Contents

1 Basic Test Results

```

------------------

Extracting tar...

------------------

Checking for required files...

------------------

Validating users...
Got the following: ['e342791191', 'archsak']
Make sure the usernames are correct

------------------

Running 'make' test
Passed basic make test

------------------

Running compilation test
Passed basic compile test
For your convenience, we ran your library and printed the output here.
    stage 1, 100.000000%
    stage 3, 0.000000%
    stage 3, 33.333332%
    stage 3, 47.619049%
    stage 3, 66.666664%
    stage 3, 85.714287%
    stage 3, 100.000000%
    Done!
    The character w appeared 1 time
    The character u appeared 2 times
    The character t appeared 5 times
    The character s appeared 7 times
    The character r appeared 6 times
    The character o appeared 4 times
    The character n appeared 4 times
    The character m appeared 1 time
    The character l appeared 3 times
    The character i appeared 8 times
    The character h appeared 3 times
    The character g appeared 2 times
    The character f appeared 2 times
    The character e appeared 6 times
    The character d appeared 3 times
    The character c appeared 4 times
    The character b appeared 1 time
    The character a appeared 7 times
    The character T appeared 1 time
    The character M appeared 1 time




###########################
# Passed pre-submit tests #
```

```
###########################
Note: There may be more outputs above with errors.
 Passing this test does not mean your library ran properly.
```

2 README

```
e342791191, archsak
Elsa Sebagh (342791191), Aharon Saksonov (207600164)
EX: 3

FILES:
MapReduceFramework.cpp
JobContext.cpp
JobContext.h
Barrier.cpp
Barrier.h
Makefile


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

```
#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 JobContext.h

```
//
// Created by user on 21/06/2024.
//

#ifndef MAPREDUCEFRAMEWORK_JOBCONTEXT_H
#define MAPREDUCEFRAMEWORK_JOBCONTEXT_H
#include "MapReduceFramework.h"
#include "Barrier.h"

#include <pthread.h>
#include <iostream>
#include <memory>
#include <atomic>
#include <vector>
#include <utility>
#include <set>
#include <algorithm>
#include <semaphore.h>

using namespace std;

struct ThreadContext;

struct K2Comparator {
    bool operator()(const K2* key1, const K2* key2) const {
        return (*key1)<(*key2);
    }
};


class JobContext
{
 public:
  JobContext (const MapReduceClient &client, const InputVec &inputVec, OutputVec &outputVec, int multiThreadLevel);
  ~JobContext ();
  void operator= (const JobContext &other);

  void waitForJob ();
  JobState getJobState ();
  void addThread (int id);
  InputVec getInputVec ();
  OutputVec getOutputVec ();
  Barrier* getBarrier ();
  long unsigned int getInputLength ();
  void setJobState (JobState state);
  const MapReduceClient &getClient () const;

  pthread_mutex_t jobMutex;
  pthread_cond_t jobCond;
  std::set<K2*, K2Comparator> getUniqueKeySet();
  std::vector<IntermediateVec> getIntermediateVectors();
  std::vector<IntermediateVec> getShuffledVectors();
  sem_t* getShuffleSemaphore();
    void insertToShuffledVectors(IntermediateVec     vectors);
    void insertToUniqueKeySet(K2* unique_key);
    void insertToIntermediateVectors(IntermediateVec intermediate_vector);
    void insertToOutputVec(K3* key, V3* value);
    void setJobFinished();
    int getMultiThreadLevel();
```

```cpp




 private:
  void setJobState (stage_t stage, float percentage, bool finished = false);
  const MapReduceClient &client;
  const InputVec &inputVec;
  OutputVec &outputVec;
  bool isWaitingForJob;
  int multiThreadLevel;
  long unsigned int inputLength;
  std::vector <pthread_t> threads;
  std::vector<ThreadContext *> threadContexts;

  JobState state;
  bool jobFinished;
  std::atomic<long unsigned int> atomic;
  std::set<K2*, K2Comparator> uniqueKeySet;
  std::vector<std::vector<std::pair<K2*, V2*>>> intermediateVectors;
  std::vector<std::vector<std::pair<K2*, V2*>>> shuffledVectors;
  Barrier *barrier;
  sem_t shuffleSemaphore;  // Semaphore to control shuffle synchronization

};

struct ThreadContext {
    int threadId;
    std::unique_ptr<std::vector<std::pair<K2*, V2*>>> intermediateVector;
    std::atomic<long unsigned int>& atomic;
    JobContext* jobContext;

    ThreadContext(int id, std::atomic<long unsigned int>& atomic, JobContext* jobContext)
        : threadId(id), intermediateVector(new std::vector<std::pair<K2*,
                                           V2*>>()), atomic
                                           (atomic),
                                           jobContext(jobContext)
          {}

};

#endif //MAPREDUCEFRAMEWORK_JOBCONTEXT_H
```

