Contents

1 README

```
itamakatz, liavst2
Itamar Katz (555792977) , Liav Steinberg (203630090)
EX: 3

===================================
=             FILES               =
===================================

- README - this file.
- Makefile
- FrameControl.h
- FrameControl.cpp
- MapReduceFramework.cpp - MapReduce implementation
- Search.cpp - first part of this EX
- 4 jpg images for question 6

===================================
=            DESIGN               =
===================================

MapReduceFramework.cpp:
======================

- Implemented a separate class for the log file.
- We set the Map chunk size to 8, and Reduce set size to 6 (arbitrarily)
  Every thread that finishes a chunk sends a wake-up signal the shuffle
  to start working(Shuffle waits in a loop, using cond_timedwait).
- We set the waiting time of shuffle to 10000000 ns (0.01 sec as writen
  in the pdf).
- Main thread joins the ExecMap and shuffle threads to ensure it can continue
  to Reduce.
- In Map and Reduce, we created a thread vector to hold the threads. The
  reason is that we need to join all of them until they finish their job,
  before we reach the next stage.
- Emit2 function writes to a map <pthread_t, queue<pair<k2,v2>>> using
  pthread_self to ensure that every thread writes to different locations,
  thus avoiding the usage of another mutex. The reason of using queue is that,
  when shuffle sorts elements it pops the head element from this queue , not
  interfering with ExecMap that may also write to the same queue, to its end.
- Emit3 function writes to a map <pthread_t, list<pair<k3,v3>>> also to ensure
  that every thread writes to different locations.
- On producing the final output stage, we created a comparator that uses
  operator < to sort the data.
- Every stage is surrounded by a tic-toc block to measure the time it takes
  to execute it.

Search.cpp:
==========

- Created three classes: Dir, Substring, File.
- For the Framework input, we prepared a list of pair<Substring*, Dir*> where
  Substring in the substring to search, given by the user, and Dir
  represents each directory path.
- Our Map function recieves the substring and a directory path and produces
  a list of <Substring*, File*> where Substring in the given substring and
  File represent a file name in the given directory.
- Our Reduce function recieves the substring and a list of files and calls
  Emit3 function only with the files that hold this substring.
- Emit3 recieves a pair<File*, NULL>, and the framework returns a list of
```

these pairs.
- On the printing stage, we collect every pair in the framework output and
print its key, which is the wanted file name.

==========
= ANSWERS =
==========

Part 2: Theoretical Questions:
==========

1. Because the whole purpose of select is to monitor multiple file
descriptors, waiting until one or more of them becomes "ready"
for IO operation, it is clear that Shuffle needs to use it, to monitor
what file descriptor is now ready for sorting its data. So it will also
be the thread that reads from the pipe. Our idea of implementing this
is to fork a process for each ExecMap thread. The processes will
communicate with Shuffle process using pipes, one for each thread.
There is also one more pipe to communicate between Shuffle and the
main process.
Each ExecMap writes to his pipe a string representing the pair he output.
Shuffle, which uses select to detect which pipe is ready, reads
this string and checks its key. Shuffle writes to the main process
pipe the pair itself, along with a position representing where he
should place this pair in his data structure. This can be done in the
same manner as g_shuffleStructs is implemented. The last process to
finish his job should send a signal to notify Shuffle to also finish
his job.

2. Because of the lack of hyper-threading support, each core will run in a
single thread mode, leaving us with multiThreadLevel of 6, as two cores
are occupied by main thread and shuffle.

