Contents

1 Basic Test Results

```
1  ============== Tar Content Test ==============
2  found README
3  found Makefile
4  tar content test PASSED!
5
6  ============== logins ==============
7  login names mentioned in file:  tomka,alonemanuel
8  Please make sure that these are the correct login names.
9
10 ============== make Command Test ==============
11 g++ -Wall -std=c++11 -g -pthread -I.   -c -o MapReduceFramework.o MapReduceFramework.cpp
12 g++ -Wall -std=c++11 -g -pthread -I.   -c -o Barrier.o Barrier.cpp
13 ar rv libMapReduceFramework.a MapReduceFramework.o Barrier.o
14 a - MapReduceFramework.o
15 a - Barrier.o
16 ranlib libMapReduceFramework.a
17
18 ar: creating libMapReduceFramework.a
19
20 make command test PASSED!
21
22 Pre-submission passed!
23 Keep in mind that this script tests only basic elements of your code.
```

2 README

```
1  tomka, alonemanuel
2  Tom Kalir (316426485), Alon Emanuel (205894058)
3  EX: 3
4
5  FILES:
6
7  MapReduceFramework.cpp
8  Barrier.h
9  Barrier.cpp
10 Makefile
11
12 REMARKS:
```

3 Barrier.h

```
#ifndef BARRIER_H
#define BARRIER_H
#include <pthread.h>

// a multiple use barrier

class Barrier {
public:
    Barrier(int numThreads);
    ~Barrier();
    void barrier();

private:
    pthread_mutex_t mutex;
    pthread_cond_t cv;
    int count;
    int numThreads;
};

#endif //BARRIER_H
```

4 Barrier.cpp

```cpp
#include "Barrier.h"
#include <cstdlib>
#include <cstdio>

Barrier::Barrier(int numThreads)
 : mutex(PTHREAD_MUTEX_INITIALIZER)
 , cv(PTHREAD_COND_INITIALIZER)
 , count(0)
 , numThreads(numThreads)
{ }


Barrier::~Barrier()
{
    if (pthread_mutex_destroy(&mutex) != 0) {
        fprintf(stderr, "[[Barrier]] error on pthread_mutex_destroy");
        exit(1);
    }
    if (pthread_cond_destroy(&cv) != 0){
        fprintf(stderr, "[[Barrier]] error on pthread_cond_destroy");
        exit(1);
    }
}


void Barrier::barrier()
{
    if (pthread_mutex_lock(&mutex) != 0){
        fprintf(stderr, "[[Barrier]] error on pthread_mutex_lock");
        exit(1);
    }
    if (++count < numThreads) {
        if (pthread_cond_wait(&cv, &mutex) != 0){
            fprintf(stderr, "[[Barrier]] error on pthread_cond_wait");
            exit(1);
        }
    } else {
        count = 0;
        if (pthread_cond_broadcast(&cv) != 0) {
            fprintf(stderr, "[[Barrier]] error on pthread_cond_broadcast");
            exit(1);
        }
    }
    if (pthread_mutex_unlock(&mutex) != 0) {
        fprintf(stderr, "[[Barrier]] error on pthread_mutex_unlock");
        exit(1);
    }
}
```

5 Makefile

```
CC=g++
CXX=g++
RANLIB=ranlib

LIBSRC=MapReduceFramework.cpp Barrier.cpp
LIBOBJ=MapReduceFramework.o Barrier.o

INCS=-I.
CFLAGS = -Wall -std=c++11 -g -pthread $(INCS)
CXXFLAGS = -Wall -std=c++11 -g -pthread $(INCS)

LIBMAPREDUCE = libMapReduceFramework.a
TARGETS = $(LIBMAPREDUCE)

TAR=tar
TARFLAGS=-cvf
TARNAME=ex3.tar
TARSRCS=$(LIBSRC) Makefile README Barrier.h

all: $(TARGETS)

$(TARGETS): $(LIBOBJ)
	$(AR) $(ARFLAGS) $@ $^
	$(RANLIB) $@

clean:
	$(RM) $(TARGETS) $(LIBUTHREADS) $(OBJ) $(LIBOBJ) *~ *core

depend:
	makedepend -- $(CFLAGS) -- $(SRC) $(LIBSRC)

tar:
	$(TAR) $(TARFLAGS) $(TARNAME) $(TARSRCS)
```

