OS 2019 – Exercise 3
MapReduce - Multi-threaded Programming

Supervisor – Dr. Neta Rozen schiff

Due to: **12.5.2019**
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 phases are 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 the keys, to enable sorting.
2) The *map* function in the *MapReduceClient* class 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*.
3) The *reduce* function in the *MapReduceClient* class with the signature:

   void reduce(const IntermediateVec* value, void* context) const

IntermediateVec is of type std::vector<std::pair<K2*,V2*>>

All pairs in the vector are expected to have the same key (but not necessarily the same instances of K2).

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 a 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-undefine, 1-Map,2-Reeduce)
   - 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).
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) startMapReduceJob – 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* or 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 1).

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

   void waitForJob(JobHandle job)

3) getJobState – this function gets a job handle and check for his current state in a 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 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)*

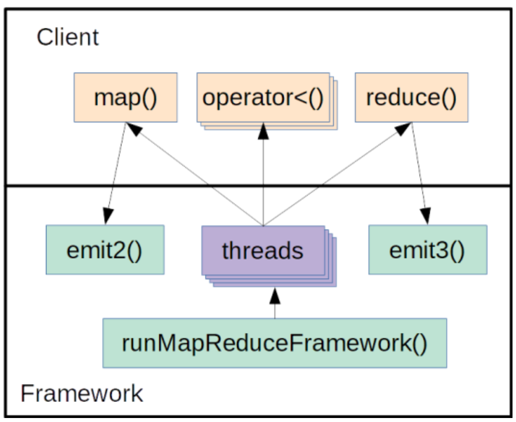
   The context can be used to get pointers into the framework's variables and data structures. Its exact type is implementation dependent.

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

   void emit2 (K3 key, V3* value, void* context)*

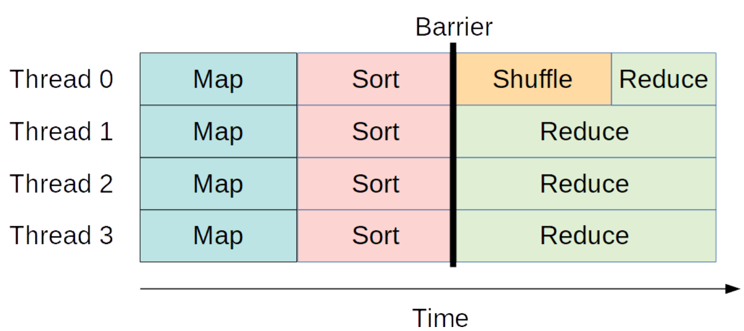
   The context can be used to get pointers into the framework's variables and data structures. Its exact type is implementation dependent.

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:



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

In this design all threads except thread 0 run three phases: Map, Sort and Reduce, while thread 0 also runs a special shuffle phase between its Sort and Reduce phases.

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 in this manner 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. We will create a vector for each thread and then *emit2* will just 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.

In the end of this phase we have *multiThreadLevel* vectors of (*k2*,*v2*) pairs and all elements in the input vector were processed.

Sort Phase

Immediately after the Map phase each thread will sort its intermediate vector according to the keys within. Since every thread has its own vector, this phase needs no special synchronisation. std::sort can be used to implement this phase with relatively little code.

The Shuffle phase must only start after all threads finished their sort phases.

In the end of this phase we must use a barrier – a synchronisation mechanism that makes sure no thread continues before all threads arrived at the barrier. Once all threads arrive, the waiting threads can continue. A sample C++ implementation of a barrier is provided together with the exercise, it is similar to the example in the presentation of Tirgul 4. You may use code from this example as is.

After the barrier, one of the threads will move on to the Shuffle phase while the rest will skip it and move directly to the Reduce phase.

Shuffle Phase

Recall that our goal in this phase is to create new sequences of (*k2*,*v2*) where in each sequence all keys are identical and all elements with a given key are in a single sequence.

Since our intermediary vectors are sorted, we know that all elements with the largest key must be at the back of each vector. Thus, creating the new sequence is simply a matter of popping these elements from the back of each vector and inserting them to a new vector. Now all elements with the second largest key are at the back of the vectors so we can repeat the process until the intermediary vectors are empty.

That is a task that is quite difficult to split efficiently into parallel threads so we run it in parallel with the Reduce phase instead. Whenever we finish creating a new vector, we put it in a queue for one of the reducing threads to pop and call *reduce* on.

Use a vector for the queue (note that it is a vector of vectors), with a semaphore for counting the number of vectors in it. Whenever a new vector is inserted to the queue we will call sem_post() on the semaphore to notify the reducing threads that they have pending work. Note that you will also need a mutex for protecting the access to this queue.

Once all intermediary vectors are empty, the shuffling thread will move on to the Reduce phase.

Reduce Phase

The reducing threads will wait for the shuffled vectors to be created by the shuffling thread. They can do so by calling sem_wait() on the aforementioned semaphore. Once they wake up, there must be at least one element in the queue, so now they can pop a vector from the back of the queue and run *reduce* on it (Remember to lock the mutex when necessary).

The *reduce* function in turn will call *emit3* to produce (*k3*,*v3*) pairs. These can be inserted directly to the output vector (*outputVec* argument of *startMapReduceJob*) under the protection of a mutex. The *emit3* function can access the output vector through its *context* argument.

Once both the Shuffle phase is finished and the queue is empty**,** waitForJob may return as the task is done.

General Remarks

1. Inside *MapReduceFramework.cpp* you are encouraged to define *JobContext* – a srtuct which includes all the parameters which are relevant to the job (e.g., the threads, state,…). The pointer to this struct can be casted 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 implement them using a single 64bit atomic variable (for example 31-bit for the processed key count, 31-bits for the total number stage keys, and 2 bits for the enum stage).

Your Assignment

- Implement the functions of the framework (those that appear in the MapReduceFramework.h header) 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. We will supply generous timeouts in the tests, but as this exercise is all about performance, you should still avoid unnecessary copying of data or other preventable performance sinks.
- You must have no memory leaks.

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 the Map and Sort phases work correctly before heading into the Shuffle & Reduce phases.
- The sample client is not enough. Make more complicated clients and test with them.

Submission

Submit a tar file containing the following:

- A 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.
- The source files for your implementation of the library.
- Your Makefile. Running *make* with no arguments should generate the *libMapReduceFramework.a* library. You can use this Makefile as an example. You may need to add the –pthread flag.

Late submission policy						
Submission time	**12.5, 23:55**	**13.5, 23:55**	**14.5, 23:55**	**15.5, 23:55**	**16.5, 23:55**	**19.5**
Penalty	0	-3	-10	-25	-40	-100