

Contents

1	Basic Test Results	2
2	README	4
3	Barrier.h	5
4	Barrier.cpp	6
5	JobContext.h	7
6	JobContext.cpp	9
7	Makefile	15
8	MapReduceFramework.cpp	16

1 Basic Test Results

```
1 -----
2
3
4 Extracting tar...
5
6 -----
7
8 Checking for required files...
9
10 -----
11
12 Validating users...
13 Got the following: ['e342791191', 'archsak']
14 Make sure the usernames are correct
15
16 -----
17
18 Running 'make' test
19 Passed basic make test
20
21 -----
22
23 Running compilation test
24 Passed basic compile test
25 For your convenience, we ran your library and printed the output here.
26     stage 1, 100.000000%
27     stage 3, 0.000000%
28     stage 3, 33.333332%
29     stage 3, 47.619049%
30     stage 3, 66.666664%
31     stage 3, 85.714287%
32     stage 3, 100.000000%
33 Done!
34 The character w appeared 1 time
35 The character u appeared 2 times
36 The character t appeared 5 times
37 The character s appeared 7 times
38 The character r appeared 6 times
39 The character o appeared 4 times
40 The character n appeared 4 times
41 The character m appeared 1 time
42 The character l appeared 3 times
43 The character i appeared 8 times
44 The character h appeared 3 times
45 The character g appeared 2 times
46 The character f appeared 2 times
47 The character e appeared 6 times
48 The character d appeared 3 times
49 The character c appeared 4 times
50 The character b appeared 1 time
51 The character a appeared 7 times
52 The character T appeared 1 time
53 The character M appeared 1 time
54
55
56
57
58 #####
59 # Passed pre-submit tests #
```

```
60 #####
61 Note: There may be more outputs above with errors.
62 Passing this test does not mean your library ran properly.
```

2 README

```
1  e342791191, archsak
2  Elsa Sebagh (342791191), Aharon Saksonov (207600164)
3  EX: 3
4
5  FILES:
6  MapReduceFramework.cpp
7  JobContext.cpp
8  JobContext.h
9  Barrier.cpp
10 Barrier.h
11 Makefile
12
13
14 REMARKS:
15
16
```

3 Barrier.h

```
1  #ifndef BARRIER_H
2  #define BARRIER_H
3  #include <pthread.h>
4
5  // a multiple use barrier
6
7  class Barrier {
8  public:
9      Barrier(int numThreads);
10     ~Barrier();
11     void barrier();
12
13 private:
14     pthread_mutex_t mutex;
15     pthread_cond_t cv;
16     int count;
17     int numThreads;
18 };
19
20 #endif //BARRIER_H
```

4 Barrier.cpp

```
1  #include "Barrier.h"
2  #include <cstdlib>
3  #include <stdio>
4
5  Barrier::Barrier(int numThreads)
6      : mutex(PTHREAD_MUTEX_INITIALIZER)
7      , cv(PTHREAD_COND_INITIALIZER)
8      , count(0)
9      , numThreads(numThreads)
10 { }
11
12
13 Barrier::~Barrier()
14 {
15     if (pthread_mutex_destroy(&mutex) != 0) {
16         fprintf(stderr, "[Barrier] error on pthread_mutex_destroy");
17         exit(1);
18     }
19     if (pthread_cond_destroy(&cv) != 0){
20         fprintf(stderr, "[Barrier] error on pthread_cond_destroy");
21         exit(1);
22     }
23 }
24
25
26 void Barrier::barrier()
27 {
28     if (pthread_mutex_lock(&mutex) != 0){
29         fprintf(stderr, "[Barrier] error on pthread_mutex_lock");
30         exit(1);
31     }
32     if (++count < numThreads) {
33         if (pthread_cond_wait(&cv, &mutex) != 0){
34             fprintf(stderr, "[Barrier] error on pthread_cond_wait");
35             exit(1);
36         }
37     } else {
38         count = 0;
39         if (pthread_cond_broadcast(&cv) != 0) {
40             fprintf(stderr, "[Barrier] error on pthread_cond_broadcast");
41             exit(1);
42         }
43     }
44     if (pthread_mutex_unlock(&mutex) != 0) {
45         fprintf(stderr, "[Barrier] error on pthread_mutex_unlock");
46         exit(1);
47     }
48 }
```

