }

                }

        }
        infile.close();
        manager_stage3(argv[1]);
        if((strcmp(stage,"stage3")==0))
        {
                 manager_stage3(argv[1]);
        }

        return 0;

}
```

8.2.2. Manager_stage3.cpp

```
/* FILE: manager_stage3.cpp
  Date last modified: 06/11/2013
*/
#include<stdio.h>
#include<fstream>
#include<iostream>
#include<string.h>
#include<string>
#include<sstream>
#include<stdlib.h>
#include<unistd.h>
#include<netdb.h>
#include<dirent.h>
#include<sys/types.h>
#include<sys/stat.h>
#include <sys/time.h>
#include<math.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<sys/wait.h>
#include<arpa/inet.h>
#include<assert.h>
#include<errno.h>
#include<cstdlib>
#include<ctime>
#include <signal.h>
#define bzero(dst, len) (void)memset(dst, 0, len)
#define BACKLOG 200 // how many pending connections queue will hold
#include "mapreduce.h"
using namespace std;
char the_path[256];



int shuffle(int index)
{
        char cmd[512];
        snprintf(cmd,sizeof(cmd),"sort %s",storage[index]);
        return 0;
}

int manager_stage3(char *argv)
{

        gettimeofday(&start,NULL);

   getcwd(the_path, 255);
```

```
        ifstream infile;
        infile.open(argv);
    char name[50];
        char *pch;
        int numberofworkers,nreducejobs;
    char inputdir[50],outputdir[50],mapperpath[50],reducerpath[50];
    int newi;
        unsigned char isfile =0x8;
        unsigned char isFile =0x8;



        char filepath2[100];//reducer
        char filepath3[100];//stage3.manager.out
        char filepath4[100];//mapper output;
    char filepath5[100];//duplicate for filepath4
        char filepath6[100];//for the input file
        char secondary[100];
        char reducerfile[100];//path for the reduce job
    char mapreduceout[100];
        char forprinting[100];
        char temp[5];
        int i;
        int count=0;
        int count1=0;
        int count2=0;
        int linecount=0;
        int keycount=0;
        int keyvaluecountr=0;
    int keyvaluecountmr=0;
        time_t rawtime,rawtime1,rawtime2,rawtime3,rawtime4,rawtime5,rawtime6,rawtime7;
        char lines[256];



        while(!infile.eof())
        {
                infile.getline(name,256);
                if((*name=='#')    ||    (strlen(name)==0)    ||    (strlen(name)==1))    //check    for
commented lines and empty spaces.
                {
                        continue;
                }
                else
                {
                        count++;

                        pch = strtok (name," ");
                        if(count==3)
                        {
                                pch = strtok (NULL, " ");
                                numberofworkers=atoi(pch);
                                continue;
                        }
                        if(count==4)
                        {
                                pch = strtok (NULL, " ");
                                for(newi=0;pch[newi]!='\0';newi++)
                                mapperpath[newi]=pch[newi];
                                mapperpath[newi]='\0';
                                continue;
                        }

                        if(count==5)
                        {
                                pch = strtok (NULL, " ");
                                for(i=0;pch[i]!='\0';i++)
                                reducerpath[i]=pch[i];
                                reducerpath[i]='\0';
                                continue;
                        }

                        if(count==6)
                        {
                                pch = strtok (NULL, " ");
                                for(i=0;pch[i]!='\0';i++)
                                inputdir[i]=pch[i];
                                inputdir[i]='\0';
                                continue;
                        }
                        if(count==7)
                        {
                                pch = strtok (NULL, " ");
                                for(i=0;pch[i]!='\0';i++)
                                outputdir[i]=pch[i];
                                outputdir[i]='\0';
                                continue;
                        }

                }
        }
infile.close();
```

```
strcpy(filepath,".");
strcat(filepath,"/");
strcat(filepath,mapperpath);
strcpy(filepath2,".");
strcat(filepath2,"/");
strcat(filepath2,reducerpath);


FILE *manageroutpath;
strcpy(filepath3,outputdir);
strcat(filepath3,"stage3.manager.out");
manageroutpath=fopen(filepath3,"w");
time(&rawtime);
fprintf(manageroutpath,"[Prepare Stage] started %s",ctime(&rawtime));

