Exploration and Innovation of Data Processing by MapReduce

Zhihao Zhang, Rongqian Zhang, Jiayu Li
Syracuse university
{zzhan154, rzhan108, jli221}@syr.edu

Abstract

MapReduce is an algorithm to process and generate large data sets. Users can employ a map function that processes a key and value pair to generate a set of intermediate key and value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.
The purpose of this paper is to explore and present possible scheduling designs and application designs that can be used on or achieved by MapReduce. As MapReduce is a flexible data process mechanism, it is suitable for many popular or new scheduling structures. The paper will present two of the practical structures and discuss the features. It is also interesting about the applications that MapReduce can serve for. The paper demonstrates two simple application types and a new innovative design idea by the authors. All designs and structures are implemented, demonstrated and analyzed. The last part is several general performance evaluations based on different MapReduce configurations.

1. Introduction

1.1 About MapReduce

MapReduce is an algorithm and can process large data sets. MapReduce mainly include two function, map function and reduce function. Basically, a map function processes a key and value pair to generate a set of intermediate key and value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

In a MapReduce program, MapReduce is composed of a Map() method that performs processing and sorting data and a Reduce() that has a summary operation after Map() function. The System adopt the MapReduce design organizes the processing by marshalling the distributed nodes, running the tasks in parallel in different nodes, managing all communications and data transfers between the various parts of the system.

MapReduce allows for distributed processing of the map and reduction operations. Each mapping operation is independent of the others and all maps can be performed in parallel. Similarly, Reduce function will implement in the reduction phase by using results of the map operation sharing the same key at the same time after the finish of map operations. When it comes across partial failure of nodes or storage during the operation, the parallelism also offers some possibility of recover. It will use rescheduled way to recover the data if the input data is available. Now we show how the two important functions work: "Map": Each worker node applies the Map() method to the data the master assigns, and writes the output to a temporary storage. A master node process only one copy of redundant input data. "Reduce": Worker nodes now process each group of output data in parallel.

1.2 Motivation

According some material and what I describe above, the most important part of the MapReduce framework is not the actual map and reduce functions, but the scalability and fault-tolerance. Based on the principle, a single-threaded employing MapReduce algorithm will not be faster than applications without MapReduce. However, multi-threaded implementations of MapReduce algorithm will be faster than both of them.

So it seems that MapReduce is an algorithm for addressing parallel problems with mass by adopting lots of computers, sometimes called nodes. Lots of computers can be organized as a cluster. Processing can occur on data stored in a filesystem or in a database.

Nowadays, we have to face mass data. So we consider how to process such huge data efficiently. According to the mechanism of the MapReduce, programs written in this functional style are automatically parallelized and executed on a large cluster of commodity 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. It has implemented the MapReduce on a large cluster of machines with processing many terabytes of data on thousands of machines. Thus, we may implement the MapReduce on a machine by using multiple process or threads to process big data efficiently because the mechanism of the MapReduce is distributed and paralleled.

1.3 Background

Nowadays, the heart of Apache Hadoop employs the mechanism of MapReduce, which the libraries have been written in many programming languages, with different levels of optimization.

Also, we find Google had done some research in the field of MapReduce. The research implements based on a large cluster of computers: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day. By 2014, Google was no longer using MapReduce as their primary Big Data processing model, and development on Apache Mahout had moved on to more capable and less disk-oriented mechanisms that incorporated full map and reduce capabilities.[1]

2. Overview of System

2.1 About SimGrid

Since employing the mechanism of parallelism, we use the platform, SimGrid, to finish our design. Now, we introduce what SimGrid is because our design is based on SimGrid.

SimGrid is a toolkit that provides core functionalities for the simulation of distributed applications in heterogeneous distributed environments. The specific goal of the project is to facilitate research in the area of parallel and distributed large scale systems, such as Grids, P2P systems and Cloud. Its use cases encompass heuristic evaluation, application prototyping or even real application development and tuning.[2]

2.2 An Example of MapReduce

In this part, we use an example to explain how the MapReduce works. Assuming that we have an original table which will be processed by master using MapReduce mechanism. Now, the original table like following table 1:

Key	Count
Cat	1
Cat	1
…	…
Dog	1
…	…
Girl	1

Table 1 Original table

The figure 1 show the most important procedure which express how the MapReduce works. According to the mechanism of MapReduce, we send the table 1 to the master. If the master receive the table 1, it will separate the table 1 into n parts (table 3 shows the one of the parts). We call each part sub-table. Then the master will send the sub-tables to workers.

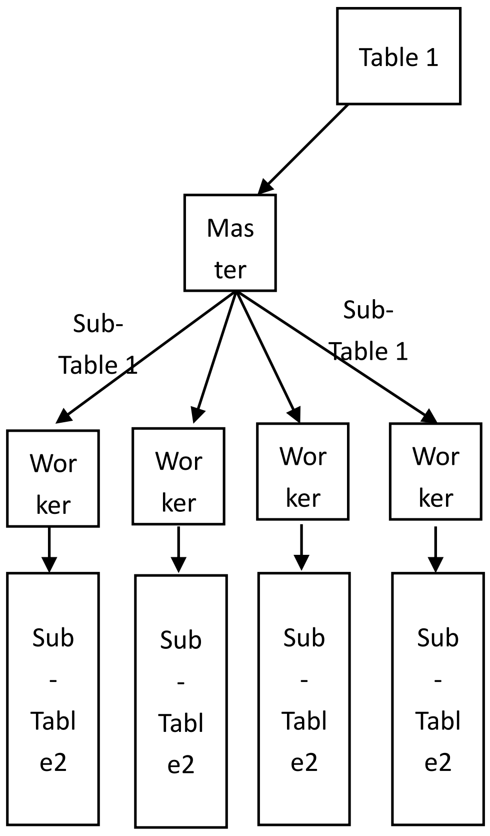


Figure 1

After getting the information from master, the workers will process sub-tables in parallel. Each worker will reduce the sub-table they receive to a new sub-table. Table 3 shows a result of finishing reducing in a worker.

Key	Count
Cat	1
Cat	1
Girl	1
Girl	1

Table 2 one of the sub-Table 1

Key	Count
Cat	2
Girl	2

Table 3 sub-Table 2

When finishing reduce workers will send results back to the master and master will merge the results. Finally, the master outputs the result to a user.

2.3 Purpose

(1) Design and realize MapReduce structures on our own.

Our first goal is to achieve MapReduce designs on our own. We take the general idea of MapReduce, however we wanted to brainstorm and create the practical structures by our own. All designs we built are from our own ideas. And in some way, these ideas are new.

(2) Design different scheduling method for workers

Scheduling is crucial in MapReduce to increase efficiency. However there are many different scheduling ideas. We tried to discuss some of the different scheduling structures by implementing different scheduling designs.

(3) Implemented simple applications to discover the modularity of MapReduce

MapReduce is used to work on real world data processing, so we would like to explore simple applications suitable for MapReduce mechanism.

(4) Innovate and implement a multilevel sorting mechanism by MapReduce

Innovation by brainstorming on the MapReduce feature and mechanism. We try to make a large scale data sorting using the benefit of distributed systems.

The mechanism:

(1) Master receive the input file. Split the file according to the number of workers.

(2) Send the split file pieces to the workers for mapping.

(3) Workers map the files. Then send the files back to master.

(4) Master merge the mapped files. Then split the file again for reduce.

(5) Send the split file pieces to the workers for reducing.

(6) Workers reduce the files. Then send the files back to master.

(7) Master merge the reduced files. Output the final file.

This mechanism has the following features:

(1) Files are split into numbers of the workers. The workers only map and reduce once.