5 JobContext.h

```
1  //
2  // Created by user on 21/06/2024.
3  //
4
5  #ifndef MAPREDUCEFRAMEWORK_JOBCONTEXT_H
6  #define MAPREDUCEFRAMEWORK_JOBCONTEXT_H
7  #include "MapReduceFramework.h"
8  #include "Barrier.h"
9
10 #include <pthread.h>
11 #include <iostream>
12 #include <memory>
13 #include <atomic>
14 #include <vector>
15 #include <utility>
16 #include <set>
17 #include <algorithm>
18 #include <semaphore.h>
19
20 using namespace std;
21
22 struct ThreadContext;
23
24 struct K2Comparator {
25     bool operator()(const K2* key1, const K2* key2) const {
26         return (*key1) < (*key2);
27     }
28 };
29
30
31 class JobContext
32 {
33 public:
34     JobContext (const MapReduceClient &client, const InputVec &inputVec, OutputVec &outputVec, int multiThreadLevel);
35     ~JobContext ();
36     void operator= (const JobContext &other);
37
38     void waitForJob ();
39     JobState getJobState ();
40     void addThread (int id);
41     InputVec getInputVec ();
42     OutputVec getOutputVec ();
43     Barrier* getBarrier ();
44     long unsigned int getInputLength ();
45     void setJobState (JobState state);
46     const MapReduceClient &getClient () const;
47
48     pthread_mutex_t jobMutex;
49     pthread_cond_t jobCond;
50     std::set<K2*, K2Comparator> getUniqueKeySet();
51     std::vector<IntermediateVec> getIntermediateVectors();
52     std::vector<IntermediateVec> getShuffledVectors();
53     sem_t* getShuffleSemaphore();
54     void insertToShuffledVectors(IntermediateVec vectors);
55     void insertToUniqueKeySet(K2* unique_key);
56     void insertToIntermediateVectors(IntermediateVec intermediate_vector);
57     void insertToOutputVec(K3* key, V3* value);
58     void setJobFinished();
59     int getMultiThreadLevel();
```

```

60
61
62
63
64 private:
65     void setJobState (stage_t stage, float percentage, bool finished = false);
66     const MapReduceClient &client;
67     const InputVec &inputVec;
68     OutputVec &outputVec;
69     bool isWaitingForJob;
70     int multiThreadLevel;
71     long unsigned int inputLength;
72     std::vector<pthread_t> threads;
73     std::vector<ThreadContext *> threadContexts;
74
75     JobState state;
76     bool jobFinished;
77     std::atomic<long unsigned int> atomic;
78     std::set<K2*, K2Comparator> uniqueKeySet;
79     std::vector<std::vector<std::pair<K2*, V2*>>> intermediateVectors;
80     std::vector<std::vector<std::pair<K2*, V2*>>> shuffledVectors;
81     Barrier *barrier;
82     sem_t shuffleSemaphore; // Semaphore to control shuffle synchronization
83
84 };
85
86 struct ThreadContext {
87     int threadId;
88     std::unique_ptr<std::vector<std::pair<K2*, V2*>>> intermediateVector;
89     std::atomic<long unsigned int>& atomic;
90     JobContext* jobContext;
91
92     ThreadContext(int id, std::atomic<long unsigned int>& atomic, JobContext* jobContext)
93         : threadId(id), intermediateVector(new std::vector<std::pair<K2*,
94                                                     V2*>>()), atomic
95                                                     (atomic),
96                                                     jobContext(jobContext)
97     {}
98
99 };
100
101 #endif //MAPREDUCEFRAMEWORK_JOBCONTEXT_H