/*Getting the input files */
struct dirent *drnt;
          DIR *dir=NULL;
   dir=opendir("./input");

          if(dir)
      {
            while(drnt = readdir(dir))
            {
                                                 if(drnt->d_type==isfile)
                                                 {
                  strcpy(forprinting,"./input/");
                  strcat(forprinting,drnt->d_name);
                                                    fprintf(manageroutpath,"map        job        m%d,        input
file: %s\n",count1,forprinting);
                                                    count1++;
                                                 }
            }
                                       time(&rawtime1);
                                       fprintf(manageroutpath,"[Prepare                                    Stage]
finished %s\n",ctime(&rawtime1));
            closedir(dir);
      }
      else
      {
            printf("Can not open directory '%s' \n", inputdir);
      }

fclose(manageroutpath);

strcpy(filepath4,".");
strcat(filepath4,"/");
ifstream abcd;
```

```
char nodename[50];
strcpy(nodename,"job");
strcat(filepath4,nodename);
strcpy(filepath5,filepath4);


//Map Stage
manageroutpath=fopen(filepath3,"a+");
time(&rawtime2);
fprintf(manageroutpath,"[Map Stage] started %s",ctime(&rawtime2));
fclose(manageroutpath);


manageroutpath=fopen(filepath3,"a+");
struct dirent *drnt1;
DIR *dir1=NULL;
dir1=opendir("./input");
int status,l;
pid_t pid;
char cmd[512];

          if(dir1)
      {

                                        while(drnt1 = readdir(dir1))
            {

                                                  if(drnt1->d_type==isFile)
                                                  {

                                                            sprintf(temp,"%d",count2);
                                                            strcat(temp,".rj1");
                                                            strcat(filepath5,temp);
                                                            strcpy(filepath6,"./");
                                                            strcat(filepath6,"input/");
                  strcat(filepath6,drnt1->d_name);
                                                            strcpy(filepath,".");
                  strcat(filepath,"/");
                  strcat(filepath,mapperpath);
                  strcpy(storage[count2],filepath5);
                                                            if((pid=fork())==-1)
                  {
                                                                      perror("fork");
                                                                      return -1;
                                                            }
                  else if(pid ==0)
                                                            {
                                                                      //composing the command for execl'ing the
```

```
mapper
                              snprintf(cmd,sizeof(cmd),"%s
< %s > %s","./word_count_mapper.py",filepath6,filepath5);
                                                                          if(-1==execl("/bin/bash", "bash", "-c", cmd, 0))
                                                                          {
                                                                                    perror("execl");
                                                                                    return -1;
                                                                          }
                                                                          else
                                                                          {
                                                                                    assert(0);
                                                                          }
                                                                 }
                                                                 else if(pid>0)
                                                                 {

                                                                          waitpid(pid, &status,0);

                                                                          if(status!=0)
                                                                          {
                                                                                    fprintf(stderr,"error occured while execl
mapper\n");

                                                                                    return -1;
                                                                          }
                                                                          fprintf(manageroutpath,"** map job m%d
completed.\n",count2);


                                                                          //Reading the number of input lines
                                                                          FILE *f1;
                                                                          char c1;

                                                                          f1 = fopen(filepath6, "r");

                                                                          if(f1 == NULL)
                                                                          return 0;

                                                                          while((c1 = fgetc(f1)) != EOF)
                                                                          if(c1 == '\n')
                                                                          linecount++;
                                                                          fprintf(manageroutpath,"     number of input
lines: %d\n",linecount);
                                                                          fclose(f1);


                                                                          //Reading the number of key value pairs

                                                                          //printf("The file path its tryin to open
is %s\n",filepath5);
                                                                          FILE *f2;
                                                                          char c2;

                                                                          f2 = fopen(filepath5, "r");

                                                                          if(f2 == NULL)
                                                                          return 0;

                                                                          while((c2 = fgetc(f2)) != EOF)
                                                                          if(c2 == '\n')
                                                                          keycount++;

                                                                          fclose(f2);

                                                                          fprintf(manageroutpath,"     number   of   output
key/value pairs: %d\n",keycount);
                                                                          count2++;
                                                                          keycount=0;
                                                                          linecount=0;
                                                                          strcpy(filepath5,filepath4);

                                                                 }



                                                           }

                                                     }
                                                     time(&rawtime3);
                                                     fprintf(manageroutpath,"[Map                          Stage]
finished %s\n",ctime(&rawtime3));
            closedir(dir1);
      }
      else
      {
            printf("Can not open directory '%s' \n", inputdir);
      }
                  fclose(manageroutpath);

                  int kk,ll;
                  manageroutpath=fopen(filepath3,"a+");
                  time(&rawtime4);
                  fprintf(manageroutpath,"[Shuffle Stage] Started %s",ctime(&rawtime4));
                  fprintf(manageroutpath," reduce job r0, input files:\n");
                  for(kk=0;kk<count2;kk++)
                  fprintf(manageroutpath,"  %s\n",storage[kk]);
                  fclose(manageroutpath);
```