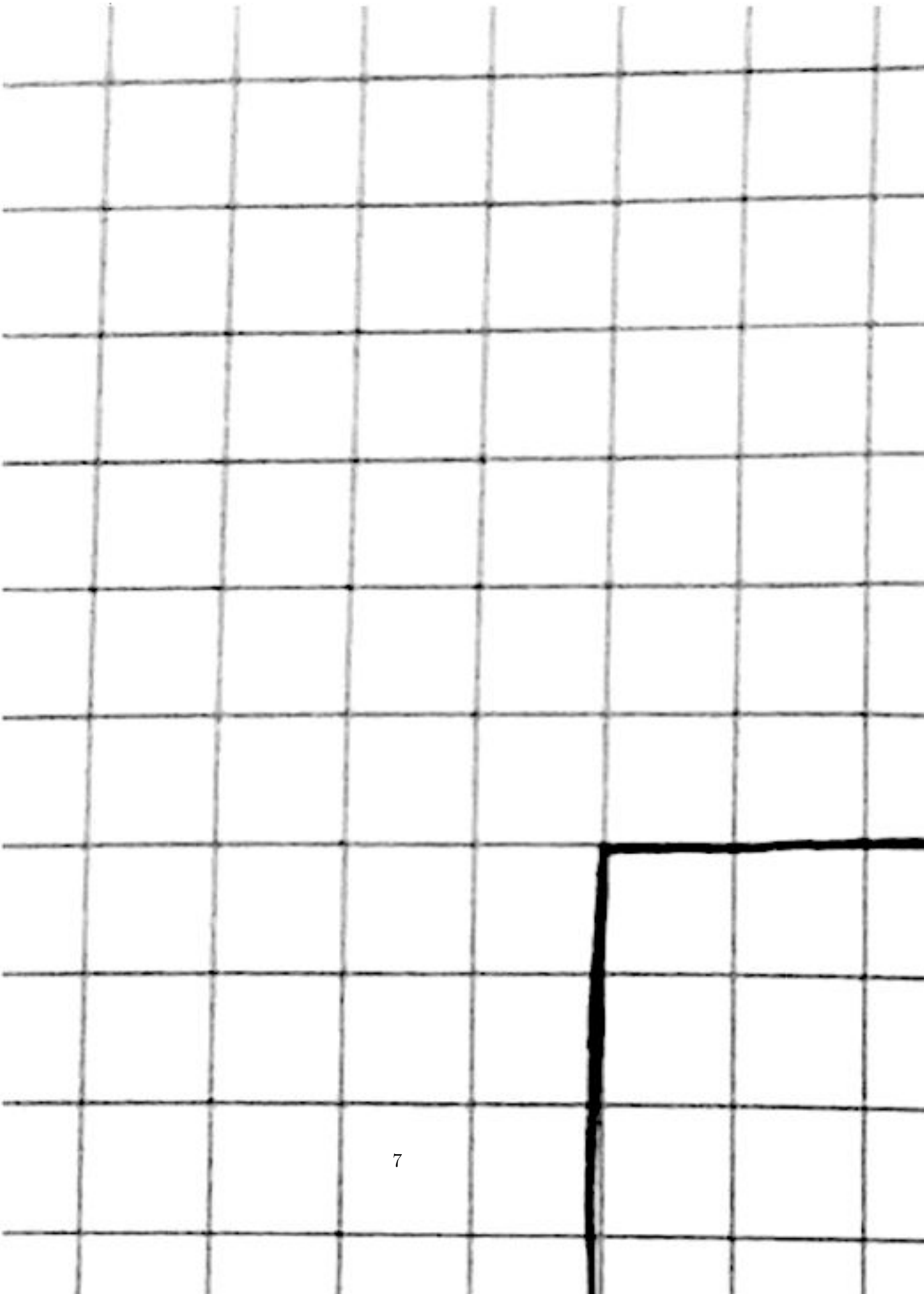
3. In Nira's case:
a. her program cannot occupy more than a single processor,
so utilization for multi cores will be at the minimum.
b. Because she uses a "single flow", there is no need for a scheduler.
c. Also there is no need for communication beacuse there is a single
thread and process.
d. Becuase of the "single flow", the whole process will stop progress
in case of a block.
e. Overall speed is relatively slow, because the relative size of the
program and the fact that is a single thread running.
In Moti's case:
a. He uses kernel-level threads which can run on different processors
and cores, so utilization can be high.
b. The scheduler is supplied by the OS so it must be sophisticated.
c. Simple communication between the threads - low communication time.
d. If a thread blocks, this does not affect the others. Meaning the
ability to progress exists.
e. Operations require a kernel trap, but little work - relatively high
overall speed.
In Danny's case:
a. All threads must share the same processor - low utilization.
b. The ability to create such scheduler is decreasing, because the
user may make some mistakes, rather than the OS itself.
c. As with kernel-level threads, simple communication - low
communication time.
d. If a thread blocks, the whole process is blocked.
e. Everything is done at user level - low overall speed.
In Galit's case:
a. Processes can run on different processors, so utilization can be
high.
b. Like kernel-level threads, the scheduler is supplied by the OS so
it must be sophisticated.
c. Require the OS to communicate (using pipes), relatively high

communication time.
d. Like kernel-level threads, if a process blocks, this does not affect the others. Meaning the ability to progress exists.
e. All oeprations require a kernel trap, and significant work. perhaps relatively low overall speed.

4. Processes, as said in the previous exercise, do not share any of the above. each process has its own stack, heap and global variables from the point of the fork command and on.
However, kernel level threads and user level threads share between them the heap and the global variables, but each one of them has its own independent stack.

5. A deadlock is a situation which occurs when a process or a thread is entering a waiting mode because the resource they request is currently held by another process\thread which is also waiting for another resource held by another waiting process\thread (circular waiting).
As with deadlock, livelock threads\processes are unable to make further progress. However, the difference is that the threads are not blocked - they are too busy to responding to each other to resume work. In other words, livelock can arise when two or more tasks use the same resource causing a circular dependency where those tasks keep running forever.
An example of a deadlock (in real life) is the philosophers problem (as mentioned in class): Five philosophers sit to dinner next to a round table where a bowl of spaghetti is placed at the center. Each philosoper needs two forks to grasp some spaghetti from the bowl, but there is only one fork for each one of them. Every philosopher takes the fork to his left, so that the fork to his left is already picked up by another philosopher, creating a situation they all sit there indefinitely.
An example of a livelock (in real life): consider two men attempting to pass each other in a corridor. Liav moves to his left to let Itamar pass, while Itamar moves to his right to let Liav pass. Seeing they are still blocking each other, Liav moves to his right while Itamar moves to his left, and so on.