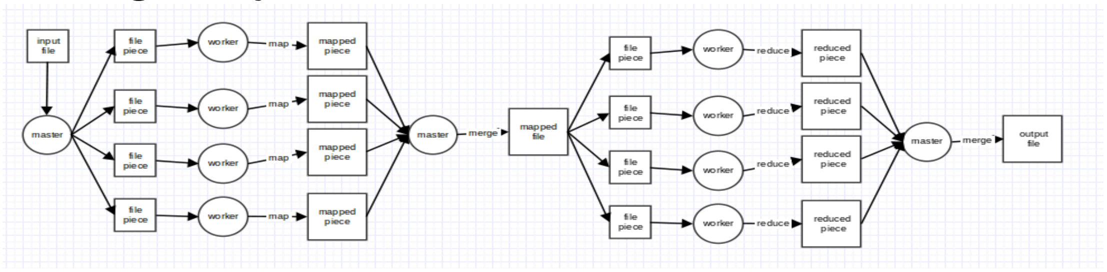
(2) Very stable work flow. All working flow behaviors are fixed before the execution.

(3) Very poor flexibility. Cannot adjust any flow behaviors. No communication between machines.

(4) Poor performance due to no scheduling and “dumb” parallelization.

Thus this is not a feasible solution for a proper MapReduce design. We abandoned this design in the progress.

3. Design of System



3.1 MapReduce Structure Design

This design is basic is simply because it only uses the fundamental ideas of MapReduce.

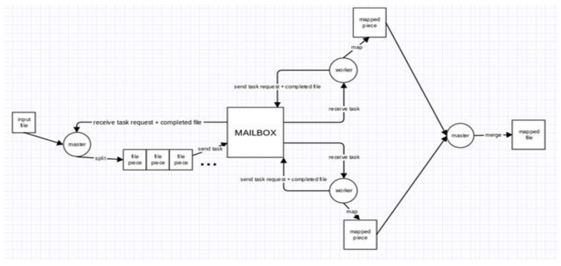
3.2 Improved MapReduce Structure Design with Scheduling

To build an improved MapReduce structure with scheduling, we adapted a functionality called a “mailbox”, which is a provided function in SimGrid.

The mailbox is a feature for mutual communication between master and workers. Any of the alive machine can send mails (including data) to the mailbox, with an “address” on it indicating who should receive this mail. Then the receiver will call a “receive” function and get the mail (along with the data).

The general idea of this design, is to use this mailbox to communicate between the master and the workers, so that there can be a dynamic control of the flow with possible adjustments.

The master holds the data for processing, and send the data with the form of a task to workers. After finishing the task, the worker can send master a mail to get a new task.



Map stage:

As shown in the picture, the map mechanism is:

(1) Master split the files according to the user configuration.

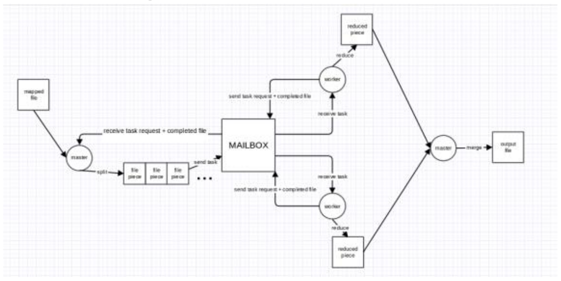
(2) Master send several of the pieces to workers to process.

(3) If any worker finished the task, the worker send a new task request to master along with the processed file.

(4) The master receive the request along with the file. It merge the file and decide whether to send a new task to this worker.

(5) Loop until finish.

Reduce stage:



The reduce stage is almost the same as the map stage.

The feature of this design:

(1) Flexible flow adjustments. It can now choose the number and the size of the file pieces freely. This provides the space to find the best performance settings.

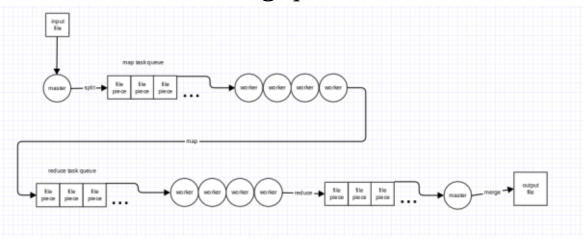
(2) Communication. This provides a dynamic control for the master. This can be feasible for fault tolerance and performance rescheduling. The master can control the task flows by evaluating the performance of the workers.

(3) More flexibility. Can be used for more complex and intelligent workflow.

(4) Needs very careful cooperation algorithm.

3.3 Another Fast Scheduling Model Using Working Queues

This design is a simpler design for the MapReduce scheduling. It uses the feature of a working queue.



Mechanism:

(1) Master split the file into pieces.

(2) Master put the pieces into a working queue as tasks for mapping.

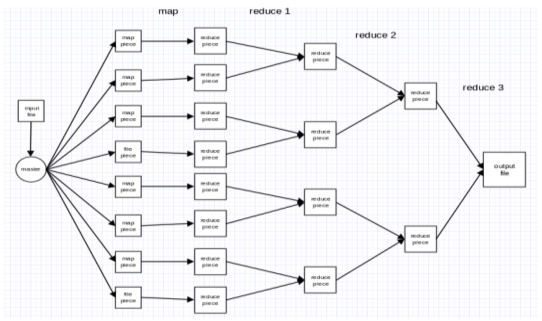
(3) Workers fetch the tasks in lines to map.

(4) Finished workers put the mapped files

into the reduce queue for reducing.
(5) Other workers do the reduce job from the reduce queue.
(6) Finish when all mapping and reducing jobs are done.
(7) Master merge the files along the way.

This design has the following features:
(1) Very efficient and fluent workflow. No any step needs a barrier, which means every worker can be kept busy until the end.
(2) No much communication. This is a one-way working flow, so there is no communication between master and workers. Which reduced the flexibility and total control.
(3) Easy to get deadlock! As the working flow is totally dynamic, any blocking along the way will stop all progress. Any mis-design can cause serious deadlock.
(4) Fatal problem: can only be used for “non-middle barrier” applications. In other words, if an application needs a merging at the middle of the progress, this design won't work. It cannot have any universal blocking at the middle of the progress. In fact it can. However if we add the barrier to this design, it will lose its advantages of being fast and dynamic.

3.4 Innovation: Multilevel MapReduce sorting design:



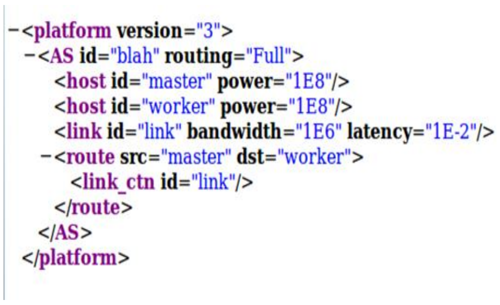
This is an application design, not an workflow design. In this design, we innovated a general purpose application for MapReduce. This is a design of a sorting mechanism.
In general, this structure is a merge sort in a distributed system scale. However it can be also seen as a “hyper sort” where we divide the list into small pieces and sort the small pieces by quick sort. Then we merge sort these small pieces into on big list. The merge sort progress works just like a normal merge sort.
What's new:
(1) Parallel sorting: using distributed system to sort, much more efficient.
(2) File based sorting: Sort the list based on file records. This makes it possible to process very long list at a moment. Regardless of the limitation of machine memory sizes.
(3) Multilevel reduce. Repeatedly reduce (merge) the files until all merged.

4. Implementation of System

4.1 Setting up Environment:

We build our design and applications on the simulation platform of SimGrid. Thus the first thing to do is to set up the environment properly.
We implemented our design in C. Actually SimGrid supports Java and Lua too and Java or Lua would be much better than C for a complicated design. However, SimGrid supports C better, so we still chose C for this implementation.
SimGrid works as a C library. To setup the simulation on top of SimGrid, we need to provide a “SimGrid style” main function and two environment configuration XML files.
Then the specific behavior should be implemented in the master.c and worker.c files.