```
        //To sort and merge the files.
        pid_t pid1;
        for(ll=0;ll<count2;ll++)
        {
                if((pid1=fork())==-1)
                {
                        perror("fork");
                        return -1;
                }
                else if(pid1 ==0)
                {
                        shuffle(ll);
                        exit(0);
                }

                waitpid(pid1, &status,0);
        }
        manageroutpath=fopen(filepath3,"a+");
        time(&rawtime5);
        fprintf(manageroutpath,"[Shuffle Stage] finished %s\n",ctime(&rawtime5));
        fclose(manageroutpath);
        //merging

        pid_t pid2;
        char cmd1[512];
        int lq;
        FILE *reducefile;
        strcpy(reducerfile,".");
        strcat(reducerfile,"/reducerinput.txt");
        reducefile=fopen(reducerfile,"w+");

        for(lq=0;lq<count2;lq++)
        {

                if((pid2=fork())==-1)
                {
                        perror("fork");
                        return -1;
                }
                else if(pid2 ==0)
                {
                snprintf(cmd1,sizeof(cmd1),"sort          %s          %s          -
o %s",reducerfile,storage[lq],reducerfile);
                        if(-1==execl("/bin/bash", "bash", "-c", cmd1, 0))
                        {
                                perror("execl");
                                return -1;
```

```
				}
				else
				{
					assert(0);
				}
				exit(0);
			}
			waitpid(pid2, &status,0);
		}
		fclose(reducefile);

		FILE *mr;
		strcpy(mapreduceout,"./");
		strcat(mapreduceout,"stage3.mapreduce.out");
		mr=fopen(mapreduceout,"w+");
		fclose(mr);

		manageroutpath=fopen(filepath3,"a+");
		time(&rawtime6);
		fprintf(manageroutpath,"[Reduce stage] started %s",ctime(&rawtime6));
		fclose(manageroutpath);

		pid_t pid3;
		char cmd2[512];

		if((pid3=fork())==-1)
		{
			perror("fork");
			return -1;
		}
		else if(pid3 ==0)
		{
			snprintf(cmd2,sizeof(cmd2),"%s
			< %s > %s","./word_count_reducer.py",reducerfile,mapreduceout);
			if(-1==execl("/bin/bash", "bash", "-c", cmd2, 0))
			{
				perror("execl");
				return -1;
			}
			else
			{
				assert(0);
			}
			exit(0);
		}
		waitpid(pid3, &status,0);

		manageroutpath=fopen(filepath3,"a+");
```

```
            fprintf(manageroutpath," ** reduce job r0 completed,\n");
            fclose(manageroutpath);


            //Reading the key value pairs in reducerfile
            FILE *ff1;
            char cc1;
            ff1 = fopen(reducerfile, "r");
            if(ff1 == NULL)
            return 0;
            while((cc1 = fgetc(ff1)) != EOF)
            if(cc1 == '\n')
            keyvaluecountr++;
            fclose(ff1);


            //Reading the key value pairs in mapreducerfile
            char aa1;
            mr = fopen(mapreduceout, "r");
            if(mr == NULL)
            return 0;
            while((aa1 = fgetc(mr)) != EOF)
            if(aa1 == '\n')
            keyvaluecountmr++;

            fclose(mr);

            manageroutpath=fopen(filepath3,"a+");
            fprintf(manageroutpath," number of input key/value pairs: %d\n",keyvaluecountr);
            fprintf(manageroutpath,"numberofoutputkey/value
        pairs: %d\n",keyvaluecountmr);
            time(&rawtime7);
            fprintf(manageroutpath,"[Reduce stage] finished %s\n",ctime(&rawtime7));
            gettimeofday(&end,NULL);
            sec = (int)end.tv_sec - (int)start.tv_sec;
            usec = (int)end.tv_usec - (int)start.tv_usec;

            fprintf(manageroutpath,"Total time used is %d.%u\n",sec,usec);
            fclose(manageroutpath);

            return 0;
}
```

8.3 Mapper and Reducer Codes for a Histogram Application

8.3.1. Histogram.cpp

```
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <ctype.h>
#include <inttypes.h>