2 FirstComeFirstServeGanttChart.jpg



3 FrameControl.h

```cpp
#ifndef EX3_LOG_H
#define EX3_LOG_H

#include "MapReduceFramework.h"

#include <fstream>
#include <ctime>
#include <sys/time.h>
#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <pthread.h>
#include <algorithm>


#define DATE_LEN 80
#define NANO_MICROSEC_ADJUST 1000
#define NANO_SEC_ADJUST 1000000000
#define FAILURE 1
#define MAP_SET_SIZE 8
#define REDUCE_SET_SIZE 6
#define SHUFFLE_DELAY 10000000
#define START -1

typedef enum{THREAD_STATUS = 1, MAP_SHUFFLE_TIME = 2, REDUCE_TIME = 3} Action;
typedef std::pair<k2Base*, v2Base*> MID_ITEM;
class LogFile;
class Comparator;
class Manager;


extern LogFile* g_lf;
extern Manager manager;
extern timeval g_timeBegin, g_timeEnd;

extern int g_run_shuffle;
extern int g_protected_index;


extern std::vector<IN_ITEM> g_InputVec;
extern std::vector<std::pair<k2Base*, V2_LIST>> g_ShuffleVec;
extern std::map<pthread_t, std::queue<MID_ITEM>*> g_MapMap;
extern std::map<k2Base*, V2_LIST*, Comparator> g_ShuffleMap;
extern std::map<pthread_t, OUT_ITEMS_LIST*> g_ReduceMap;


extern pthread_mutex_t ExecReduce_lock;
extern pthread_mutex_t ExecMap_lock;
extern pthread_mutex_t index_lock;
extern pthread_mutex_t log_lock;
extern pthread_mutex_t shuffle_lock;
extern pthread_mutex_t shuffle_condition;
extern pthread_cond_t shuffle_wakeup;


/**
 * functor for sorting the shuffle map
 */
```

```
60  class Comparator {
61  public:
62      bool operator()(const k2Base* key1, const k2Base* key2) const;
63  };
64
65
66  class Manager {
67  private:
68
69      /**
70       * clear the shuffle output
71       */
72      void _clearg_ShuffleMap();
73
74      /**
75       * clear Map output
76       */
77      void _clearg_MapMap();
78
79      /**
80       * clear Reduce output
81       */
82      void _clearg_ReduceMap();
83
84      /**
85       * destroy all mutexes
86       */
87      void _mutex_destroy_();
88
89  public:
90
91      /**
92       * clear the global data structures
93       */
94      void clearStructs();
95
96      /**
97       * called whenever a framework function fails
98       * @param who - which function failed.
99       */
100     void Abort(std::string who);
101
102     /**
103      * init all the framework muitexes
104      */
105     void mutex_init();
106
107 };
108
109
110 /**
111  * implements the log file
112  */
113 class LogFile{
114 private:
115     std::ofstream __lfile;
116
117 public:
118     /**
119      * constructor
120      */
121     LogFile(int mThreadLevel);
122
123     /**
124      * destructor
125      */
126     ~LogFile();
127
```

```
128         /**
129          * documents thread creation and termination
130          */
131         void tstatus(std::string name, std::string state);
132 
133         /**
134          * documents map and shuffle time
135          */
136         void MapShuffleTime(double time);
137 
138         /**
139          * documents reduce time
140          */
141         void ReduceTime(double time);
142 
143         /**
144          * opens the log file for documentation
145          */
146         static void open_log_file(int multiThreadLevel);
147 
148 
149         /**
150          * start time measurement
151          */
152         static void tic();
153 
154 
155         /**
156          * end time measurement
157          */
158         static void toc();
159 
160 
161         /**
162          * calculate the time elapsed from tic to toc
163          */
164         static double elapsedTime();
165 
166 
167         /**
168          * documents progress to the log file
169          * @param action - which action
170          * @param name - the name of the thread
171          * @param state - created or terminated
172          */
173         static void document(Action action, std::string name, std::string state);
174 };
175 
176 #endif //EX3_LOG_H
```

