OS 2020 – Exercise 3
MapReduce - Multi-Threaded Programming

Supervisor – Yuval Jacoby

Due: 1.6.2020 (June 1st 2020)

As stated in the guidelines, the deadline will not be extended

Note: This exercise takes a lot of time. Start early!

High Level Overview

Performance is the major motivation for multi-threaded programming. Multiple processors can execute multiple threads at the same time and do the same amount of computations in less time than it will take a single processor.

Two challenges complicate multi-threaded programming:

1) In many cases it is difficult to split the big task into small parts that can run in parallel.
2) Running in multiple threads requires synchronisation and communication between threads. This introduces an overhead which without careful design can increase the total runtime significantly.

Over the years, several designs were proposed in order to solve these challenges. In this exercise we will implement one of these designs, named MapReduce.

MapReduce is used to parallelise tasks of a specific structure. Such tasks are defined by two functions, *map* and *reduce*, used as follows:

1) The input is given as a sequence of input elements.
2) (Map phase) The *map* function is applied to each input element, producing a sequence of intermediary elements.
3) (Sort/Shuffle phases) The intermediary elements are sorted into new sequences (more on this later).
4) (Reduce phase) The *reduce* function is applied to each of the sorted sequences of intermediary elements, producing a sequence of output elements.
5) The output is a concatenation of all sequences of output elements.

Example (From TA5): counting character frequency in strings

1) The input is a sequence of strings.
2) (Map phase) In each string we count how many times each character appears and then produce a sequence of the results.
3) (Sort/Shuffle phases) We sort the counts according to the character, creating new sequences in the process. Now for every character we have a sequence of all counts of this character from all strings.
4) (Reduce phase) For each character we sum over its respective sequence and produce the sum as a single output.

5) The output is a sequence of the sums.

Design

The implementation of this design can be split into two parts

1) Implementing the functions *map* and *reduce*. This will be different for every task. We call this part the client.
2) Implementing everything else – the partition into phases, distribution of work between threads, synchronisation etc. This will be identical for different tasks. We call this part the framework.

Using this split, we can code the framework once and then for every new task, we can just code the significantly smaller and simpler client.

Constructing the framework is the main goal of this exercise. In the next sections we will break this goal down into subgoals and provide a more detailed design for you to implement.

Client Overview

Since part of the task is sorting, every element must have a key that allows us to compare elements and sort them. For this reason each element is given as a pair (key,value).

We have three types of elements, each having its own key type and value type:

1) Input elements – we denote their key type *k1* and value type *v1*.
2) Intermediary elements – we denote their key type *k2* and value type *v2*.
3) Output elements – we denote their key type *k3* and value type *v3*.

The *map* function receives a key of type *k1* and a value of type *v1* as input and produces pairs of (*k2*,*v2*).

The *reduce* function receives a sequence of pairs (*k2*,*v2*) as input, where all keys are identical, and produces pairs of (*k3*,*v3*).

It is now obvious why the sort/shuffle phase is needed. We must sort the intermediary elements according to their keys and then create new sequences such that *reduce* will run exactly once for each *k2*.

A header *MapReduceClient.h* and a sample client are provided with this exercise. An implementation of a client contains the following:

1) Key/Value classes inheriting from *k1*,*k2*,*k3* and *v1*,*v2*,*v3* including a < operator for K2, to enable sorting.
2) Implementation of the MapReduceClient class:
    a. The *map* function with the signature:

    ```
    void map(const K1* key, const V1* value, void* context) const
    ```

    This function will produce intermediate pairs by calling the framework function *emit2*(*K2*,*V2*,context).

The context argument is provided to allow *emit2* to receive information from the function that called *map*.

b. The *reduce* function in the with the signature:

```
void reduce(const K2* key, const std::vector<V2*> &values, void* context) const
```

This function will produce output pairs by calling the framework function *emit3*(*K3*,*V3*,context).

The context argument is provided to allow *emit3* to receive information from the function that called *reduce*.

Framework Interface Overview

The framework will support running MapReduce operations as an asynchrony job, together with ability to query the current state of a job while it is running. A header *MapReduceFramework.h* is provided with the exercise.

Two types of variables are used in the header:

1. *JobState* - a struct which quantizes the state of a job, including:
   - stage_t stage – an enum (0-undefined, 1-Map, 2-Shuffle, 3-Reduce)
     - We run the shuffle thread in parallel to the map phase (more on that later), you should report "shuffle phase" starting after the last map thread finished and until the first reduce thread started.
   - float percentage – job progress of current stage (i.e., the percentage of keys that were processed out of all the keys that should be processed in the stage).
     - The shuffle reports amount of values already copied to the map out of all valued map produced (again more on that later)
2. JobHandle – void*, an identifier of a running job. Returned when starting a job and used by other framework functions (for example to get the state of a job).