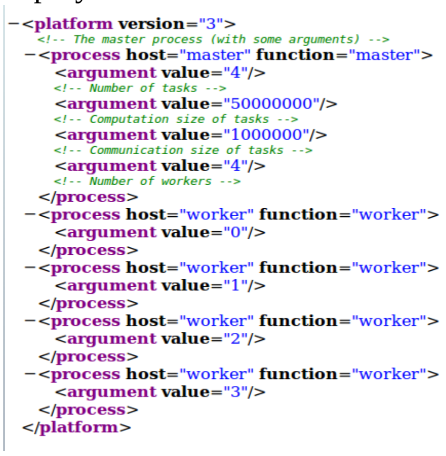
platform.xml:



```
-<platform version="3">
  -<AS id="blah" routing="Full">
     <host id="master" power="1E8"/>
     <host id="worker" power="1E8"/>
     <link id="link" bandwidth="1E6" latency="1E-2"/>
    -<route src="master" dst="worker">
       <link_ctn id="link"/>
     </route>
   </AS>
 </platform>
```

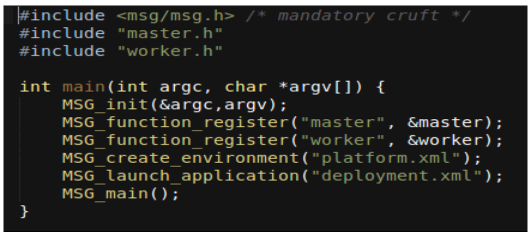
This file defines the basic environment set up for the process. Here we defined two character types, master and worker. Then we define a "link" and a "routine" to define a communication path between master and worker. This can be seen as the "hardware" setup in real world distributed systems.

deployment.xml:



```
-<platform version="3">
    <!-- The master process (with some arguments) -->
  -<process host="master" function="master">
     <argument value="4"/>
     <!-- Number of tasks -->
     <argument value="50000000"/>
     <!-- Computation size of tasks -->
     <argument value="1000000"/>
     <!-- Communication size of tasks -->
     <argument value="4"/>
     <!-- Number of workers -->
   </process>
  -<process host="worker" function="worker">
     <argument value="0"/>
   </process>
  -<process host="worker" function="worker">
     <argument value="1"/>
   </process>
  -<process host="worker" function="worker">
     <argument value="2"/>
   </process>
  -<process host="worker" function="worker">
     <argument value="3"/>
   </process>
 </platform>
```

This is the specific machine set up. Here we set up one master and four workers for operation. Then the main function should look like:



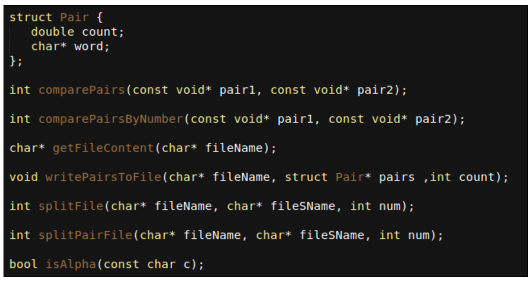
```
#include <msg/msg.h> /* mandatory cruft */
#include "master.h"
#include "worker.h"

int main(int argc, char *argv[]) {
    MSG_init(&argc,argv);
    MSG_function_register("master", &master);
    MSG_function_register("worker", &worker);
    MSG_create_environment("platform.xml");
    MSG_launch_application("deployment.xml");
    MSG_main();
}
```

This is the way of linking all configuration files and source files to combine into the simulation.

4.2 Mailbox Based Scheduling Design

This design we uses the simple application for "finding all word count in an article" to demonstrate the feature. The function is to count different words inside an article, and output a file with the list of numbers of appearing words in this article.
MapReduce first fetch all single words in the article, map them into pairs of "word 1", and write all these pairs into mapped files.
Then the master merge the mapped files, then send pieces to reducers.
Reducers merge the same words from the list and add up all numbers to form the total count.

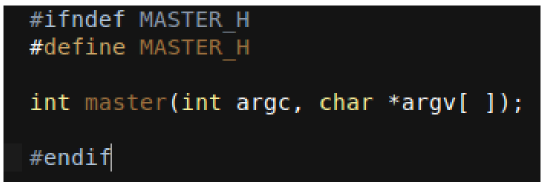
A basic library is written to keep all basic functions and structure for usage. It is called "basic.h".
Content of basic.h:



```
struct Pair {
    double count;
    char* word;
};

int comparePairs(const void* pair1, const void* pair2);

int comparePairsByNumber(const void* pair1, const void* pair2);

char* getFileContent(char* fileName);

void writePairsToFile(char* fileName, struct Pair* pairs ,int count);

int splitFile(char* fileName, char* fileSName, int num);

int splitPairFile(char* fileName, char* fileSName, int num);

bool isAlpha(const char c);
```

Here the struct "Pair" is used to store the pairs mapped by mapper.

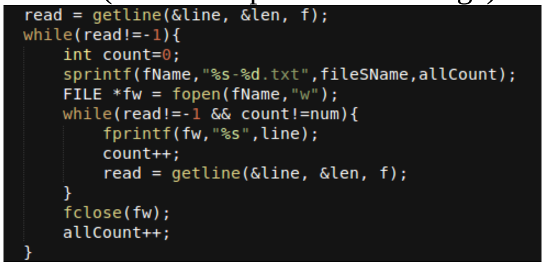
Implementation of master:
Simple declaration of master in master.h:



```
#ifndef MASTER_H
#define MASTER_H

int master(int argc, char *argv[ ]);

#endif
```

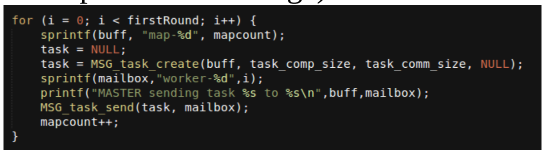
The definition for master can be divided into parts, according to different behaviors in different steps:

Split the input file by splitFile function in basic.h (both in map and reduce stage):



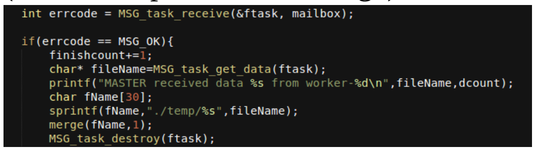
```
read = getline(&line, &len, f);
while(read!=-1){
    int count=0;
    sprintf(fName,"%s-%d.txt",fileSName,allCount);
    FILE *fw = fopen(fName,"w");
    while(read!=-1 && count!=num){
        fprintf(fw,"%s",line);
        count++;
        read = getline(&line, &len, f);
    }
    fclose(fw);
    allCount++;
}
```

Send the first set of tasks to workers (both in map and reduce stage):



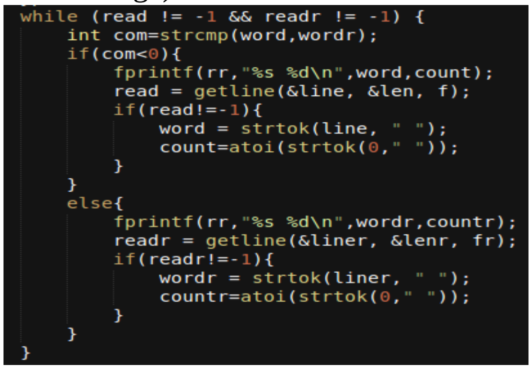
```
for (i = 0; i < firstRound; i++) {
    sprintf(buff, "map-%d", mapcount);
    task = NULL;
    task = MSG_task_create(buff, task_comp_size, task_comm_size, NULL);
    sprintf(mailbox,"worker-%d",i);
    printf("MASTER sending task %s to %s\n",buff,mailbox);
    MSG_task_send(task, mailbox);
    mapcount++;
}
```

Waiting and receiving completed files (both in map and reduce stage):