4 FrameControl.cpp

```cpp
#include "FrameControl.h"

////////////////   class LogFile implementation  ////////////////////

LogFile::LogFile(int mThreadLevel){
    __lfile.open(".MapReduceFramework.log", std::ios_base::app);
    __lfile << "runMapReduceFramework started with " <<
    mThreadLevel << " threads" << std::endl;
}

LogFile::~LogFile(){
    __lfile << "runMapReduceFramework finished" << std::endl;
    __lfile.close();
}

void LogFile::tstatus(std::string name, std::string state){
    time_t rawtime;
    char date[DATE_LEN];
    std::time(&rawtime);
    struct tm* timeinfo = localtime(&rawtime);
    strftime(date,DATE_LEN,"[%d.%m.%Y %H:%M:%S]",timeinfo);
    __lfile << "Thread " << name << state << date << std::endl;
}

void LogFile::MapShuffleTime(double time){
    __lfile << "Map and Shuffle took " << time << " ns" << std::endl;
}

void LogFile::ReduceTime(double time){
    __lfile << "Reduce took " << time << " ns" << std::endl;
}

void LogFile::open_log_file(int multiThreadLevel){
    try {
        g_lf = new LogFile(multiThreadLevel);
    } catch (std::bad_alloc& err){
        manager.Abort("new");
    }
}

/**
* start time measurement
*/
void LogFile::tic() {
    if (gettimeofday(&g_timeBegin, NULL)) {
        manager.Abort("gettimeofday");
    }
    return;
}


/**
* end time measurement
*/
void LogFile::toc() {
    if (gettimeofday(&g_timeEnd, NULL)) {
        manager.Abort("gettimeofday");
    }
    return;
```

```
60  }
61
62
63  /**
64  * calculate the time elapsed from tic to toc
65  */
66  double LogFile::elapsedTime() {
67      return (g_timeEnd.tv_sec - g_timeBegin.tv_sec) * NANO_SEC_ADJUST +
68             (g_timeEnd.tv_usec - g_timeBegin.tv_usec) * NANO_MICROSEC_ADJUST;
69  }
70
71
72  /**
73  * documents progress to the log file
74  * @param action - which action
75  * @param name - the name of the thread
76  * @param state - created or terminated
77  */
78  void LogFile::document(Action action, std::string name, std::string state) {
79      pthread_mutex_lock(&log_lock);
80      switch (action) {
81          case THREAD_STATUS:
82              g_lf->tstatus(name, state);
83              break;
84          case MAP_SHUFFLE_TIME:
85              g_lf->MapShuffleTime(elapsedTime());
86              break;
87          case REDUCE_TIME:
88              g_lf->ReduceTime(elapsedTime());
89              break;
90          default:
91              break;
92      }
93      pthread_mutex_unlock(&log_lock);
94      return;
95  }
96
97  ////////////////   class Comparator implementation  ////////////////////
98
99
100 bool Comparator::operator()(const k2Base* key1, const k2Base* key2) const{
101     return *key1 < *key2;
102 }
103
104
105 ////////////////   class LogFile implementation  ////////////////////
106
107
108 void Manager::_clearg_ShuffleMap(){
109     for (auto& mapItem: g_ShuffleMap){
110         if (mapItem.second) {
111             delete mapItem.second;
112         }
113     }
114     g_ShuffleMap.clear();
115 }
116
117 void Manager::_clearg_MapMap(){
118     for (auto& mapItem: g_MapMap){
119         if (mapItem.second) {
120             delete mapItem.second;
121         }
122     }
123     g_MapMap.clear();
124 }
125
126 void Manager::_clearg_ReduceMap(){
127     for (auto& mapItem: g_ReduceMap){
```

```
128             if (mapItem.second) {
129                 delete mapItem.second;
130             }
131         }
132         g_ReduceMap.clear();
133     }
134
135     void Manager::mutex_init() {
136         ExecReduce_lock = PTHREAD_MUTEX_INITIALIZER;
137         ExecMap_lock = PTHREAD_MUTEX_INITIALIZER;
138         index_lock= PTHREAD_MUTEX_INITIALIZER;
139         log_lock= PTHREAD_MUTEX_INITIALIZER;
140         shuffle_lock= PTHREAD_MUTEX_INITIALIZER;
141         shuffle_condition = PTHREAD_MUTEX_INITIALIZER;
142         shuffle_wakeup = PTHREAD_COND_INITIALIZER;
143     }
144
145     void Manager::_mutex_destroy_(){
146         if (pthread_mutex_destroy(&ExecMap_lock)){
147             Abort("pthread_mutex_destroy");
148         }
149         if (pthread_mutex_destroy(&index_lock)){
150             Abort("pthread_mutex_destroy");
151         }
152         if (pthread_mutex_destroy(&log_lock)){
153             Abort("pthread_mutex_destroy");
154         }
155         if (pthread_mutex_destroy(&shuffle_lock)){
156             Abort("pthread_mutex_destroy");
157         }
158         if (pthread_mutex_destroy(&shuffle_condition)){
159             Abort("pthread_mutex_destroy");
160         }
161         if (pthread_cond_destroy(&shuffle_wakeup)){
162             Abort("pthread_mutex_destroy");
163         }
164     }
165
166     void Manager::clearStructs(){
167         _clearg_ShuffleMap();
168         _clearg_MapMap();
169         _clearg_ReduceMap();
170         g_ShuffleVec.clear();
171         g_InputVec.clear();
172         _mutex_destroy_();
173         if (g_lf){
174             delete(g_lf);
175         }
176         return;
177     }
178
179     void Manager::Abort(std::string who) {
180         std::cerr << "MapReduceFramework Failure: " << who
181                                             << " failed." << std::endl;
182         manager.clearStructs();
183         exit(FAILURE);
184     }
```

5 Makefile

```
CFLAGS = -g -Wall -std=c++11
LDFLAGS = -pthread
TAR_NAME = ex3.tar

FWLIB_SOURCES = FrameControl.cpp MapReduceFramework.cpp
EXEC_SOURCES = $(FWLIB_SOURCES) Search.cpp

HEADERS = FrameControl.h MapReduceFramework.h MapReduceClient.h


EXEC_OBJS = $(EXEC_SOURCES:.cpp=.o)
FWLIB_OBJS = $(FWLIB_SOURCES:.cpp=.o)

IMAGES = FirstComeFirstServeGanttChart.jpg PrioritySchedulingGanttChart.jpg\
        RoundRobinGanttChart.jpg ShortestRemainingTimeFirstGanttChart.jpg
EXTRA_FILES = README Makefile FrameControl.h
TAR_FILES = $(EXEC_SOURCES) $(EXTRA_FILES) $(IMAGES)

EXECUTABLE = Search
FWLIB = MapReduceFramework.a

all: $(EXECUTABLE) $(FWLIB)


$(EXECUTABLE): $(EXEC_OBJS)
    $(CXX) $(LDFLAGS) $^ -o $@

$(FWLIB): $(FWLIB_OBJS)
    ar rcs $@ $^

%.o: %.cpp $(HEADERS)
    $(CXX) -c $(CFLAGS) $<

tar:
    tar -cvf $(TAR_NAME) $(TAR_FILES)


clean:
    rm -rf $(TAR_NAME) $(EXEC_OBJS) $(FWLIB) $(EXECUTABLE)


.PHONY:
    all tar clean
```