```


6 JobContext.cpp

```
1  #include "JobContext.h"
2  #include <pthread.h>
3  #include <iostream>
4  #include <algorithm>
5
6  using namespace std;
7
8  void jobSystemError (string text)
9  {
10     std::cout << "system error: " << text << std::endl;
11     exit (1);
12 }
13
14 void *runThread (void *context)
15 {
16     auto *castContext = static_cast<ThreadContext *>(context);
17     auto &jobContext = castContext->jobContext;
18
19     /**
20      *----- MAP PHASE -----
21      */
22     long unsigned int old_value = castContext->atomic.fetch_add (1);
23
24     while (old_value < jobContext->getInputLength ())
25     {
26         // If this is the first iteration, set the job state to 0 - we're
27         // entering map stage
28         if (old_value == 0)
29         {
30             jobContext->setJobState ({MAP_STAGE, 0});
31         }
32         // Map over the input we got
33         jobContext->getClient ().map (jobContext->getInputVec ()[old_value].first,
34                                     jobContext->getInputVec ()[old_value].second,
35                                     context);
36         // Update state
37         float result = static_cast<float>(100.0f
38                                     * static_cast<float>(castContext->atomic.load ())
39                                     /
40                                     jobContext->getInputLength ());
41         if (castContext->atomic.load ()
42             >= jobContext->getInputLength () - 1)
43         {
44             jobContext->setJobState ({MAP_STAGE, 100.0f});
45             break;
46         }
47         else if (old_value < jobContext->getInputLength () - 1)
48         {
49             old_value = castContext->atomic.fetch_add (1);
50             jobContext->setJobState ({MAP_STAGE, result});
51         }
52     }
53
54     /**
55      * Sorting
56      */
57     if (!castContext->intermediateVector->empty ())
58     {
59         // Sorting the keys
```

```

60     std::sort (castContext->intermediateVector->begin (),
61               castContext->intermediateVector->end (),
62               [] (const IntermediatePair &a, const IntermediatePair &b)
63               {
64                   return *a.first < *b.first;
65               });
66     jobContext->insertToIntermediateVectors (*(castContext->intermediateVector));
67
68     for (size_t i = 0; i < castContext->intermediateVector->size (); i++)
69     {
70         jobContext->insertToUniqueKeySet ((*castContext->intermediateVector)[i].first);
71     }
72 }
73 }
74 jobContext->getBarrier ()->barrier (); // Wait for everyone
75
76 /**
77 * ----- SHUFFLE PHASE -----
78 */
79
80 if (castContext->threadId == 0)
81 {
82     castContext->atomic.exchange (0);
83     jobContext->setJobState ({SHUFFLE_STAGE, 0});
84     auto uniqueKeySet = jobContext->getUniqueKeySet ();
85
86     // Convert the set to a vector
87     std::vector < K2 *
88     > uniqueKeySetVector (uniqueKeySet.begin (), uniqueKeySet.end ());
89
90     // Sort the vector of pointers
91     std::sort (uniqueKeySetVector.begin (), uniqueKeySetVector.end (),
92               [] (const K2 *a, const K2 *b)
93               {
94                   return *b < *a;
95               });
96     auto intermediateVectors = jobContext->getIntermediateVectors ();
97     for (K2 *key: uniqueKeySetVector)
98     {
99         IntermediateVec key_vector;
100         for (auto &vector: intermediateVectors) // Note the use of '&' here
101         {
102             while (!vector.empty () && !(*(vector.back ().first) < *key || *key <
103                                     *(vector
104                                     .back ().first)))
105             {
106                 key_vector.push_back (vector.back ());
107                 vector.pop_back ();
108                 castContext->atomic.fetch_add (1);
109                 float result = static_cast<float>(100.0f
110                                     * static_cast<float>(castContext->atomic.load ())
111                                     /
112                                     jobContext->getInputLength ());
113                 jobContext->setJobState ({SHUFFLE_STAGE, result});
114             }
115         }
116         jobContext->insertToShuffledVectors (key_vector);
117     }
118
119     jobContext->setJobState ({SHUFFLE_STAGE, 100.0f});
120     for (int i = 1; i < jobContext->getMultiThreadLevel(); ++i) {
121         sem_post(jobContext->getShuffleSemaphore());
122     }
123     castContext->atomic.exchange (0);
124 }
125 else
126 {
127     sem_wait (jobContext->getShuffleSemaphore ()); // Wait until shuffle is done