#include "map_reduce.h"
#include "stddefines.h"
#include "histogram.h"


bool load_bitmap_header(char *pFileName, BITMAPFILEHEADER* pBmpFileHdr ,
BITMAPINFOHEADER* pBmpInfoHdr)
{
	int i;
	LONG nBitsSize, nBISize;
	//BITMAPFILEHEADER* pBmpFileHdr = new BITMAPFILEHEADER;
	LONG nBytes;
	FILE *fp=fopen(pFileName,"rb");

	if(fp==NULL)return false;

	// Read the file header.  This will tell us the file size and
	// where the data bits start in the file.
	//printf("size of BITMAPFILEHEADER is %d\n", sizeof(BITMAPFILEHEADER));

	//nBytes = fread(&BmpFileHdr, 1, sizeof(BITMAPFILEHEADER), fp);
	//if ( nBytes != sizeof(BITMAPFILEHEADER) )
	fread(&pBmpFileHdr->bfType, 1, 2, fp);
	fread(&pBmpFileHdr->bfSize, 1, 4, fp);
	fread(&pBmpFileHdr->bfReserved1, 1, 2, fp);
	fread(&pBmpFileHdr->bfReserved2, 1, 2, fp);
	fread(&pBmpFileHdr->bfOffBits, 1, 4, fp);

	//printf("FileSize = %x\n", pBmpFileHdr->bfSize);

	// Do we have "BM" at the start indicating a bitmap file?
	if ( pBmpFileHdr->bfType != 0x4D42 )
	{
		fclose(fp);
		return false;
	}

	// Assume for now that the file is a Wondows DIB.
	// Read the BITMAPINFOHEADER.
	//BITMAPINFOHEADER * pBmpInfoHdr = new BITMAPINFOHEADER;
	nBytes = fread(pBmpInfoHdr, 1, sizeof(BITMAPINFOHEADER), fp);
	//printf("size of BITMAPINFOHEADER is %d\n", sizeof(BITMAPINFOHEADER));
	if ( nBytes != 40 )
	{
		fclose(fp);
		return false;
	}

	//only support BGR24 Windows DIB file
	if((pBmpInfoHdr->biBitCount!=24)||(pBmpInfoHdr->biSize                    !=
sizeof(BITMAPINFOHEADER)))
	{
		fclose(fp);
		return false;
	}

	fclose(fp);
	return true;
}


/* valcmp
 * Comparison function
 */
int valcmp(const void *v1, const void *v2)
{
   short value1 = *((short *)v1);
   short value2 = *((short *)v2);

   if (value1 < value2) {
      return -1;
   }
   else if (value1 > value2) {
      return 1;
   }
   else {
      return 0;
   }
}


//hist_merger
void *hist_merger (iterator_t *iter)
```

```
{
 intptr_t sum = 0;
 void *value;

 while (iter_next (iter, &value)) {
  sum = sum + (intptr_t)value;
 }

 return (void *)sum;
}

// hist_mapper()
void hist_mapper(map_args_t *args)
{
  int i;
  WORD *key;
  intptr_t red[256];
  intptr_t green[256];
  intptr_t blue[256];

  BYTE *data = (BYTE *)args->data;

  for(i=0; i< 256; i++) {
   blue[i] = 0;
   green[i] = 0;
   red[i] = 0;
  }


  for (i = 0; i < (args->length) * 3; i+=3)
  {
   blue[data[i]] +=1;;
   green[data[i+1]] +=1;
   red[data[i+2]]+=1;
  }

  for (i = 0; i < 256; i++)
  {

    if (red[i] > 0) {
      key = (WORD *) &(red_keys[i]);
      emit_intermediate((void *)key, (void *)red[i], (int)sizeof(WORD));
    }

    if (green[i] > 0) {
      key = (WORD *) &(green_keys[i]);
      emit_intermediate((void *)key, (void *)green[i], (int)sizeof(WORD));
    }
```

```
        if (blue[i] > 0) {
            key = (WORD *) &(blue_keys[i]);
          emit_intermediate((void *)key, (void *)blue[i], (int)sizeof(WORD));
        }
    }
}

//hist_reducer()
void hist_reducer(void *key_in, iterator_t *iter)
{
  void *value;
  intptr_t sum = 0;
  WORD *key = (WORD *)key_in;


  while (iter_next (iter, &value)) {
    sum = sum + (intptr_t)value;
  }

  emit(key, (void *)sum);
}
```

8.3.2. Main.cpp

```
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <ctype.h>
#include <inttypes.h>
#include <iostream>
#include <ctime>
#include "map_reduce.h"
#include "stddefines.h"
#include "histogram.h"