6 JobContext.cpp

```
#include "JobContext.h"
#include <pthread.h>
#include <iostream>
#include <algorithm>

using namespace std;

void jobSystemError (string text)
{
  std::cout << "system error: " << text << std::endl;
  exit (1);
}

void *runThread (void *context)
{
  auto *castContext = static_cast<ThreadContext *>(context);
  auto &jobContext = castContext->jobContext;

  /**
   *------------------------------ MAP PHASE -------------------------------
   */
  long unsigned int old_value = castContext->atomic.fetch_add (1);

  while (old_value < jobContext->getInputLength ())
  {
//    If this is the first iteration, set the job state to 0 - we're
//    entering map stage
    if (old_value == 0)
    {
      jobContext->setJobState ({MAP_STAGE, 0});
    }
//    Map over the input we got
    jobContext->getClient ().map (jobContext->getInputVec ()[old_value].first,
                                  jobContext->getInputVec ()[old_value].second,
                                  context);
//    Update state
    float result = static_cast<float>(100.0f
                                      * static_cast<float>(castContext->atomic.load ())
                                      /
                                      jobContext->getInputLength ());
    if (castContext->atomic.load ()
        >= jobContext->getInputLength () - 1)
    {
      jobContext->setJobState ({MAP_STAGE, 100.0f});
      break;
    }
    else if (old_value < jobContext->getInputLength () - 1)
    {
      old_value = castContext->atomic.fetch_add (1);
      jobContext->setJobState ({MAP_STAGE, result});
    }
  }

  /**
   * Sorting
   */
  if (!castContext->intermediateVector->empty ())
  {
//     Sorting the keys
```

```
    std::sort (castContext->intermediateVector->begin (),
               castContext->intermediateVector->end (),
               [] (const IntermediatePair &a, const IntermediatePair &b)
               {
                   return *a.first < *b.first;
               });
    jobContext->insertToIntermediateVectors (*(castContext->intermediateVector));

    for (size_t i = 0; i < castContext->intermediateVector->size (); i++)
    {
      jobContext->insertToUniqueKeySet ((*castContext->intermediateVector)[i].first);

    }
  }
  jobContext->getBarrier ()->barrier (); // Wait for everyone

  /**
   * ------------------------------ SHUFFLE PHASE ------------------------------
   */

  if (castContext->threadId == 0)
  {
    castContext->atomic.exchange (0);
    jobContext->setJobState ({SHUFFLE_STAGE, 0});
    auto uniqueKeySet = jobContext->getUniqueKeySet ();

    // Convert the set to a vector
    std::vector < K2 *
    > uniqueKeySetVector (uniqueKeySet.begin (), uniqueKeySet.end ());

    // Sort the vector of pointers
    std::sort (uniqueKeySetVector.begin (), uniqueKeySetVector.end (),
               [] (const K2 *a, const K2 *b)
               {
                   return *b < *a;
               });
    auto intermediateVectors = jobContext->getIntermediateVectors ();
    for (K2 *key: uniqueKeySetVector)
    {
      IntermediateVec key_vector;
      for (auto &vector: intermediateVectors)  // Note the use of '&' here
      {
        while (!vector.empty () && !(*(vector.back ().first) < *key || *key <
                                                                       *(vector
                                                                           .back ().first)))
        {
          key_vector.push_back (vector.back ());
          vector.pop_back ();
          castContext->atomic.fetch_add (1);
          float result = static_cast<float>(100.0f
                                            * static_cast<float>(castContext->atomic.load ())
                                            /
                                            jobContext->getInputLength ());
          jobContext->setJobState ({SHUFFLE_STAGE, result});
        }
      }
      jobContext->insertToShuffledVectors (key_vector);
    }

    jobContext->setJobState ({SHUFFLE_STAGE, 100.0f});
    for (int i = 1; i < jobContext->getMultiThreadLevel(); ++i) {
      sem_post(jobContext->getShuffleSemaphore());
    }
    castContext->atomic.exchange (0);
  }
  else
  {
    sem_wait (jobContext->getShuffleSemaphore ());  // Wait until shuffle is done
  }
  /**
   * ------------------------------ REDUCE PHASE ------------------------------
   */
  old_value = castContext->atomic.fetch_add (1);

  while (old_value < jobContext->getShuffledVectors ().size ())
  {
//    If this is the first iteration, set the job state to 0 - we're
//    entering reduce stage
    if (old_value == 0)
    {
      jobContext->setJobState ({REDUCE_STAGE, 0});
    }
//    Reduce over the intermediate we got
    jobContext->getClient ().reduce
        (&(jobContext->getShuffledVectors ()[old_value]),
         context);
//    Update state
    float result = static_cast<float>(100.0f
                                      * static_cast<float>(castContext->atomic.load ())
                                      /
                                      jobContext->getShuffledVectors ().size ());