```

```

128     }
129     /**
130     * ----- REDUCE PHASE -----
131     */
132     old_value = castContext->atomic.fetch_add (1);
133
134     while (old_value < jobContext->getShuffledVectors ().size ())
135     {
136         // If this is the first iteration, set the job state to 0 - we're
137         // entering reduce stage
138         if (old_value == 0)
139         {
140             jobContext->setJobState ({REDUCE_STAGE, 0});
141         }
142         // Reduce over the intermediate we got
143         jobContext->getClient ().reduce
144             (&(jobContext->getShuffledVectors ()[old_value]),
145              context);
146         // Update state
147         float result = static_cast<float>(100.0f
148                                     * static_cast<float>(castContext->atomic.load ())
149                                     /
150                                     jobContext->getShuffledVectors ().size ());
151         if (castContext->atomic.load ()
152             >= jobContext->getShuffledVectors ().size () - 1)
153         {
154             jobContext->setJobState ({REDUCE_STAGE, 100.0f});
155             jobContext->setJobFinished();
156             break;
157         }
158         else if (old_value < jobContext->getShuffledVectors ().size () - 1)
159         {
160             old_value = castContext->atomic.fetch_add (1);
161             jobContext->setJobState ({REDUCE_STAGE, result});
162         }
163     }
164     return nullptr;
165 }
166
167 JobContext::JobContext (const MapReduceClient &client, const InputVec &inputVec,
168                        OutputVec &outputVec, int multiThreadLevel)
169     : client (client), inputVec (inputVec), outputVec (outputVec), isWaitingForJob(false),
170     multiThreadLevel (multiThreadLevel), state ({UNDEFINED_STAGE, 0}),
171     jobFinished (false), atomic (0)
172 {
173     pthread_mutex_init (&jobMutex, nullptr);
174     pthread_cond_init (&jobCond, nullptr);
175     inputLength = inputVec.size ();
176     barrier = new Barrier (multiThreadLevel);
177     sem_init (&shuffleSemaphore, 0, 0); // Initialize semaphore with value 0
178
179     // TODO Save this in bits somehow
180     for (int i = 0; i < multiThreadLevel; i++)
181     {
182         addThread (i);
183     }
184
185 }
186
187 JobContext::~JobContext ()
188 {
189     for (auto context: threadContexts)
190     {
191         context->intermediateVector->clear ();
192         delete context;
193     }
194     for (auto it = threads.begin (); it != threads.end ();)
195     {

```

```

196     pthread_cancel (*it); // Cancel the thread
197     // Erase the thread from the vector and advance the iterator
198     it = threads.erase (it);
199 }
200 pthread_mutex_destroy (&jobMutex);
201 pthread_cond_destroy (&jobCond);
202 sem_destroy (&shuffleSemaphore);
203 delete barrier;
204 }
205 //void JobContext::operator= (const JobContext &other)
206 //{
207 //    client = other.client;
208 //    inputVec = other.inputVec;
209 //    outputVec = other.outputVec;
210 //    multiThreadLevel = other.multiThreadLevel;
211 //    jobFinished = other.jobFinished;
212 //    atomic_length = other.atomic_length;
213 //    inputLength = other.inputLength;
214 //    state = other.state;
215 //    threads = other.threads;
216 //    threadContexts = other.threadContexts;
217 //    jobMutex = other.jobMutex;
218 //    jobCond = other.jobCond;
219 //}
220
221 void JobContext::waitForJob ()
222 {
223     // if(isWaitingForJob) return;
224     while (!jobFinished)
225     {
226         pthread_cond_wait(&jobCond, &jobMutex);
227     }
228
229     for(int i = 0; i < multiThreadLevel; i++){
230         if(pthread_join(threads[i], nullptr) != 0){
231             jobSystemError ("Error joining thread");
232         }
233     }
234     isWaitingForJob = true;
235 }
236
237
238
239 JobState JobContext::getJobState ()
240 {
241     if (state.stage == MAP_STAGE)
242     {
243         float result = static_cast<float>(atomic) / inputLength * 100.0f;
244         if (result > 100) result = 100.0f;
245         state.percentage = result;
246     }
247     return state;
248 }
249
250 void JobContext::addThread (int id)
251 {
252     pthread_t thread;
253     auto *context = new ThreadContext (id, atomic, this);
254
255     pthread_attr_t attr;
256     pthread_attr_init (&attr);
257     if (pthread_create (&thread, &attr, runThread, static_cast<void *>
258 (context)) != 0)
259     {
260         jobSystemError ("Could not create thread");
261     };
262     pthread_attr_destroy (&attr);
263     threadContexts.push_back (context);