WORD red_keys[256];
WORD green_keys[256];
WORD blue_keys[256];
```

```
using namespace std;

int main(int argc, char *argv[]) {

   final_data_t hist_vals;
   int i;
   int fd;
   char *pdata;
   //struct stat finfo;
   char * filename;
   BITMAPFILEHEADER BmpFileHdr;
   BITMAPINFOHEADER BmpInfoHdr;
   clock_t start_time, end_time;


   //Record Start time
   start_time = clock();

   // Load Command Line
   if (argc != 2)
   {
      printf("[USAGE]: %s <BMP filename>\n", argv[0]);
      exit(1);
   }

   filename = argv[1];

   cout << "[INFO] Loading BMP File Header from "<< filename <<"!" << endl;
   CHECK_ERROR(load_bitmap_header(filename, &BmpFileHdr, &BmpInfoHdr)==false);


   // Read in the file
   CHECK_ERROR((fd = open(filename, O_RDONLY)) < 0);
   // Get the file info (for file length)
   //CHECK_ERROR(fstat(fd, &finfo) < 0);

   // Memory map the file

   cout << "[INFO] Loading Image Data from "<< filename <<"!" << endl;
   CHECK_ERROR((pdata = (char*)mmap(0, BmpFileHdr.bfSize + 1, PROT_READ | PROT_WRITE,
MAP_PRIVATE, fd, 0)) == NULL);

   int img_byte_size = BmpFileHdr.bfSize - BmpFileHdr.bfOffBits;

   cout << "[INFO] Total Image Byte Size is " << img_byte_size << endl;

   //Initialize Keys
   for (i = 0; i < 256; i++) {
      blue_keys[i] = i;
      green_keys[i] = 1000 + i;
      red_keys[i] = 2000 + i;
   }

   CHECK_ERROR (map_reduce_init ());

   // Configure map reduce parameters
   cout << "[INFO] Configuring Map Reduce Parameters... " << endl;

   map_reduce_args_t map_reduce_args;
   memset(&map_reduce_args, 0, sizeof(map_reduce_args_t));

   map_reduce_args.task_data = &pdata[BmpFileHdr.bfOffBits];    //&hist_data;
   map_reduce_args.data_size = img_byte_size;
   map_reduce_args.result = &hist_vals;


   map_reduce_args.map = hist_mapper;
   map_reduce_args.reduce = hist_reducer;
   map_reduce_args.combiner = hist_merger;
   map_reduce_args.splitter = NULL;
   map_reduce_args.key_cmp = valcmp;

   map_reduce_args.unit_size = 3;  // 3 bytes per pixel
   map_reduce_args.partition = NULL;

   map_reduce_args.L1_cache_size = 32*1024;
   map_reduce_args.num_map_threads = 4;
   map_reduce_args.num_reduce_threads = 2;
   map_reduce_args.num_merge_threads = 1;
   map_reduce_args.num_procs = 2;
   map_reduce_args.key_match_factor = 2;

   cout << "[INFO] Starting Map Reduce For Histogram ... " << endl;

   CHECK_ERROR( map_reduce (&map_reduce_args) < 0);

   // Final Stage
   CHECK_ERROR (map_reduce_finalize ());

   //Record end time
   end_time = clock();
   printf("[INFO] MapReduce Completion Time is %.2f seconds\n", (double)(end_time-
start_time)/CLOCKS_PER_SEC);

   // Print Histogram
   WORD pix_val;
   intptr_t freq;
   WORD prev = 0;
   int pixel_num = img_byte_size/3;
   cout <<"[INFO] Printing Histogram Results:" << endl;
   cout <<"\nBlue\n";
   cout <<"----------\n\n";
   for (i = 0; i < hist_vals.length; i++)
   {
     keyval_t * curr = &((keyval_t *)hist_vals.data)[i];
     pix_val = *((short *)curr->key);
     freq = (intptr_t)curr->val;

     if (pix_val - prev > 700) {
      if (pix_val >= 2000) {
        cout << "\n\nRed\n";
        cout << "----------\n\n";
      }
      else if (pix_val >= 1000) {
        cout << "\n\nGreen\n";
        cout << "----------\n\n";
      }
     }

          int stars = (double)freq / (pixel_num*1.0) * 512.0;

          printf("%3d(%7d) ", pix_val % 1000, freq);
          for(int j=0;j<stars&&j<128; j++) {
           cout << "X";
          }

          cout << endl;

     prev = pix_val;
   }

   CHECK_ERROR (munmap (pdata, BmpFileHdr.bfSize + 1) < 0);
   free(hist_vals.data);
   close (fd);

   return 0;
}
```