    if (castContext->atomic.load ()
        >= jobContext->getShuffledVectors ().size () - 1)
    {
      jobContext->setJobState ({REDUCE_STAGE, 100.0f});
      jobContext->setJobFinished();
      break;
    }
    else if (old_value < jobContext->getShuffledVectors ().size () - 1)
    {
      old_value = castContext->atomic.fetch_add (1);
      jobContext->setJobState ({REDUCE_STAGE, result});
    }
  }
  return nullptr;
}

JobContext::JobContext (const MapReduceClient &client, const InputVec &inputVec,
                        OutputVec &outputVec, int multiThreadLevel)
    : client (client) ,inputVec (inputVec), outputVec (outputVec), isWaitingForJob(false),
      multiThreadLevel (multiThreadLevel), state ({UNDEFINED_STAGE, 0}),
      jobFinished (false), atomic (0)
{
  pthread_mutex_init (&jobMutex, nullptr);
  pthread_cond_init (&jobCond, nullptr);
  inputLength = inputVec.size ();
  barrier = new Barrier (multiThreadLevel);
  sem_init (&shuffleSemaphore, 0, 0);  // Initialize semaphore with value 0

//  TODO Save this in bits somehow
  for (int i = 0; i < multiThreadLevel; i++)
  {
    addThread (i);
  }

}

JobContext::~JobContext ()
{
  for (auto context: threadContexts)
  {
    context->intermediateVector->clear ();
    delete context;
  }
  for (auto it = threads.begin (); it != threads.end ();)
  {
    pthread_cancel (*it);  // Cancel the thread
    // Erase the thread from the vector and advance the iterator
    it = threads.erase (it);
  }
  pthread_mutex_destroy (&jobMutex);
  pthread_cond_destroy (&jobCond);
  sem_destroy (&shuffleSemaphore);
  delete barrier;
}
//void JobContext::operator= (const JobContext &other)
//{
//  client = other.client;
//  inputVec = other.inputVec;
//  outputVec = other.outputVec;
//  multiThreadLevel = other.multiThreadLevel;
//  jobFinished = other.jobFinished;
//  atomic_length = other.atomic_length;
//  inputLength = other.inputLength;
//  state = other.state;
//  threads = other.threads;
//  threadContexts = other.threadContexts;
//  jobMutex = other.jobMutex;
//  jobCond = other.jobCond;
//}

void JobContext::waitForJob ()
{
//  if(isWaitingForJob) return;
  while (!jobFinished)
  {
    pthread_cond_wait(&jobCond, &jobMutex);
  }

  for(int i = 0; i < multiThreadLevel; i++){
    if(pthread_join(threads[i], nullptr) != 0){
      jobSystemError ("Error joining thread");
    }
  }
  isWaitingForJob = true;

}


JobState JobContext::getJobState ()
{
  if (state.stage == MAP_STAGE)
  {
    float result = static_cast<float>(atomic) / inputLength * 100.0f;
    if (result > 100) result = 100.0f;
    state.percentage = result;
  }
  return state;
}

void JobContext::addThread (int id)
{
  pthread_t thread;
  auto *context = new ThreadContext (id, atomic, this);

  pthread_attr_t attr;
  pthread_attr_init (&attr);
  if (pthread_create (&thread, &attr, runThread, static_cast<void *>
  (context)) != 0)
  {
    jobSystemError ("Could not create thread");
  };
  pthread_attr_destroy (&attr);
  threadContexts.push_back (context);
  threads.push_back (thread);
}

void JobContext::insertToShuffledVectors (IntermediateVec vectors)
{
  pthread_mutex_lock (&jobMutex);
  shuffledVectors.push_back (vectors);
  pthread_cond_broadcast (&jobCond);
  pthread_mutex_unlock (&jobMutex);
}

InputVec JobContext::getInputVec ()
{
  return inputVec;
}

OutputVec JobContext::getOutputVec ()
{
  return outputVec;
}

long unsigned int JobContext::getInputLength ()
{
  return inputLength;
}

const MapReduceClient &JobContext::getClient () const
{
  return client;
}

Barrier *JobContext::getBarrier ()
{
  return barrier;
}

std::vector <IntermediateVec> JobContext::getShuffledVectors ()
{
  return shuffledVectors;
}

std::vector <IntermediateVec> JobContext::getIntermediateVectors ()
{
  return intermediateVectors;
}

void JobContext::setJobState (JobState state)
{
  pthread_mutex_lock (&jobMutex);
  this->state = state;
  pthread_cond_broadcast (&jobCond);
  pthread_mutex_unlock (&jobMutex);
}

void JobContext::insertToUniqueKeySet (K2 *uniqueKey)
{
  pthread_mutex_lock (&jobMutex);
  uniqueKeySet.insert (uniqueKey);
  pthread_cond_broadcast (&jobCond);
  pthread_mutex_unlock (&jobMutex);
}

std::set<K2 *, K2Comparator> JobContext::getUniqueKeySet ()
{
  return uniqueKeySet;
}

sem_t *JobContext::getShuffleSemaphore ()
```

```
{
  return &shuffleSemaphore;
}

void
JobContext::insertToIntermediateVectors (IntermediateVec intermediateVector)
{
  pthread_mutex_lock (&jobMutex);
  intermediateVectors.push_back (intermediateVector);
  pthread_cond_broadcast (&jobCond);
  pthread_mutex_unlock (&jobMutex);
}

void JobContext::insertToOutputVec (K3 *key, V3 *value)
{
  pthread_mutex_lock (&jobMutex);
  outputVec.push_back (std::make_pair (key, value));
  pthread_cond_broadcast (&jobCond);
  pthread_mutex_unlock (&jobMutex);
}

void JobContext::setJobFinished ()
{
  pthread_mutex_lock (&jobMutex);
  jobFinished = true;
  pthread_cond_broadcast (&jobCond);
  pthread_mutex_unlock (&jobMutex);
}

int JobContext::getMultiThreadLevel ()
{
  return multiThreadLevel;
}
```