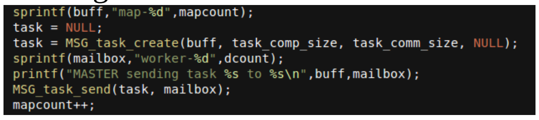
```
int errcode = MSG_task_receive(&ftask, mailbox);

if(errcode == MSG_OK){
    finishcount+=1;
    char* fileName=MSG_task_get_data(ftask);
    printf("MASTER received data %s from worker-%d\n",fileName,dcount);
    char fName[30];
    sprintf(fName,"./temp/%s",fileName);
    merge(fName,1);
    MSG_task_destroy(ftask);
```

Merge the file pieces (both in map and reduce stage):



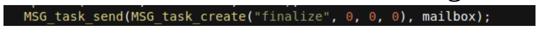
```
while (read != -1 && readr != -1) {
    int com=strcmp(word,wordr);
    if(com<0){
        fprintf(rr,"%s %d\n",word,count);
        read = getline(&line, &len, f);
        if(read!=-1){
            word = strtok(line, " ");
            count=atoi(strtok(0," "));
        }
    }
    else{
        fprintf(rr,"%s %d\n",wordr,countr);
        readr = getline(&liner, &lenr, fr);
        if(readr!=-1){
            wordr = strtok(liner, " ");
            countr=atoi(strtok(0," "));
        }
    }
}
```

Sending new tasks to finished workers:



```
sprintf(buff,"map-%d",mapcount);
task = NULL;
task = MSG_task_create(buff, task_comp_size, task_comm_size, NULL);
sprintf(mailbox,"worker-%d",dcount);
printf("MASTER sending task %s to %s\n",buff,mailbox);
MSG_task_send(task, mailbox);
mapcount++;
```

Send message to terminate all workers if no more new tasks after reduce stage:

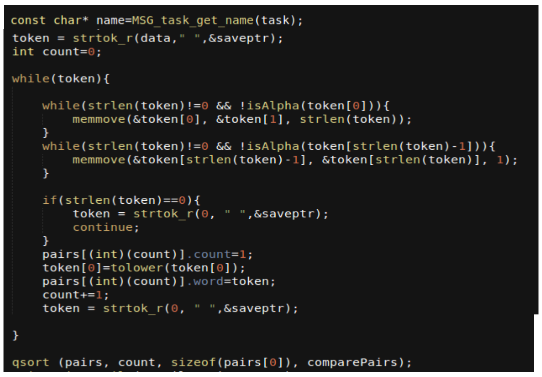


```
MSG_task_send(MSG_task_create("finalize", 0, 0, 0), mailbox);
```

Implementation of worker:

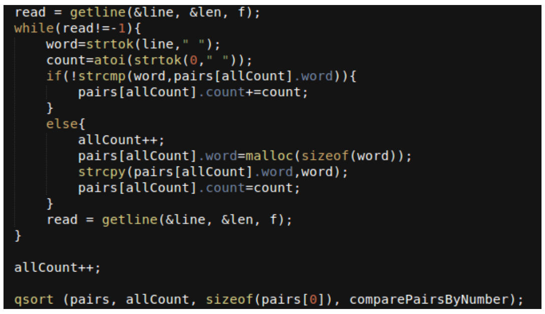
Receive tasks from master and work on it according to the task type (map or reduce), then send the file back to master:

Map function:



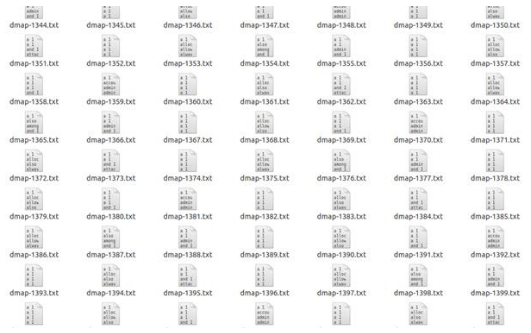
```
const char* name=MSG_task_get_name(task);
token = strtok_r(data," ",&saveptr);
int count=0;

while(token){

    while(strlen(token)!=0 && !isAlpha(token[0])){
        memmove(&token[0], &token[1], strlen(token));
    }
    while(strlen(token)!=0 && !isAlpha(token[strlen(token)-1])){
        memmove(&token[strlen(token)-1], &token[strlen(token)], 1);
    }

    if(strlen(token)==0){
        token = strtok_r(0, " ",&saveptr);
        continue;
    }
    pairs[(int)(count)].count=1;
    token[0]=tolower(token[0]);
    pairs[(int)(count)].word=token;
    count+=1;
    token = strtok_r(0, " ",&saveptr);

}

qsort (pairs, count, sizeof(pairs[0]), comparePairs);
```

Reduce function:

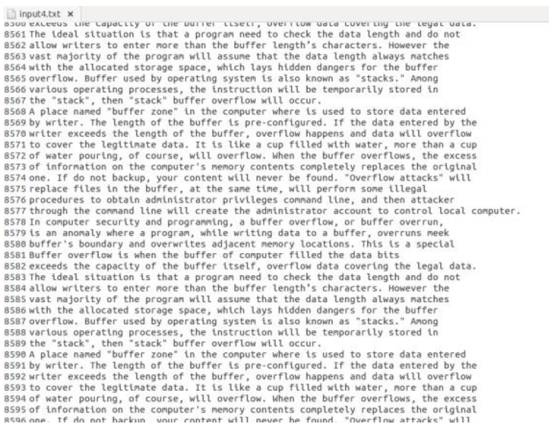


```
read = getline(&line, &len, f);
while(read!=-1){
    word=strtok(line," ");
    count=atoi(strtok(0," "));
    if(!strcmp(word,pairs[allCount].word)){
        pairs[allCount].count+=count;
    }
    else{
        allCount++;
        pairs[allCount].word=malloc(sizeof(word));
        strcpy(pairs[allCount].word,word);
        pairs[allCount].count=count;
    }
    read = getline(&line, &len, f);
}

allCount++;

qsort (pairs, allCount, sizeof(pairs[0]), comparePairsByNumber);
```

The temporary middle files are stored in the folder "temp". The file pieces are named with numbers, file pieces ready to map are "map-(number).txt" and the file pieces after mapping is "dmap-(number).txt". The file pieces ready to reduce are "reduce-(number).txt" and the file pieces after reducing is "dreduce-(number).txt". Middle merged file is "mid.txt" and the final output is "result.txt" which is not in the temp folder but in the main directory.

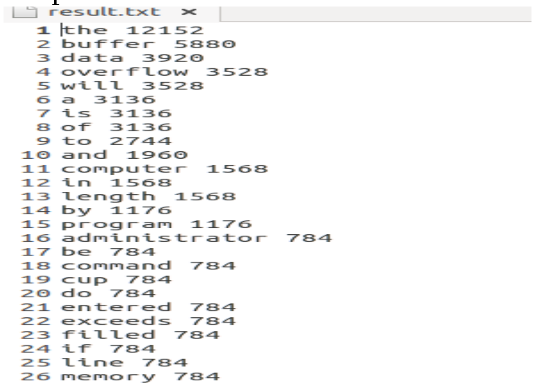


Sample input file:



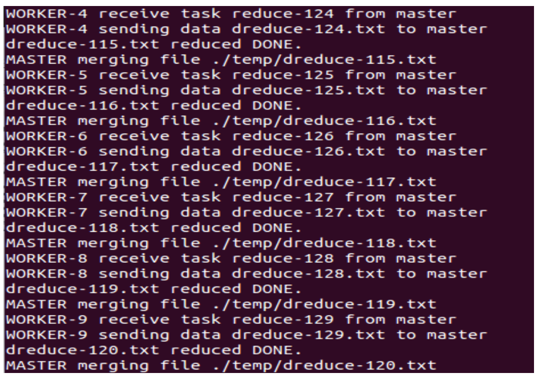
```
exceeds the capacity of the buffer itself, overflow data covering the legal data.
The ideal situation is that a program need to check the data length and do not
allow writers to enter more than the buffer length's characters. However the
vast majority of the program will assume that the data length always matches
with the allocated storage space, which lays hidden dangers for the buffer
overflow. Buffer used by operating system is also known as "stacks." Among
various operating processes, the instruction will be temporarily stored in
the "stack", then "stack" buffer overflow will occur.
A place named "buffer zone" in the computer where is used to store data entered
by writer. The length of the buffer is pre-configured. If the data entered by the
writer exceeds the length of the buffer, overflow happens and data will overflow
to cover the legitimate data. It is like a cup filled with water, more than a cup
of water pouring, of course, will overflow. When the buffer overflows, the excess
of information on the computer's memory contents completely replaces the original
one. If do not backup, your content will never be found. "Overflow attacks" will
replace files in the buffer, at the same time, will perform some illegal
procedures to obtain administrator privileges command line, and then attacker
through the command line will create the administrator account to control local computer.
In computer security and programming, a buffer overflow, or buffer overrun,
is an anomaly where a program, while writing data to a buffer, overruns meek
buffer's boundary and overwrites adjacent memory locations. This is a special
Buffer overflow is when the buffer of computer filled the data bits
exceeds the capacity of the buffer itself, overflow data covering the legal data.
The ideal situation is that a program need to check the data length and do not
allow writers to enter more than the buffer length's characters. However the
vast majority of the program will assume that the data length always matches
with the allocated storage space, which lays hidden dangers for the buffer
overflow. Buffer used by operating system is also known as "stacks." Among
various operating processes, the instruction will be temporarily stored in
the "stack", then "stack" buffer overflow will occur.
A place named "buffer zone" in the computer where is used to store data entered
by writer. The length of the buffer is pre-configured. If the data entered by the
writer exceeds the length of the buffer, overflow happens and data will overflow
to cover the legitimate data. It is like a cup filled with water, more than a cup
of water pouring, of course, will overflow. When the buffer overflows, the excess
of information on the computer's memory contents completely replaces the original
one. If do not backup, your content will never be found. "Overflow attacks" will
```

Output:

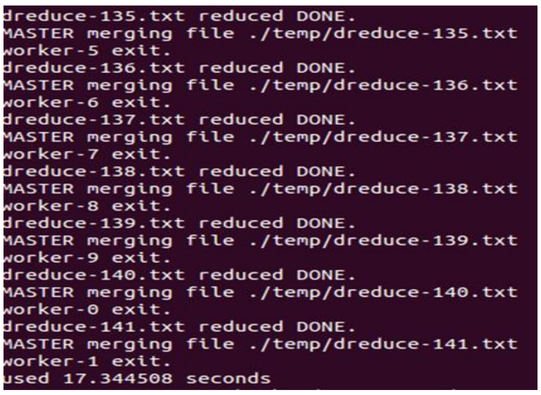


```
the  12152
buffer  5880
data  3920
overflow  3528
will  3528
a  3136
is  3136
of  3136
to  2744
and  1960
computer  1568
in  1568
length  1568
by  1176
program  1176
administrator  784
be  784
command  784
cup  784
do  784
entered  784
exceeds  784
filled  784
if  784
line  784
memory  784
```

Process log:



```
WORKER-4 receive task reduce-124 from master
WORKER-4 sending data dreduce-124.txt to master
dreduce-115.txt reduced DONE.
MASTER merging file ./temp/dreduce-115.txt
WORKER-5 receive task reduce-125 from master
WORKER-5 sending data dreduce-125.txt to master
dreduce-116.txt reduced DONE.
MASTER merging file ./temp/dreduce-116.txt
WORKER-6 receive task reduce-126 from master
WORKER-6 sending data dreduce-126.txt to master
dreduce-117.txt reduced DONE.
MASTER merging file ./temp/dreduce-117.txt
WORKER-7 receive task reduce-127 from master
WORKER-7 sending data dreduce-127.txt to master
dreduce-118.txt reduced DONE.
MASTER merging file ./temp/dreduce-118.txt
WORKER-8 receive task reduce-128 from master
WORKER-8 sending data dreduce-128.txt to master
dreduce-119.txt reduced DONE.
MASTER merging file ./temp/dreduce-119.txt
WORKER-9 receive task reduce-129 from master
WORKER-9 sending data dreduce-129.txt to master
dreduce-120.txt reduced DONE.
MASTER merging file ./temp/dreduce-120.txt
```



```
dreduce-135.txt reduced DONE.
MASTER merging file ./temp/dreduce-135.txt
worker-5 exit.
dreduce-136.txt reduced DONE.
MASTER merging file ./temp/dreduce-136.txt
worker-6 exit.
dreduce-137.txt reduced DONE.
MASTER merging file ./temp/dreduce-137.txt
worker-7 exit.
dreduce-138.txt reduced DONE.
MASTER merging file ./temp/dreduce-138.txt
worker-8 exit.
dreduce-139.txt reduced DONE.
MASTER merging file ./temp/dreduce-139.txt
worker-9 exit.
dreduce-140.txt reduced DONE.
MASTER merging file ./temp/dreduce-140.txt
worker-0 exit.
dreduce-141.txt reduced DONE.
MASTER merging file ./temp/dreduce-141.txt
worker-1 exit.
used 17.344508 seconds
```

Discussion on this implementation:

(1) Sending data in mails:

Our first approach is to send the real data by mails through mailbox to achieve a design of using machine memory to process, send and receive data. This is actually a supported feature in the mailbox functionality. However in practice it causes a lot of memory failure. In some way the receiver cannot receive the data, or the received data is corrupted. As the data is sent by pointers, it is easy to encounter “scale issues”. Also considering that using memory to process data is generally not practical, so we abandoned this approach.

(2) Memory failures:

In C, trying to create dynamic data needs a lot of memory operations. The annoying part is that, these operation needs to be very precise. Any mistake in allocating a memory may cause failure. And this failure can happen at anywhere, not necessary to be the place where you make the mistake. Because a memory corruption may not be detected by the time you make the wrong move, debugging can be confusing as you don't know where to look at.

(3) File system management

This design needs a in program control of the files on the disk. That means, the program must have the ability to create, open, close, read, write, delete, move or rename the files. These features are not all supported by Windows, so this implementation is based on Linux glibc.

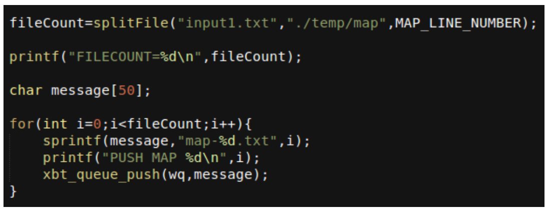
4.3 Multilevel MapReduce Sorting by Workqueue Based Scheduling

This is a implementation combination of the application design of “multilevel MapReduce sorting” by the scheduling design of workqueue. It uses the working queue workflow to realize the multilevel sorting by MapReduce introduced in the “Design of the System” part.

The general environment is the same as before. The modification is on the behavior of master and worker, along with the map and reduce functions.
The input is an article, and the output would be the sorted list of all the words.