The framework interface consists of six functions:

1) startMapReduceFramework – This function starts running the MapReduce algorithm (with several threads) and returns a JobHandle.

```
JobHandle startMapReduceJob(const MapReduceClient& client, const InputVec& inputVec, OutputVec& outputVec, int multiThreadLevel);
```

client – The implementation of *MapReduceClient* class, in other words the task that the framework should run.

inputVec – a vector of type std::vector<std::pair<K1*, V1*>>, the input elements

outputVec – a vector of type std::vector<std::pair<K3*, V3*>>, to which the output elements will be added before returning. You can assume that outputVec is empty.

multiThreadLevel – the number of worker threads to be used for running the algorithm. You can assume this argument is valid (greater or equal to 2).

2) waitForJob – a function gets the job handle returned by *startMapReduceFramework* and waits until it is finished.

   void waitForJob(JobHandle job)

3) getJobState – this function gets a job handle and updates the state of the job into the given JobState struct.

   void getJobState(JobHandle job, JobState state)*

4) closeJobHandle – Releasing all resources of a job. You should prevent releasing resources before the job is finished. After this function is called the job handle will be invalid.

   void closeJobHandle(JobHandle job)

5) emit2 – This function produces a (K2*,V2*) pair. It has the following signature:

   void emit2 (K2 key, V2* value, void* context)*

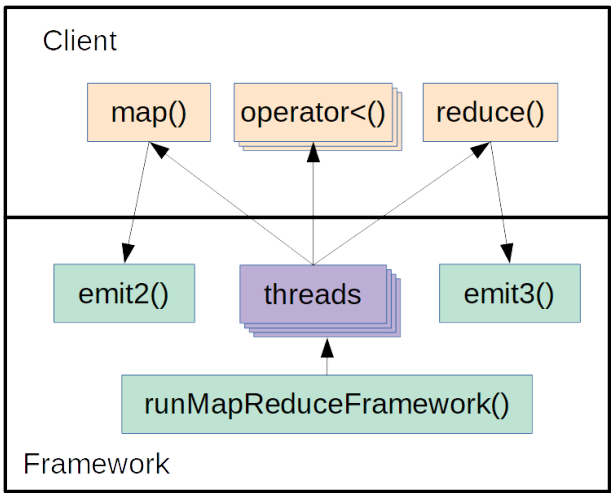
   The context can be used to get pointers into the framework's variables and data structures. Its exact type is implementation dependent.

6) emit3 – This function produces a (K3*,V3*) pair. It has the following signature:

   void emit3 (K3 key, V3* value, void* context)*

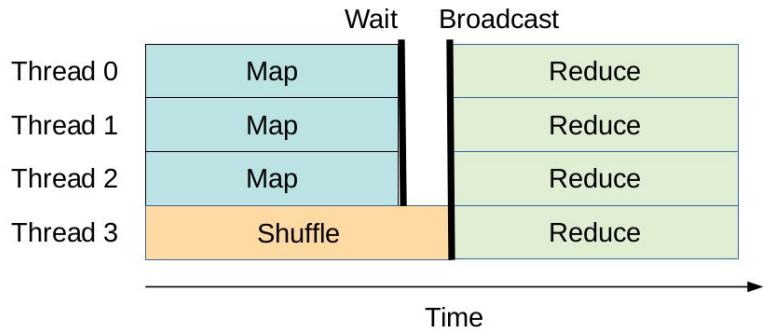
   The context can be used to get pointers into the framework's variables and data structures. Its exact type is implementation dependent.

The following diagram contains a summary of the functions in the client and the framework. An arrow from function a to function b means that a calls b.



Framework Implementation Details

We will implement a variant of the MapReduce model according to the following diagram[1]:



This diagram only shows *4* threads which is the case where the *multiThreadLevel* argument is *4*.