7 Makefile

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

LIBSRC=MapReduceFramework.cpp JobContext.cpp JobContext.h Barrier.cpp Barrier.h Makefile
LIBOBJ=$(LIBSRC:.cpp=.o)

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

MAPREDUCELIB = libMapReduceFramework.a
TARGETS = $(MAPREDUCELIB)

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

all: $(TARGETS)

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

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

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

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

8 MapReduceFramework.cpp

```
#include <pthread.h>
#include "MapReduceFramework.h"
#include "JobContext.h"
#include <iostream>
#include <vector>
#include <string>
#include <map>

using namespace std;

std::map<JobHandle, JobContext *> jobs;

void systemError (string text)
{
  std::cout << "system error: " << text << std::endl;
  exit (1);
}

// Context is whatever we want it to be - we get it from map, and we're the
// ones to call map.
// Should probably contain the input, output and intermediate pointers
void emit2 (K2 *key, V2 *value, void *context)
{
//  Add key and value to the intermediate vector of the calling thread

  auto *castContext = static_cast<ThreadContext *>(context);
  pthread_mutex_lock (&castContext->jobContext->jobMutex);
  castContext->intermediateVector->push_back (std::make_pair (key,
                                                              value));
  pthread_mutex_unlock (&castContext->jobContext->jobMutex);

}

void emit3 (K3 *key, V3 *value, void *context)
{
    auto* castContext = static_cast<ThreadContext*>(context);
    castContext->jobContext->insertToOutputVec (key, value);

}

JobHandle startMapReduceJob (const MapReduceClient &client,
                             const InputVec &inputVec, OutputVec &outputVec,
                             int multiThreadLevel)
{
  JobContext *job = new JobContext (client, inputVec, outputVec,
                                    multiThreadLevel);
  JobHandle jobHandle = new JobHandle ();
  jobs[jobHandle] = job;
  return jobHandle;
}

void waitForJob (JobHandle job)
{
  if (jobs.find (job) == jobs.end ())
  {
    systemError ("Job not found");
  }
  jobs[job]->waitForJob ();
}
```

```
void getJobState (JobHandle job, JobState *state)
{
  auto it = jobs.find (job);
  if (it == jobs.end ())
  {
    systemError ("Job not found");
  }
  *state = it->second->getJobState ();
}

void closeJobHandle (JobHandle job)
{
  {
//    std::lock_guard<std::mutex> lock(jobsMutex);
    auto it = jobs.find (job);
    if (it == jobs.end ())
    {
      systemError ("Job not found");
    }
    waitForJob (job);
    delete it->second;
    jobs.erase (it);
  }
}
```