6 MapReduceFramework.cpp

```cpp
#include "MapReduceFramework.h"
#include "FrameControl.h"


using namespace std;


LogFile* g_lf;
timeval g_timeBegin, g_timeEnd;
Manager manager;

int g_run_shuffle;
int g_protected_index;


vector<IN_ITEM> g_InputVec;
vector<pair<k2Base*, V2_LIST>> g_ShuffleVec;
map<pthread_t, queue<MID_ITEM>*> g_MapMap;
map<k2Base*, V2_LIST*, Comparator> g_ShuffleMap;
map<pthread_t, OUT_ITEMS_LIST*> g_ReduceMap;


pthread_mutex_t ExecReduce_lock;
pthread_mutex_t ExecMap_lock;
pthread_mutex_t index_lock;
pthread_mutex_t log_lock;
pthread_mutex_t shuffle_lock;
pthread_mutex_t shuffle_condition;
pthread_cond_t shuffle_wakeup;


/**
 * multithreaded function to execute Map function in parallel
 * @param mapReduceVoid - the struct holding the Map function
 */
void* ExecMap(void* mapReduceVoid){

    LogFile::document(THREAD_STATUS, "ExecMap", " created ");
    MapReduceBase* mapReduce = (MapReduceBase*) mapReduceVoid;
    int InputVecSize = (int)g_InputVec.size();

    int i = 0;
    do
    {
        pthread_mutex_lock(&index_lock);
        g_protected_index++;
        i = g_protected_index * MAP_SET_SIZE;
        pthread_mutex_unlock(&index_lock);

        // prepare the correct number of iterations
        int iterations = min(MAP_SET_SIZE, InputVecSize - i);

        for(int v = i; iterations > 0 ; iterations--, ++v){
            mapReduce->Map(g_InputVec[v].first, g_InputVec[v].second);
        }

        pthread_cond_signal(&shuffle_wakeup);

    } while (i < InputVecSize);
```

```
60
61      pthread_mutex_lock(&shuffle_condition);
62      g_run_shuffle--; // when it gets to zero, shuffle stop its routine
63      pthread_mutex_unlock(&shuffle_condition);
64
65      LogFile::document(THREAD_STATUS, "ExecMap", " terminated ");
66
67      pthread_exit(NULL);
68      return NULL;
69  }
70
71  /**
72   * shuffling routine for shuffle thread
73   */
74  void sortMapItems(){
75
76      // <--- shuffling routine ---> //
77
78      for (map<pthread_t, queue<MID_ITEM>*>::iterator it = g_MapMap.begin();
79           it != g_MapMap.end(); it++){
80          while(!it->second->empty()) {
81              MID_ITEM next_item = it->second->front();
82              if (g_ShuffleMap.find(next_item.first) == g_ShuffleMap.end()){
83                  try {
84                      g_ShuffleMap[next_item.first] = new V2_LIST();
85                  } catch (bad_alloc& err){
86                      manager.Abort("new");
87                  }
88              }
89              g_ShuffleMap[next_item.first]->push_back(next_item.second);
90
91              it->second->pop();
92          }
93      }
94      return;
95  }
96
97
98  /**
99   * execute the shuffle routine
100  */
101 void* Shuffle(void*){
102
103     LogFile::document(THREAD_STATUS, "Shuffle", " created ");
104     struct timespec waitTime;
105     waitTime.tv_sec = 0;
106     waitTime.tv_nsec = SHUFFLE_DELAY;
107
108     do
109     {
110         pthread_mutex_lock(&shuffle_lock);
111         pthread_cond_timedwait(&shuffle_wakeup, &shuffle_lock, &waitTime);
112         pthread_mutex_unlock(&shuffle_lock);
113
114         sortMapItems();
115
116     } while (g_run_shuffle);
117
118     sortMapItems(); // if the loop ended too soon
119
120     LogFile::document(THREAD_STATUS, "Shuffle", " terminated ");
121     pthread_exit(NULL);
122     return NULL;
123 }
124
125 /**
126  * multithreaded function to execute Reduce function in parallel
127  * @param mapReduceVoid - the struct holding the Reduce function
```