6 MapReduceFramework.cpp

```cpp
#include <pthread.h>
#include <atomic>
#include <iostream>
#include <utility>
#include <algorithm>
#include <semaphore.h>
#include "MapReduceFramework.h"
#include "Barrier.h"

using std::cout;
using std::endl;
using std::vector;
using std::pair;

struct JobContext;
struct ThreadContext;

// Comparator.
bool comparePtrToPair(IntermediatePair a, IntermediatePair b)
{ return a.first->operator<(*b.first); }

/**
 * @brief Context of a thread.
 */
typedef struct ThreadContext
{
    // Job that spawned the thread.
    JobContext *jobContext;
    // Number of thread given upon initialization.
    int threadNum;
    // Input vector given by client.
    InputVec inputVec;
    // Intermediate vector.
    vector<IntermediatePair> *interVec;
    // Semaphore.
    sem_t *sem;
    // Atomic counter.
    std::atomic<int> *atomicCounter;
    // Output vector.
    OutputVec *outputVec;
    // Client.
    const MapReduceClient *client;
    // Barrier.
    Barrier *barrier;
    // Mutex.
    pthread_mutex_t *mutex;

    // Ctor.
    ThreadContext(JobContext *_jobContext, int _threadNum, const InputVec &_inputVec, sem_t *_sem,
                  std::atomic<int> *_atomicCounter, OutputVec &_outputVec, const MapReduceClient *_client,
                  Barrier *_barrier, pthread_mutex_t *_mutex) :
            jobContext(_jobContext), threadNum(_threadNum), inputVec(_inputVec),
            interVec(new IntermediateVec()), sem(_sem), atomicCounter(_atomicCounter),
            outputVec(&_outputVec), client(_client), barrier(_barrier), mutex(_mutex)
    {}

} ThreadContext;

/**
```

```
 * @brief Context of a job.
 */
typedef struct JobContext
{
    // Vector of threads alive within this job.
    vector<ThreadContext *> *threads{};
    pthread_t *threadArr;
    // State of the current job.
    JobState *state;

    // Ctor for a JobContext instance. Receives _threads as pointer.
    JobContext(vector<ThreadContext *> *_threads, pthread_t *_threadArr) : threads(_threads), threadArr(_threadArr)
    {
        // Inits state.
        state = new JobState();
        // Sets state.
        state->stage = UNDEFINED_STAGE;
        state->percentage = 0;
    }

    ~JobContext()
    {
        if (!threads->empty())
        {
            delete threads->back()->barrier;
            delete threads->back()->mutex;
            delete threads->back()->atomicCounter;
            delete threads->back()->sem;
        }
        for (ThreadContext *tc:*threads)
        {
            delete tc->interVec;
            delete tc;
        }
        delete threads;
        delete[] threadArr;
        delete state;
    }
} JobContext;


/**
 * @brief Emits pairs into context = intermediate vector.
 */
void emit2(K2 *key, V2 *value, void *context)
{
    auto *interVec = (vector<IntermediatePair> *) context;
    auto p = IntermediatePair(key, value);
    interVec->push_back(p);
}

void emit3(K3 *key, V3 *value, void *context)
{
    auto curr_context = (ThreadContext *) context;
    auto p = OutputPair(key, value);
    curr_context->outputVec->push_back(p);

}

// Mapping
void mapPhase(ThreadContext *context)
{
    context->jobContext->state->stage = MAP_STAGE;
    vector<IntermediatePair> *interVec = context->interVec;
    int oldValue;
    // Use atomic to avoid race conditions.
    while ((unsigned long int) context->atomicCounter->load() < context->inputVec.size())
    {
```

```
128             oldValue = (*(context->atomicCounter))++;
129             InputPair currPair = context->inputVec.at(static_cast<unsigned int>(oldValue));
130             // Map each pair.
131             context->client->map(currPair.first, currPair.second, interVec);
132             // Update percentage.
133             context->jobContext->state->percentage = ((oldValue + 1) / (float) context->inputVec.size() * 100);
134         }
135     }
136
137     // Sorting.
138     void sortPhase(ThreadContext *context)
139     {
140         std::sort(context->interVec->begin(), context->interVec->end(), comparePtrToPair);
141     }
142
143     void shufflePhase(vector<IntermediateVec> *reduceQueue, ThreadContext *context)
144     {
145         context->jobContext->state->stage = REDUCE_STAGE;
146
147         // Throw all pairs into a single vector.
148         IntermediateVec newInter;
149         for (ThreadContext *tc:*context->jobContext->threads)
150         {
151             newInter.insert(newInter.begin(), tc->interVec->begin(), tc->interVec->end());
152         }
153         std::sort(newInter.begin(), newInter.end(), comparePtrToPair);
154
155         // Go over all pairs.
156         K2 *k2max = newInter.back().first;
157         IntermediatePair *currPair = &newInter.back();
158         IntermediateVec currVec;