In the general case (where *multiThreadLevel=n*):

- Threads *0* through *(n-2)* runs Map and then Reduce
- Thread *(n-1)* runs Shuffle (simultaneously to the Map phase), and then Reduce

Map Phase

In this phase each thread reads pairs of (*k1*,*v1*) from the input vector and calls the *map* function on each of them. The *map* function in turn will call the *emit2* function to output (*k2*,*v2*) pairs. We have two synchronisation challenges here:

1) Splitting the input values between the threads – this will be done using an atomic variable shared between the threads, an example of using an atomic variable is provided together with the exercise. Read it and run it before continuing.
   The variable will be initialised to 0, then each thread will increment the variable and check its old value. The thread can safely call *map* on the pair in the index *old_value* knowing that no other thread will do so. This is repeated until *old_value* is after the end of the input vector, as that means that all pairs have been processed and the Map phase has ended.
2) Prevent output race conditions – This will be done by separating the outputs. To do that, create a std::vector<k2*,v2*> for each thread and in *emit2* function append the new *(k2,v2)* pair into the calling thread's vector. Accessing the calling thread's vector can be done by using the *context* argument.
   - This solution will solve race condition between the different Map threads (since every Map thread will work with a different vector)
   - The Shuffle thread will combine those vectors into a single map. Make sure there is no race condition between the Shuffle thread and the Map thread (more on that later).

[1] The diagram is schematic, and the time proportions are probably wrong

If you follow our suggestion, at the end of this phase you will have *n* vectors[2] of (*k2*,*v2*) pairs and all elements in the input vector are processed.

Shuffle Phase

This phase reads the Map phase output and combines them into a single IntermediateMap (of type std::map< k2*, std::vector<v2*> >).

Since this phase writes into a shared object, it is difficult to split it between multiple threads, so we use a single Shuffle thread. To increase parallelization in the framework, the Shuffle phase will be in parallel to the Map phase, of course that the shuffle should wait for more inputs while Map threads are still running.

The shuffle workflow is as follows: Iterate through the Map output (recall each map thread maintains a separate output sequence), for each out *(k2, v2)*, move the *v2* to the IntermediateMap according to *k2*. Since the Shuffle thread runs in parallel to the Map thread, and they both share an object. you need to protect the access to the object, use a mutex per map thread to do so (in total *multiThreadLevel - 1* mutexes for this synchronization). Only once the Shuffle thread finishes collecting all of the Map's outputs, can the Reduce phase start.

We have three synchronisation challenges here:

1. Synchronizing Map and Shuffle - Each "map" thread writes to its own object, and the Shuffle thread needs to read from there. Use a vector of mutexes, a mutex per Map thread and access this as a shared resource (this is similar to the Producer - Consumer problem).

2. Notify Map done - The Shuffle needs to work until all Map threads finished. Use a barrier and a shared variable, or an atomic variable (that counts how many Map threads had finished) to report the Map phase is done.

3. Notify Reduce Threads - Once the shuffle thread finished shuffling, the Reduce phase should start, using a conditional variable. Use *pthread_cond_wait* when the threads finish executing the map, and *pthread_cond_broadcast* once the shuffle thread is done.

Reduce Phase

In this phase each thread reads a key k2 and calls the *reduce* function using the key and it's value from the IntermediateMap. The *reduce* function in turn will call the *emit3* function to output (*k3*,*v3*) pairs which can be inserted directly to the output vector (*outputVec* argument of *startMapReduceJob*) under the protection of a mutex.

In this phase we only read from the IntermediateMap, but we still need to synchronize to ensure that every key will be processed once. Again there are multiple options and you can choose any implementation, one suggestion is to create a vector of (unique) *k2* (in the shuffle thread), and similar to the map phase, use an atomic variable to select a key for each thread, then safely read from IntermediateMap using that key (why is it safe?).

[2] where n = multiThreadLevel - 1

Once all the keys in the Reduce phase are processed, *waitForJob* may return. The task is done.

General Remarks

1. Inside *MapReduceFramework.cpp* you are encouraged to define *JobContext* – a struct which includes all the parameters which are relevant to the job (e.g., the threads, state, mutexes…). The pointer to this struct can be cast to JobHandle.
2. In order to check and update the job state, you **may** use atomic variables which are shared with running threads. Note that accessing multiple atomic variables is not atomic – take this under consideration or try implementing them using a single 64bit atomic variable[3]: There are 4 different stages (UNDEFINED, MAP, SHUFFLE, REDUCE), keep 2 bits to flag the stage, then you need to store the number of already processed keys and number of total keys to process (to calculate the task progress), use 31-bits for each number.
3. When a system call or standard library function fails (such as *pthread_create* or *std::atomic load*) you should print a single line in the following format: *"system error: text\n"*, text is a description of the error, and then exit(1).
4. We supply two Makefiles, in the main directory that creates *libMapReduceFramework.a* and in SampleClient that complies that client and links against the library (it looks for the library in current dir or parent). You don't have to use any of them, we expect that running ***make*** (with no arguments) will create *libMapReduceFramework.a* library

Part 1: Coding Assignment

Implementation (90 pts)

Implement the functions of the framework (those that appear in the MapReduceFramework.h) according to the details above and compile them into a static library libMapReduceFramework.a
Don't change the header files.