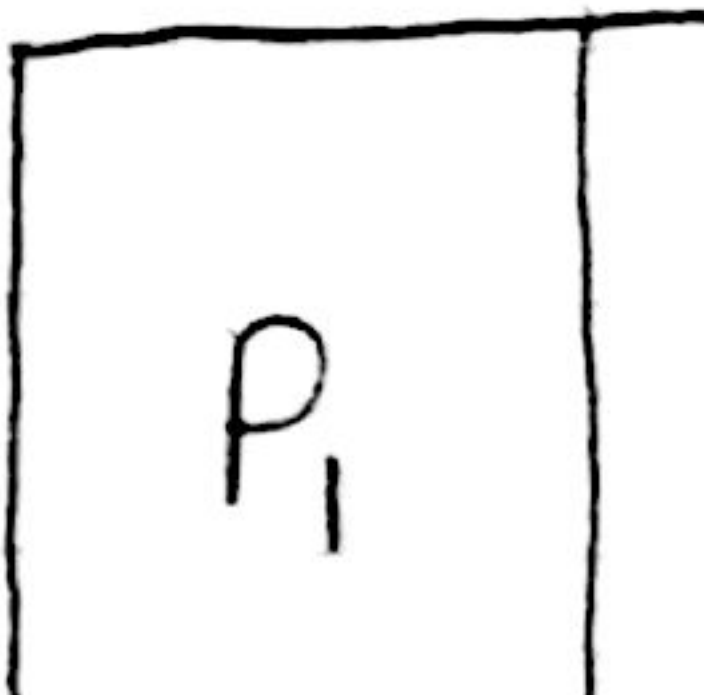
```
128  */
129 void* ExecReduce(void* mapReduceVoid){
130
131     LogFile::document(THREAD_STATUS, "ExecReduce", " created ");
132     MapReduceBase* mapReduce = (MapReduceBase*) mapReduceVoid;
133     int ShuffleVecSize = (int)g_ShuffleVec.size();
134
135     int i = 0;
136     do
137     {
138         pthread_mutex_lock(&index_lock);
139         g_protected_index++;
140         i = g_protected_index * REDUCE_SET_SIZE;
141         pthread_mutex_unlock(&index_lock);
142
143         // prepare the correct number of iterations
144         int iterations = min(REDUCE_SET_SIZE, ShuffleVecSize - i);
145
146         for(int v = i; iterations > 0 ; iterations--, ++v){
147             mapReduce->Reduce(g_ShuffleVec[v].first, g_ShuffleVec[v].second);
148         }
149
150     } while (i < ShuffleVecSize);
151
152     LogFile::document(THREAD_STATUS, "ExecReduce", " terminated ");
153     pthread_exit(NULL);
154     return NULL;
155 }
156
157 /**
158  * execute the Map and Shuffle procedures
159  * @param mapReduce - a struct holding the Map function
160  * @param itemsList - the input of the framework
161  * @param multiThreadLevel - the thread amount threshold
162  */
163 void runMapShuffle(MapReduceBase& mapReduce,
164                    IN_ITEMS_LIST& itemsList, int multiThreadLevel){
165     g_run_shuffle = multiThreadLevel;
166     g_protected_index = START;
167     manager.mutex_init();
168     pthread_t shuffleThread;
169     pthread_t MapThreads[multiThreadLevel];
170
171     // converting from list to vector
172     for (auto& item: itemsList) {
173         g_InputVec.push_back(item);
174     }
175
176     // ensuring that Emit2 would not raise segmentation fault
177     pthread_mutex_lock(&ExecMap_lock);
178
179     // create the ExecMap thread pool
180     for (int i = 0; i < multiThreadLevel; i++) {
181         if (pthread_create(&MapThreads[i], NULL, ExecMap, (void*)&mapReduce)){
182             manager.Abort("pthread_create");
183         }
184     }
185
186     try {
187         for (int i = 0; i < multiThreadLevel; i++) {
188             g_MapMap[MapThreads[i]] = new queue<MID_ITEM>();
189         }
190     } catch (bad_alloc &err) {
191         manager.Abort("new");
192     }
193
194     // now ExecMap threads can write to their structs
195     pthread_mutex_unlock(&ExecMap_lock);
```

```
196
197
198     if (pthread_create(&shuffleThread, NULL, Shuffle, nullptr)){
199         manager.Abort("pthread_create");
200     }
201
202     // joining the ExecMap and Shuffle threads
203     for (int i = 0; i < multiThreadLevel; i++){
204         if (pthread_join(MapThreads[i], NULL)){
205             manager.Abort("pthread_join");
206         }
207     }
208
209     if (pthread_join(shuffleThread, NULL)){
210         manager.Abort("pthread_join");
211     }
212
213     return;
214 }
215
216 /**
217  * Execute Reduce procedure
218  * @param mapReduce - the struct holding the Reduce function
219  * @param multiThreadLevel - the thread amount threshold
220  */
221 void runReduce(MapReduceBase& mapReduce, int multiThreadLevel){
222
223     g_protected_index = START;
224     pthread_t ReduceThreads[multiThreadLevel];
225
226     // converting from map to vector
227     for (auto& item: g_ShuffleMap) {
228         g_ShuffleVec.push_back
229                 (pair<k2Base*,V2_LIST>(item.first,*(item.second)));
230     }
231
232     // ensuring that writing in Emit3 would not raise segmentation fault
233     pthread_mutex_lock(&ExecReduce_lock);
234
235     // create the ExecReduce thread pool
236     for (int i = 0; i < multiThreadLevel; i++) {
237         if (pthread_create(&ReduceThreads[i], NULL, ExecReduce,
238                                                     (void*)&mapReduce)){
239             manager.Abort("pthread_create");
240         }
241     }
242
243     try {
244         for (int i = 0; i < multiThreadLevel; i++) {
245             g_ReduceMap[ReduceThreads[i]] = new OUT_ITEMS_LIST();
246         }
247     } catch (bad_alloc &err) {
248         manager.Abort("new");
249     }
250
251     // now ExecReduce threads can write to their structs
252     pthread_mutex_unlock(&ExecReduce_lock);
253
254     // joining the ExecReduce threads
255     for (int i = 0; i < multiThreadLevel; i++){
256         if (pthread_join(ReduceThreads[i], NULL)){
257             manager.Abort("pthread_join");
258         }
259     }
260
261     return;
262 }
263
```