159         while (!newInter.empty())
160         {
161             if (*currPair->first < *k2max)
162             {
163                 if (pthread_mutex_lock(context->mutex) != 0)
164                 {
165                     fprintf(stderr, "Shuffle: error on pthread_mutex_lock");
166                     exit(1);
167                 }
168                 reduceQueue->push_back(currVec);
169                 sem_post(context->sem);
170                 if (pthread_mutex_unlock(context->mutex) != 0)
171                 {
172                     fprintf(stderr, "Shuffle: error on pthread_mutex_unlock");
173                     exit(1);
174                 }
175                 currVec = IntermediateVec();
176                 k2max = currPair->first;
177             }
178             currVec.push_back(*currPair);
179             newInter.pop_back();
180             currPair = &newInter.back();
181         }
182
183
184         if (pthread_mutex_lock(context->mutex) != 0)
185         {
186             fprintf(stderr, "Shuffle: error on pthread_mutex_lock");
187             exit(1);
188         }
189         if (!currVec.empty())
190         {
191             reduceQueue->push_back(currVec);
192             sem_post(context->sem);
193
194         }
195         if (pthread_mutex_unlock(context->mutex) != 0)
```

```
196     {
197         fprintf(stderr, "Shuffle: error on pthread_mutex_unlock");
198         exit(1);
199     }
200 
201 }
202 
203 // Reducing
204 void reducePhase(vector<IntermediateVec> *reduceQueue, int reduceSize, ThreadContext *context)
205 {
206     while (!reduceQueue->empty())
207     {
208         sem_wait(context->sem);
209 
210         if (pthread_mutex_lock(context->mutex) != 0)
211         {
212             fprintf(stderr, "Reduce: error on pthread_mutex_lock");
213             exit(1);
214         }
215 
216         if (!reduceQueue->empty())
217         {
218             context->client->reduce(&reduceQueue->back(), context);
219             reduceQueue->pop_back();
220             context->jobContext->state->percentage = (reduceSize - reduceQueue->size()) / (float) reduceSize * 100;
221         }
222         else
223         {
224             sem_post(context->sem);
225         }
226 
227         if (pthread_mutex_unlock(context->mutex) != 0)
228         {
229             fprintf(stderr, "Reduce: error on pthread_mutex_lock");
230             exit(1);
231         }
232     }
233 }
234 
235 /**
236  * @brief The main function of each thread.
237  */
238 void threadMapReduce(ThreadContext *context)
239 {
240     mapPhase(context);
241     sortPhase(context);
242     context->barrier->barrier(); // waiting for unlock.
243     auto *reduceQueue = new vector<IntermediateVec>();
244     if (context->threadNum == 0)
245     {
246         shufflePhase(reduceQueue, context);
247     }
248     reducePhase(reduceQueue, reduceQueue->size(), context);
249     delete reduceQueue;
250 }
251 
252 JobHandle startMapReduceJob(const MapReduceClient &client, const InputVec &inputVec, OutputVec &outputVec,
253                             int multiThreadLevel)
254 {
255     auto *sem = new sem_t();
256 
257     // atomic counter to be used as input vec index.
258     auto *atomic_counter = new std::atomic<int>(0);
259     auto *threads = new vector<ThreadContext *>();
260     auto *threadArr = new pthread_t[multiThreadLevel];
261     auto *jobContext = new JobContext(threads, threadArr);
262     auto *barrier = new Barrier(multiThreadLevel);
263     auto *mutex = new pthread_mutex_t(PTHREAD_MUTEX_INITIALIZER);
```

```
264     for (int i = 0; i < multiThreadLevel; ++i)
265     {
266         ThreadContext *context = new ThreadContext(jobContext, i, inputVec, sem,
267                                                    atomic_counter, outputVec, &client,
268                                                    barrier, mutex);
269         threads->push_back(context);
270         pthread_create(threadArr + i, nullptr, (void *(*)(void *)) threadMapReduce, context);
271     }
272     return jobContext;
273 }
274
275 void waitForJob(JobHandle job)
276 {
277     auto *jobContext = (JobContext *) job;
278     while (jobContext->state->stage != REDUCE_STAGE || jobContext->state->percentage < 100)
279     {
280     }
281 }
282
283 void getJobState(JobHandle job, JobState *state)
284 {
285     auto *jc = (JobContext *) job;
286     *state = *jc->state;
287 }
288
289 void closeJobHandle(JobHandle job)
290 {
291     waitForJob(job);
292     auto jobContext = (JobContext *) job;
293     for (int i = 0; i < (int)jobContext->threads->size(); ++i)
294     {
295         pthread_join(jobContext->threadArr[i], nullptr);
296     }
297     delete jobContext;
298 }
```