Implementation of master:

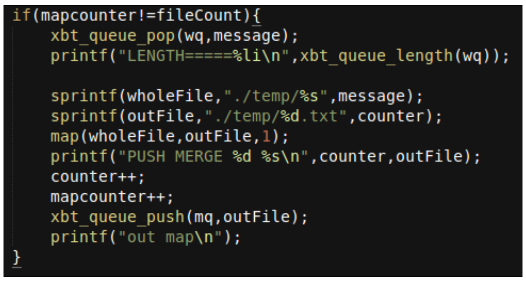
This time, master has little things to do. It only need to split the file and send the map tasks to the map queue. Then it's job is done.



```
fileCount=splitFile("input1.txt","./temp/map",MAP_LINE_NUMBER);

printf("FILECOUNT=%d\n",fileCount);

char message[50];

for(int i=0;i<fileCount;i++){
    sprintf(message,"map-%d.txt",i);
    printf("PUSH MAP %d\n",i);
    xbt_queue_push(wq,message);
}
```

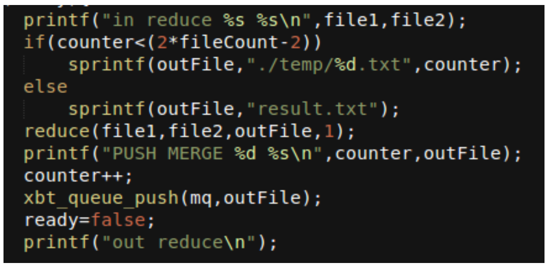
Implementation of worker:

This time the worker need to be careful and has many jobs.
Map phase:



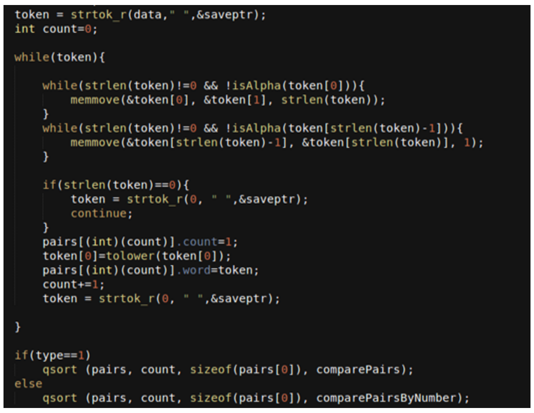
```
if(mapcounter!=fileCount){
    xbt_queue_pop(wq,message);
    printf("LENGTH=====%li\n",xbt_queue_length(wq));

    sprintf(wholeFile,"./temp/%s",message);
    sprintf(outFile,"./temp/%d.txt",counter);
    map(wholeFile,outFile,1);
    printf("PUSH MERGE %d %s\n",counter,outFile);
    counter++;
    mapcounter++;
    xbt_queue_push(mq,outFile);
    printf("out map\n");
}
```

Reduce phase:



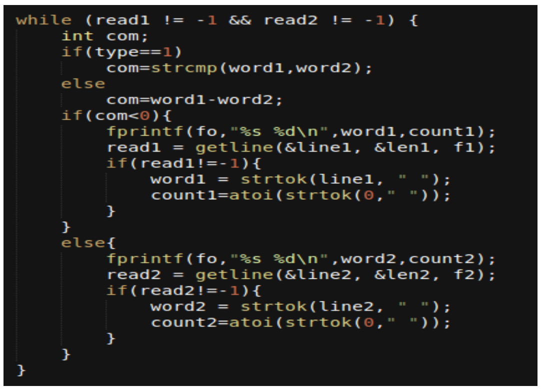
```
printf("in reduce %s %s\n",file1,file2);
if(counter<(2*fileCount-2))
    sprintf(outFile,"./temp/%d.txt",counter);
else
    sprintf(outFile,"result.txt");
reduce(file1,file2,outFile,1);
printf("PUSH MERGE %d %s\n",counter,outFile);
counter++;
xbt_queue_push(mq,outFile);
ready=false;
printf("out reduce\n");
```

Map function:



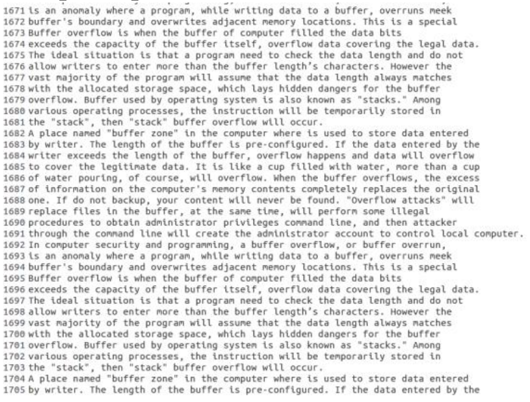
```
token = strtok_r(data," ",&saveptr);
int count=0;

while(token){

    while(strlen(token)!=0 && !isAlpha(token[0])){
        memmove(&token[0], &token[1], strlen(token));
    }
    while(strlen(token)!=0 && !isAlpha(token[strlen(token)-1])){
        memmove(&token[strlen(token)-1], &token[strlen(token)], 1);
    }

    if(strlen(token)==0){
        token = strtok_r(0, " ",&saveptr);
        continue;
    }
    pairs[(int)(count)].count=1;
    token[0]=tolower(token[0]);
    pairs[(int)(count)].word=token;
    count+=1;
    token = strtok_r(0, " ",&saveptr);

}

if(type==1)
    qsort (pairs, count, sizeof(pairs[0]), comparePairs);
else
    qsort (pairs, count, sizeof(pairs[0]), comparePairsByNumber);
```

Reduce function:



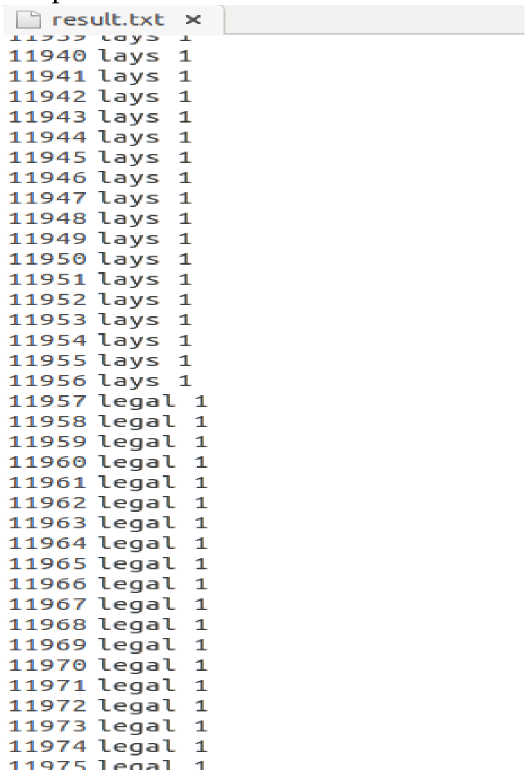
```
while (read1 != -1 && read2 != -1) {
    int com;
    if(type==1)
        com=strcmp(word1,word2);
    else
        com=word1-word2;
    if(com<0){
        fprintf(fo,"%s %d\n",word1,count1);
        read1 = getline(&line1, &len1, f1);
        if(read1!=-1){
            word1 = strtok(line1, " ");
            count1=atoi(strtok(0," "));
        }
    }
    else{
        fprintf(fo,"%s %d\n",word2,count2);
        read2 = getline(&line2, &len2, f2);
        if(read2!=-1){
            word2 = strtok(line2, " ");
            count2=atoi(strtok(0," "));
        }
    }
}
```

Sample input file:



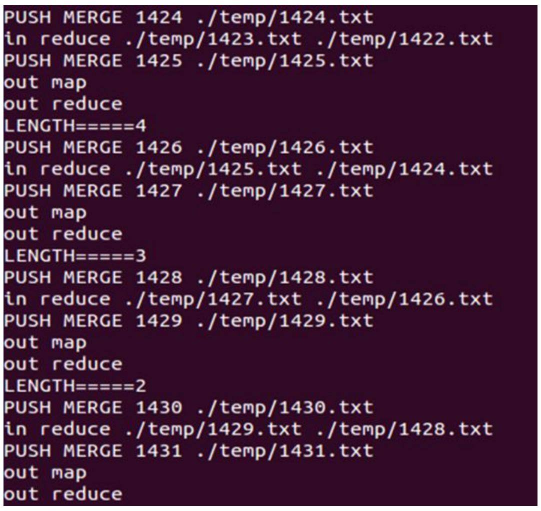
```
is an anomaly where a program, while writing data to a buffer, overruns meek
buffer's boundary and overwrites adjacent memory locations. This is a special
Buffer overflow is when the buffer of computer filled the data bits
exceeds the capacity of the buffer itself, overflow data covering the legal data.
The ideal situation is that a program need to check the data length and do not
allow writers to enter more than the buffer length’s characters. However the
vast majority of the program will assume that the data length always matches
with the allocated storage space, which lays hidden dangers for the buffer
overflow. Buffer used by operating system is also known as "stacks." Among
various operating processes, the instruction will be temporarily stored in
the "stack", then "stack" buffer overflow will occur.
A place named "buffer zone" in the computer where is used to store data entered
by writer. The length of the buffer is pre-configured. If the data entered by the
writer exceeds the length of the buffer, overflow happens and data will overflow
to cover the legitimate data. It is like a cup filled with water, more than a cup
of water pouring, of course, will overflow. When the buffer overflows, the excess
of information on the computer's memory contents completely replaces the original
one. If do not backup, your content will never be found. "Overflow attacks" will
replace files in the buffer, at the same time, will perform some illegal
procedures to obtain administrator privileges command line, and then attacker
through the command line will create the administrator account to control local computer.
In computer security and programming, a buffer overflow, or buffer overrun,
is an anomaly where a program, while writing data to a buffer, overruns meek
buffer's boundary and overwrites adjacent memory locations. This is a special
Buffer overflow is when the buffer of computer filled the data bits
exceeds the capacity of the buffer itself, overflow data covering the legal data.
The ideal situation is that a program need to check the data length and do not
allow writers to enter more than the buffer length’s characters. However the
vast majority of the program will assume that the data length always matches
with the allocated storage space, which lays hidden dangers for the buffer
overflow. Buffer used by operating system is also known as "stacks." Among
various operating processes, the instruction will be temporarily stored in
the "stack", then "stack" buffer overflow will occur.
A place named "buffer zone" in the computer where is used to store data entered
by writer. The length of the buffer is pre-configured. If the data entered by the
```

Output file:



```
result.txt
11940 lays 1
11941 lays 1
11942 lays 1
11943 lays 1
11944 lays 1
11945 lays 1
11946 lays 1
11947 lays 1
11948 lays 1
11949 lays 1
11950 lays 1
11951 lays 1
11952 lays 1
11953 lays 1
11954 lays 1
11955 lays 1
11956 lays 1
11957 legal 1
11958 legal 1
11959 legal 1
11960 legal 1
11961 legal 1
11962 legal 1
11963 legal 1
11964 legal 1
11965 legal 1
11966 legal 1
11967 legal 1
11968 legal 1
11969 legal 1
11970 legal 1
11971 legal 1
11972 legal 1
11973 legal 1
11974 legal 1
11975 legal 1
```

Process log:



```
PUSH MERGE 1424 ./temp/1424.txt
in reduce ./temp/1423.txt ./temp/1422.txt
PUSH MERGE 1425 ./temp/1425.txt
out map
out reduce
LENGTH=====4
PUSH MERGE 1426 ./temp/1426.txt
in reduce ./temp/1425.txt ./temp/1424.txt
PUSH MERGE 1427 ./temp/1427.txt
out map
out reduce
LENGTH=====3
PUSH MERGE 1428 ./temp/1428.txt
in reduce ./temp/1427.txt ./temp/1426.txt
PUSH MERGE 1429 ./temp/1429.txt
out map
out reduce
LENGTH=====2
PUSH MERGE 1430 ./temp/1430.txt
in reduce ./temp/1429.txt ./temp/1428.txt
PUSH MERGE 1431 ./temp/1431.txt
out map
out reduce
```

Discussion on this implementation:

(1) Deadlock:

Working queues are easy to get into deadlock. We even put
two proceeding queues in this design. Thus we made a lot of modifications to avoid deadlock. The current mechanism is like this:

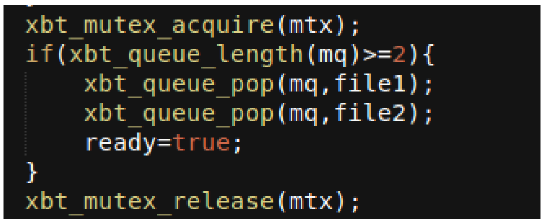
We uses a safe queue provided by SimGrid called “xbt_queue”. Anyone who pushes into this queue will be blocked if the queue is full. Anyone who pops from this queue will be block if the queue is empty.

Think of this scenario: we split the files into 100 pieces, the map queue is of 20 capacity and the reduce queue is of 20 capacity. Every mapping of the file generates a reduce-ready file and pushed into the reduce queue. However no one is popping from reduce queue because all workers are busy mapping files from the map queue. Thus very soon the reduce queue is full and the next push is blocked. All workers are all blocked by the reduce queue and no one is popping from the reduce queue. DEADLOCK.

Thus to make the workflow out of blocking, the reduce queue is prior to the map queue. In a round, the worker checks first if there is a reduce task in the reduce queue, if so, do the reduce task. If the reduce queue is empty, then go back to do the map task.

(2) Popping from reduce queue twice in a row:

The content of the reduce queue is the files needs to be merged. Thus for a success reduce job, the worker need to pop two files from the reduce queue at one time. As this is a parallel work, this may cause problem. What if there are only two files left in the reduce queue, and two workers each fetched one? Another deadlock. To make sure one worker can fetch twice at a time, a mutex is used:



```
xbt_mutex_acquire(mtx);
if(xbt_queue_length(mq)>=2){
    xbt_queue_pop(mq,file1);
    xbt_queue_pop(mq,file2);
    ready=true;
}
xbt_mutex_release(mtx);
```

5. Evaluation of System

inputline count	time(second)
132	0.073753
264	0.164555
528	0.404362
1056	1.049457
2112	3.217088
4224	10.902204
8448	40.719748

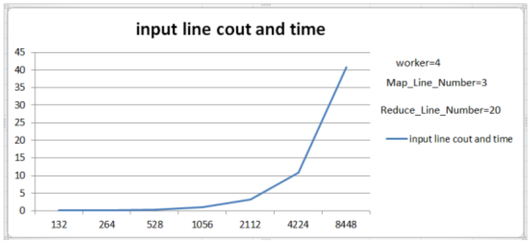


Figure 2 input line count and time

As the Figure 1 shows above, obviously, when the input file's line getting larger and larger, the time we need to compute this MapReduce getting longer and longer (Note: the number of worker is 4; Map_Line_Number is 3 and Reduce line number is 20)

map line number	time(second)
3	1.054861
4	0.804639
11	0.342488
22	0.249915
66	0.144744
88	0.136451
100	0.135523
200	0.134078
300	0.122912
500	0.158094
600	0.16076
900	0.181796

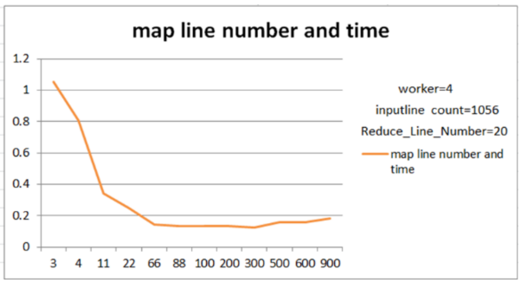


Figure 3 map line number and time

Map line number is the line number of a file that is assigned to each worker. As Figure 2 shows above, when the number of work, the number of input line and Reduce Line Number are fixed, the time using to do MapReduce is as Figure 2. Note that the least time is when we choosing map line number as 500. When the number become larger than 500, the time go up again, which is because more lines are assigned to a worker and the total input line is just 1056, so parallelism is impacted.

reduce line number	time(second)
10	0.17254
50	0.116823
100	0.099816
150	0.087064
250	0.08272
500	0.078204
1000	0.063569
3000	0.058741