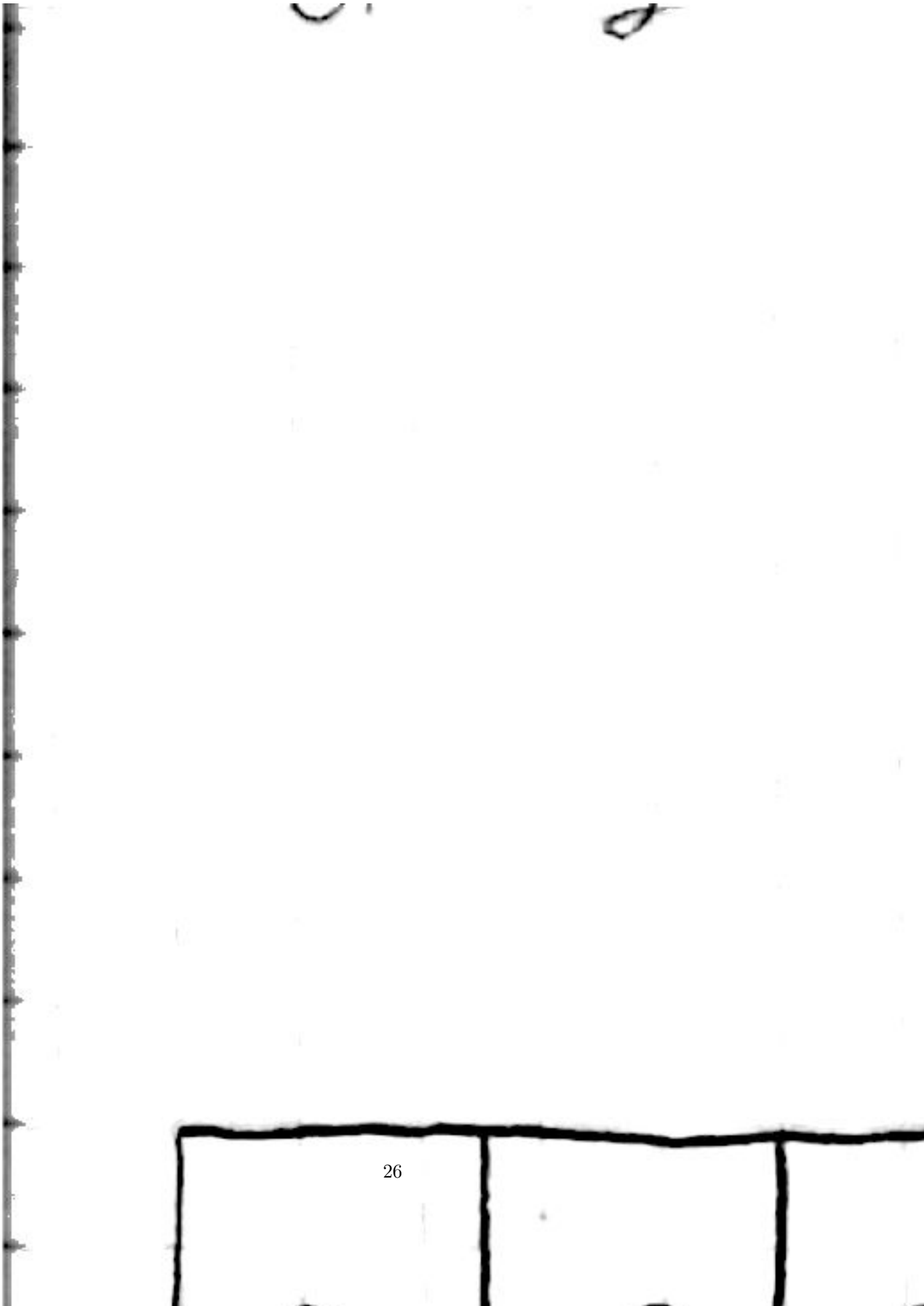
```
264
265 /**
266  * producing the final output
267  */
268 OUT_ITEMS_LIST produceFinalOutput(){
269     OUT_ITEMS_LIST finalOutput;
270     auto comparator = [](const OUT_ITEM& o1, const OUT_ITEM& o2){
271         return *o1.first < *o2.first;
272     };
273
274     // merging and sorting
275     for(map<pthread_t, OUT_ITEMS_LIST*>::iterator it = g_ReduceMap.begin();
276         it != g_ReduceMap.end(); it++){
277         it->second->sort(comparator);
278         finalOutput.merge(*(it->second));
279     }
280     finalOutput.sort(comparator);
281     return finalOutput;
282 }
283
284 /**
285  * the main MapReduce framework of this project
286  * @param mapReduce - a struct holding the Map function
287  * @param itemsList - the input of the framework
288  * @param multiThreadLevel - the thread amount threshold
289  */
290 OUT_ITEMS_LIST runMapReduceFramework(MapReduceBase& mapReduce,
291                     IN_ITEMS_LIST& itemsList, int multiThreadLevel){
292
293     LogFile::tic();
294     LogFile::open_log_file(multiThreadLevel);
295
296     ////////////////////// starting map and shuffle ////////////////////////
297
298     runMapShuffle(mapReduce, itemsList, multiThreadLevel);
299     LogFile::toc();
300
301     LogFile::document(MAP_SHUFFLE_TIME, "", "");
302
303     ///////////////// ended map and shuffle, starting reduce ////////////////
304
305     LogFile::tic();
306     runReduce(mapReduce, multiThreadLevel);
307     OUT_ITEMS_LIST finalOutput = produceFinalOutput();
308     LogFile::toc();
309
310     LogFile::document(REDUCE_TIME, "", "");
311
312     ////////////////////////// end of framework ////////////////////////////
313
314     manager.clearStructs();
315     return finalOutput;
316 }
317
318 /**
319  * add a pair to g_MapMap
320  */
321 void Emit2 (k2Base* k2, v2Base* v2){
322     pthread_mutex_lock(&ExecMap_lock);
323     g_MapMap[pthread_self()]->push(MID_ITEM(k2, v2));
324     pthread_mutex_unlock(&ExecMap_lock);
325 }
326
327 /**
328  * add a pair to g_ReduceMap
329  */
330 void Emit3 (k3Base* k3, v3Base* v3){
331     pthread_mutex_lock(&ExecReduce_lock);
```

```
332       g_ReduceMap[pthread_self()]->push_back(OUT_ITEM(k3, v3));
333       pthread_mutex_unlock(&ExecReduce_lock);
334   }
```