You must use the pthread library, for creating threads, mutexes etc. as was taught in class. It is forbidden to use c++'s threads, mutexes, etc. You are also not allowed to use pipes, user level threads or forks. The only exception to this rule is using std::atomic.

Your code must be Thread-safe, startMapReduceJob must work correctly when called from two different threads simultaneously. Think what implications this has on your design.

Pay attention to your runtime and complexity, this exercise is all about performance, you should avoid unnecessary copying of data or other preventable performance pains.
You must have no memory leaks.
The code you submit must not contain a main function, and should not print anything.

[3] std::atomic<uint64_t>

Evaluation - FileWordCounter (5 pts)

One of the challenges in parallel computing is to determine the parallelization level, in our case with how many threads to run. That depends between applications and hardware

We would like to test your MapReduce framework with the FileWordCounter client. As a benchmark you should use *test_word_count* directory (untar[4] the supplied *test_word_count.tar* file).
Test the framework with multiple different *multiThreadLevel* options, and plot the running time as a function of *multiThreadLevel*, submit it as *fileWordCounter_comparison.png* (or jpeg).
In your README file, describe the hardware you used (#cores, cpu model, see *lscpu* command), and explain your results.

To get accurate results connect to an "aquarium" computer (see message), and average the runtime of multiple iterations.

File Word Counter

1. The input is a directory name
2. (Map phase) for each file in the directory we count how many times each word appeared (a word is anything between two spaces)
3. (Sort/Shuffle phases) Combine all counts of each word, now for every word (that appears in any of the files) we have a sequence of counts, one per file
4. (Reduce phase) For each word we sum over its respective sequence and produce the sum as a single output.
5. The output is a sequence of the sums.

In FileWordCounter.cpp we supply a (not very efficient) implementation and a main function to run the FileWordCounter, after compiling with your static library you should run:

FileWordCounter <DIRECTORY_NAME> <MultiThreadLevel>

for example:

FileWordCounter test_word_count 10

will count the number of words in each file of test_client using 10 threads.

Part 2: Theoretical Questions (5 pts)

1. foo is a CPU bound application, What will be the optimal number of (kernel-level) threads for foo? You can assume there is no memory access. *(2 pt)*

2. Recall barrierdemo.cpp from TA4, and assume *MT_LEVEL=n*, and we switched the *foo* function with the following *foo* implementation
    a. How many lines will be printed (as a function of *n*) *(1 pt)*
    b. The order of the lines will not be deterministic, how many different options there are? *(2 pt)*

[4] run: *tar xf test_word_count.tar*

```
void* foo(void* arg)
{
    ThreadContext* tc = (ThreadContext*) arg;
    int pid = fork();
    if (pid != 0) {
        waitpid(pid, NULL, 0);
        tc->barrier->barrier();
    }
    printf("%d: after barrier pid: %d\n", tc->threadID, pid);

    return 0;
}
```

(you can assume fork command never fails)

Tips

- Since the keys only have the < operator and not ==, you can check if two keys ***a***,***b*** are identical by checking whether both ***a*** < ***b*** and ***b*** < ***a*** are false.
- Test early, test often: Using the example client make sure everything works when the Shuffle thread runs after the Map, only then solve the Map-Shuffle synchronization challenge. Think how you can test in a way that truly demonstrates all threads are being used properly.
- The sample client is not enough. Make more complicated clients and test with them.
- You can use *sleep* and *usleep* to "recommend " certain scheduling (for example *sleep* before starting the shuffle to simulate a situation where shuffle is getting CPU time first).

Submission

Submit a tar file containing the following:

- README file - The README should be structured according to the course guidelines. In order to be compliant with the guidelines, please use the README template that we provided. We expect to find in the file:
    - hardware description and explanation on the evaluation graph
    - Answers to the theoretical questions
- Makefile - Running *make* with no arguments should generate the *libMapReduceFramework.a* library.
- The source files for your implementation of the library
- *fileWordCounter_comparison.png* - the graph from section 1.2

Late submission policy						
Submission time	**1/6, 23:55**	**2/6, 23:55**	**3/6, 23:55**	**4/6, 23:55**	**5/6, 23:55**	**6/6, 23:55**
Penalty	**0**	-3	-10	-25	-40	-100