```

```

264     threads.push_back (thread);
265 }
266
267 void JobContext::insertToShuffledVectors (IntermediateVec vectors)
268 {
269     pthread_mutex_lock (&jobMutex);
270     shuffledVectors.push_back (vectors);
271     pthread_cond_broadcast (&jobCond);
272     pthread_mutex_unlock (&jobMutex);
273 }
274
275 InputVec JobContext::getInputVec ()
276 {
277     return inputVec;
278 }
279
280 OutputVec JobContext::getOutputVec ()
281 {
282     return outputVec;
283 }
284
285 long unsigned int JobContext::getInputLength ()
286 {
287     return inputLength;
288 }
289
290 const MapReduceClient &JobContext::getClient () const
291 {
292     return client;
293 }
294
295 Barrier *JobContext::getBarrier ()
296 {
297     return barrier;
298 }
299
300 std::vector <IntermediateVec> JobContext::getShuffledVectors ()
301 {
302     return shuffledVectors;
303 }
304
305 std::vector <IntermediateVec> JobContext::getIntermediateVectors ()
306 {
307     return intermediateVectors;
308 }
309
310 void JobContext::setJobState (JobState state)
311 {
312     pthread_mutex_lock (&jobMutex);
313     this->state = state;
314     pthread_cond_broadcast (&jobCond);
315     pthread_mutex_unlock (&jobMutex);
316 }
317
318 void JobContext::insertToUniqueKeySet (K2 *uniqueKey)
319 {
320     pthread_mutex_lock (&jobMutex);
321     uniqueKeySet.insert (uniqueKey);
322     pthread_cond_broadcast (&jobCond);
323     pthread_mutex_unlock (&jobMutex);
324 }
325
326 std::set<K2 *, K2Comparator> JobContext::getUniqueKeySet ()
327 {
328     return uniqueKeySet;
329 }
330
331 sem_t *JobContext::getShuffleSemaphore ()

```

```

332 {
333     return &shuffleSemaphore;
334 }
335
336 void
337 JobContext::insertToIntermediateVectors (IntermediateVec intermediateVector)
338 {
339     pthread_mutex_lock (&jobMutex);
340     intermediateVectors.push_back (intermediateVector);
341     pthread_cond_broadcast (&jobCond);
342     pthread_mutex_unlock (&jobMutex);
343 }
344
345 void JobContext::insertToOutputVec (K3 *key, V3 *value)
346 {
347     pthread_mutex_lock (&jobMutex);
348     outputVec.push_back (std::make_pair (key, value));
349     pthread_cond_broadcast (&jobCond);
350     pthread_mutex_unlock (&jobMutex);
351 }
352
353 void JobContext::setJobFinished ()
354 {
355     pthread_mutex_lock (&jobMutex);
356     jobFinished = true;
357     pthread_cond_broadcast (&jobCond);
358     pthread_mutex_unlock (&jobMutex);
359 }
360
361 int JobContext::getMultiThreadLevel ()
362 {
363     return multiThreadLevel;
364 }