7 PrioritySchedulingGanttChart.jpg



8 RoundRobinGanttChart.jpg



9 Search.cpp

```cpp
#include "MapReduceFramework.h"
#include <iostream>
#include <dirent.h>
#include <algorithm>
#include <memory>

#define FALIURE -1
#define SUCCESS 0
#define MULTI_THREAD_LEVEL 5
#define USAGE "Usage: <substring to search> <folders, separated by space>"

/**
 * representing the directory name
 */
class Dir : public v1Base {
private:
    std::string __dir;

public:

    explicit Dir(std::string dir_str){
        __dir = dir_str;
    }

    explicit Dir(const Dir& dir_str){
        __dir = const_cast<Dir&>(dir_str).__dir;
    }

    virtual ~Dir(){}

    std::string getDir() const{
        return __dir;
    }

};


/**
 * representing the substring
 */
class Substring : public k1Base, public k2Base {
private:
    std::string __sub;

public:

    explicit Substring(std::string sub){
        __sub = sub;
    }

    explicit Substring(const Substring& sub_str){
        __sub = const_cast<Substring&>(sub_str).__sub;
    }

    virtual ~Substring(){}

    virtual bool operator<(const k2Base &other) const{
        return ((__sub.compare(((const Substring&)other).__sub) < 0));
    }
```

```
60
61      virtual bool operator<(const k1Base &other) const{
62          return ((__sub.compare(((const Substring&)other).__sub) < 0));
63      }
64
65      std::string getSub() const{
66          return __sub;
67      }
68  };
69
70
71  /**
72   * representing a file name
73   */
74  class File : public v2Base, public k3Base{
75  private:
76      std::string __filename;
77
78  public:
79
80      explicit File(std::string str){
81          __filename = str;
82      }
83
84      explicit File(const File& filename_str){
85          __filename = const_cast<File&>(filename_str).__filename;
86      }
87
88      virtual ~File(){}
89
90      virtual bool operator< (const k3Base &other) const{
91          return ((__filename.compare(((const File&)other).__filename) < 0));
92      }
93      std::string getFile() const{
94          return __filename;
95      }
96      void print(){
97          std::cout << __filename << std::endl;
98      }
99  };
100
101 /**
102  * an abstract class holding the desired map and reduce functions
103  * used inside the framework
104  */
105 class MapReduce : public MapReduceBase{
106
107 public:
108
109     /**
110      * implementing the map function
111      */
112     virtual void Map(const k1Base *const key, const v1Base *const val) const{
113         Substring* sub = dynamic_cast<Substring*>(const_cast<k1Base*>(key));
114         Dir* val1 = dynamic_cast<Dir*>(const_cast<v1Base*>(val));
115         DIR* dir;
116         struct dirent* ent;
117         if ((dir = opendir(val1->getDir().c_str())) != NULL){
118             while ((ent = readdir(dir)) != NULL) {
119                 File* file = new File(ent->d_name);
120                 Emit2(sub, file);
121             }
122             closedir(dir);
123         }
124         return;
125     }
126
127     /**
```

```
128         * implementing the reduce function
129         */
130        virtual void Reduce(const k2Base *const key, const V2_LIST &val) const{
131            Substring* sub = dynamic_cast<Substring*>(const_cast<k2Base*>(key));
132            for(V2_LIST::const_iterator it = val.cbegin(); it != val.cend();it++){
133                File* file = dynamic_cast<File*>(const_cast<v2Base*>(*it));
134                if (file->getFile().find(sub->getSub()) != std::string::npos){
135                    Emit3(file, NULL);
136                } else {
137                    delete(file);
138                }
139            }
140            return;
141        }
142    };
143
144
145    /**
146     * prints the file names the framework returned
147     * @param outlist - the list of the desired file names
148     */
149    void printFiles(OUT_ITEMS_LIST &outlist){
150        while (!outlist.empty()){
151            OUT_ITEM& out = outlist.front();
152            ((File*)out.first)->print();
153            delete(out.first);
154            outlist.pop_front();
155        }
156    }
157
158    /**
159     * clears the input of the framework
160     */
161    void _clear_inlist(IN_ITEMS_LIST& inlist){
162        for (auto& item: inlist){
163            if (item.second){
164                delete(item.second);
165            }
166        }
167        inlist.clear();
168    }
169
170
171
172    /**
173     * the main function for this program
174     */
175    int main(int argc, char* argv[]){
176
177        // there are no arguments
178        if (argc == 1){
179            std::cerr << USAGE << std::endl;
180            return FALIURE;
181        }
182
183        // the g_FWinput is the substring only
184        if (argc == 2){
185            return SUCCESS;
186        }
187
188        IN_ITEMS_LIST inlist;
189        //Substring* substring = new Substring(argv[1]);
190        Substring* substring = new Substring(argv[1]);
191
192        // filling the g_InputVec list
193        for(int i = 2; i < argc; inlist.push_back
194                (IN_ITEM(&(*substring), new Dir(argv[i++]))));
195
```

```
196       MapReduce MR;
197
198       // running the framework and printing the desired files
199       OUT_ITEMS_LIST outlist = runMapReduceFramework
200                                       (MR, inlist, MULTI_THREAD_LEVEL);
201       printFiles(outlist);
202
203       delete(substring);
204       _clear_inlist(inlist);
205       return SUCCESS;
206   }
```

10 ShortestRemainingTimeFirstGanttChart.jpg