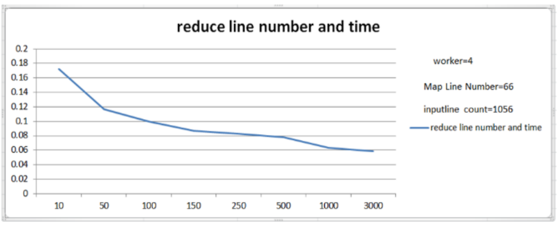
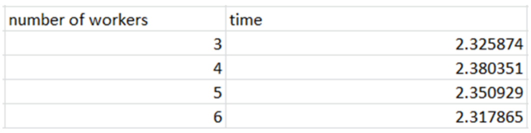


Figure 4 Reduce line number and time

Reduce line number is the number assigned to workers again after mater merging the result of map function. As we can see above, when the reduce line number getting larger, the time we spend on MapReduce function getting less.



number of workers	time
3	2.325874
4	2.380351
5	2.350929
6	2.317865

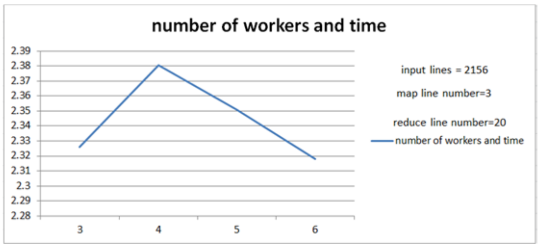


Figure 5 Number of workers and time

When input lines= 2156, map line number=3 and reduce line number=20, the time correspond to the number of workers is shown in Figure 4, where 4 workers consumes more time.

6. Applications

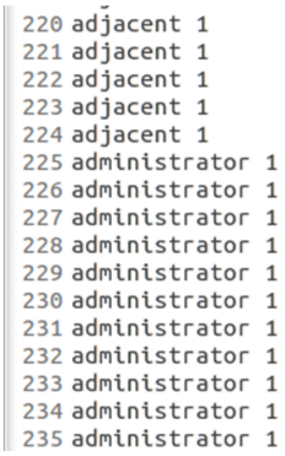
Here we implement two applications that using MapReduce technology to deal with big mass data efficiently.

6.1 The First Application: Keyword

As we mentioned above in this paper, Keyword takes a big mass data from file and count every words' appear time. As the size of the input getting larger and larger, it will be more and more time consuming. With MapReduce, the counting will be distributed to many workers and the workers will do the job in parallel.

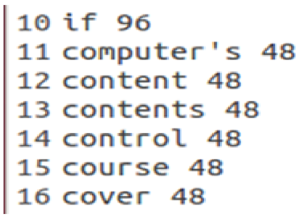
Process

1. Master separate the whole input into pieces of sub data and assign to each worker, each worker gets a piece of input article.
2. After receiving data from master, workers begin to parse the part of article and separate each words, which is call map function, like below:



```
adjacent 1
adjacent 1
adjacent 1
adjacent 1
adjacent 1
administrator 1
administrator 1
administrator 1
administrator 1
administrator 1
administrator 1
administrator 1
administrator 1
administrator 1
administrator 1
administrator 1
```

3. After the workers done the map part, they send the result back to master, master merge the result to form a big table.
4. Then the master distributed this big table using the separating method again to works. Each worker gets a piece of this big table and start to combine the same word together and count how many of it, which is reduce function, As below shows.

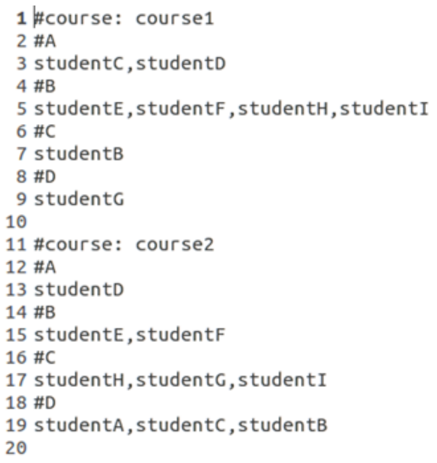


```
if 96
computer's 48
content 48
contents 48
control 48
course 48
cover 48
```

5. After workers done the job in 4, they would again send the result back to master, and master merge them again to form the real result.

6.2 The Second Application: GPA Calculating

Given a file that stores students courses and grade, how to calculate students' GPA efficiently. The file is as below:

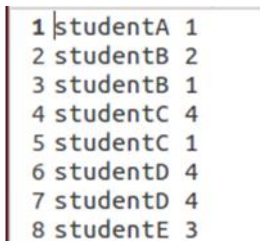


```
#course: course1
#A
studentC,studentD
#B
studentE,studentF,studentH,studentI
#C
studentB
#D
studentG

#course: course2
#A
studentD
#B
studentE,studentF
#C
studentH,studentG,studentI
#D
studentA,studentC,studentB

```

In this case, we use map function to get pairs which includes students' name and their grade. In reduce function we calculate the GPA of each student.

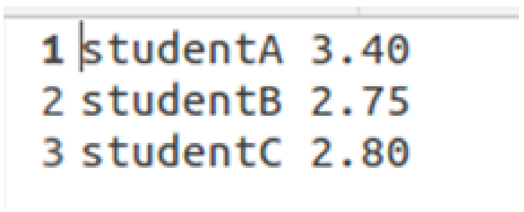
Processes

1. Separate the input data by courses, we can choose each part have one course or more. Just need to multiply 10 to the map_line_number in the master. Then assign to workers.
2. Workers creates pairs includes students' name and their grade using map function, the result see as below:



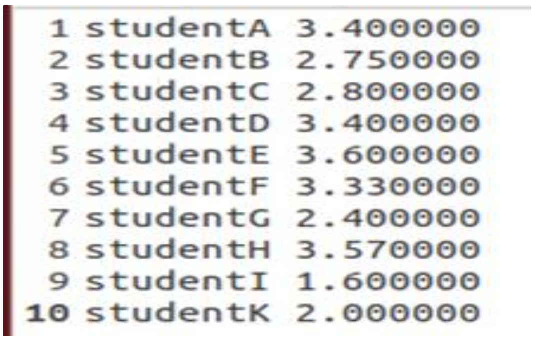
```
studentA 1
studentB 2
studentB 1
studentC 4
studentC 1
studentD 4
studentD 4
studentE 3
```

3. After the workers done the map part, they send the result back to master, master merge the result to form a big table.
4. Then the master distributed this big table using the separating method again to works. Each worker gets a piece of this big table and start to combine the same word together and count how many of it, which is reduce function, As below shows.



```
studentA 3.40
studentB 2.75
studentC 2.80
```

5. After workers done the job in 4, they would again send the result back to master, and master merge them again to form the real result. As below shows:



```
studentA 3.400000
studentB 2.750000
studentC 2.800000
studentD 3.400000
studentE 3.600000
studentF 3.330000
studentG 2.400000
studentH 3.570000
studentI 1.600000
studentK 2.000000
```

7. Future

7.1 MapReduce in Real Distributed Systems

Without real distributed systems in our design, it is difficult for us to demonstrate the performance of the file system. We still wants to perform MapReduce in its hometown. We are not supposed to find large clusters of computers but we will try to connect at least ten laptops to form a small network to build MapReduce.

7.2 MapReduce in An Operating System of A Single Computer

We use thse platform SimGrid to simulate the distributed environment. Even if MapReduce is designed for distributed systems, the principles and ideas are the same. MapReduce can possibly be useful even in a single machine with the structure of multithread or multiprocessor. We will look into that.

8. Conclusion

We demonstrate the algorithm of MapReduce, build two applications based on the technique and try out possible innovations. MapReduce technique is mostly designed for distributed systems, however it would be difficult for us to demonstrate the features on distributed systems, as for students like us cannot easily access to a whole network of clusters of computers. Instead, we use a simulator of clusters on a single PC. The simulator simulates a distributed system for us and we have done the experiments in this simulated environment.

References

[1]https://en.wikipedia.org/wiki/MapReduce
[2]https://en.wikipedia.org/wiki/SimGrid