```

7 Makefile

```
1  CC=g++
2  CXX=g++
3  RANLIB=ranlib
4
5  LIBSRC=MapReduceFramework.cpp JobContext.cpp JobContext.h Barrier.cpp Barrier.h Makefile
6  LIBOBJ=$(LIBSRC:.cpp=.o)
7
8  INCS=-I.
9  CFLAGS = -Wall -std=c++11 -g $(INCS)
10 CXXFLAGS = -Wall -std=c++11 -g $(INCS)
11
12 MAPREDUCELIB = libMapReduceFramework.a
13 TARGETS = $(MAPREDUCELIB)
14
15 TAR=tar
16 TARFLAGS=-cvf
17 TARNAME=ex3.tar
18 TARSRC=$(LIBSRC) Makefile README
19
20 all: $(TARGETS)
21
22 $(TARGETS): $(LIBOBJ)
23     $(AR) $(ARFLAGS) $@ $^
24     $(RANLIB) $@
25
26 clean:
27     $(RM) $(TARGETS) $(OSMLIB) $(OBJ) $(LIBOBJ) *~ *core
28
29 depend:
30     makedepend -- $(CFLAGS) -- $(SRC) $(LIBSRC)
31
32 tar:
33     $(TAR) $(TARFLAGS) $(TARNAME) $(TARSRC)
```

8 MapReduceFramework.cpp

```
1  #include <pthread.h>
2  #include "MapReduceFramework.h"
3  #include "JobContext.h"
4  #include <iostream>
5  #include <vector>
6  #include <string>
7  #include <map>
8
9  using namespace std;
10
11  std::map<JobHandle, JobContext *> jobs;
12
13  void systemError (string text)
14  {
15      std::cout << "system error: " << text << std::endl;
16      exit (1);
17  }
18
19  // Context is whatever we want it to be - we get it from map, and we're the
20  // ones to call map.
21  // Should probably contain the input, output and intermediate pointers
22  void emit2 (K2 *key, V2 *value, void *context)
23  {
24      // Add key and value to the intermediate vector of the calling thread
25
26      auto *castContext = static_cast<ThreadContext *>(context);
27      pthread_mutex_lock (&castContext->jobContext->jobMutex);
28      castContext->intermediateVector->push_back (std::make_pair (key,
29                                                                    value));
30      pthread_mutex_unlock (&castContext->jobContext->jobMutex);
31  }
32
33
34  void emit3 (K3 *key, V3 *value, void *context)
35  {
36      auto* castContext = static_cast<ThreadContext*>(context);
37      castContext->jobContext->insertToOutputVec (key, value);
38  }
39
40
41  JobHandle startMapReduceJob (const MapReduceClient &client,
42                              const InputVec &inputVec, OutputVec &outputVec,
43                              int multiThreadLevel)
44  {
45      JobContext *job = new JobContext (client, inputVec, outputVec,
46                                         multiThreadLevel);
47      JobHandle jobHandle = new JobHandle ();
48      jobs[jobHandle] = job;
49      return jobHandle;
50  }
51
52  void waitForJob (JobHandle job)
53  {
54      if (jobs.find (job) == jobs.end ())
55      {
56          systemError ("Job not found");
57      }
58      jobs[job]->waitForJob ();
59  }
```



```

60
61 void getJobState (JobHandle job, JobState *state)
62 {
63     auto it = jobs.find (job);
64     if (it == jobs.end ())
65     {
66         systemError ("Job not found");
67     }
68     *state = it->second->getJobState ();
69 }
70
71 void closeJobHandle (JobHandle job)
72 {
73     {
74         // std::lock_guard<std::mutex> lock(jobsMutex);
75         auto it = jobs.find (job);
76         if (it == jobs.end ())
77         {
78             systemError ("Job not found");
79         }
80         waitForJob (job);
81         delete it->second;
82         jobs.erase (it);
83     }
84 }
85
86
87

```