Report to the Final Project MapReduce of ECE 56300

Student: Shuzhan Sun

Team member: Chang Yang and Shuzhan Sun

GitHub: https://github.com/charlesyangc/MapReduce

1. Explanations to key functions and mechanisms

1.1. Hash map function: `int HashMap(std::string word, int num_reducer)`

The hash map function takes each word as a key and return a reducer id for the word. Based on the relative frequencies of the first letters in English language, each word is evenly distributed to the reducers based on its first letter. The relative frequencies of first letters are from the Wikipedia:

```
// Relative frequencies of the first letters of a word in the English language
// source: https://en.wikipedia.org/wiki/Letter_frequency
//std::map<char, float> FreqFirstLetter = {
//        { 'a', 0.11682 }, { 'b', 0.04434 }, { 'c', 0.05238 }, { 'd', 0.03174 }, { 'e', 0.02799 },
//        { 'f', 0.04027 }, { 'g', 0.01642 }, { 'h', 0.04200 }, { 'i', 0.07294 }, { 'j', 0.00511 },
//        { 'k', 0.00456 }, { 'l', 0.02415 }, { 'm', 0.03826 }, { 'n', 0.02284 }, { 'o', 0.07631 },
//        { 'p', 0.04319 }, { 'q', 0.00222 }, { 'r', 0.02826 }, { 's', 0.06686 }, { 't', 0.15978 },
//        { 'u', 0.01183 }, { 'v', 0.00824 }, { 'w', 0.05497 }, { 'x', 0.00045 }, { 'y', 0.00763 },
//        { 'z', 0.00045 } };
```

Evaluations: This hash map function can give a fairly good load balance according to our test results for the given files. Even though the mechanism can only assign up to 26 reducers limited by the number of English letters, it is enough for our problem here.

1.2. Generate work queue: `#include "concurrentqueue-master/concurrentqueue.h"`

We use online open source queue function (https://github.com/cameron314/concurrentqueue), which enables us to enqueue and dequeue certain type of data structures. Because of the built-in race protection mechanism, this library allows these queues being accessed by different threads without race in OpenMP. This also can improve the load balance when distributing works among threads, because whenever a thread is idle, that thread can grab a work from the queue.

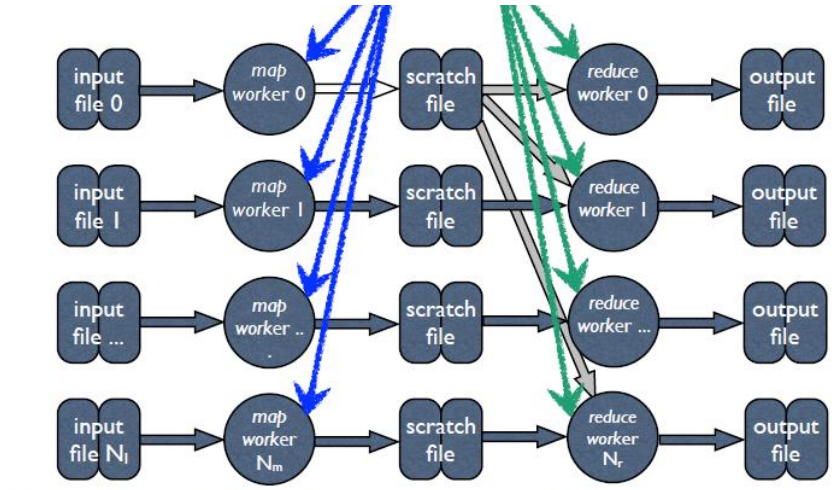
1.3. Count the word locally at each mapper thread: `#include <map>`

Instead of self-defining a tuple structure and adding new elements to it at each time, we adopt existing library `<map>` in C++, which simply does the work of counting. The count for each word can simply be done by one line of code: `++WordCount[word]`.

2. Different versions

2.1. Version 1:

In the first implementation, we follow the control flow diagram at the “ProjectOverview.pdf”:



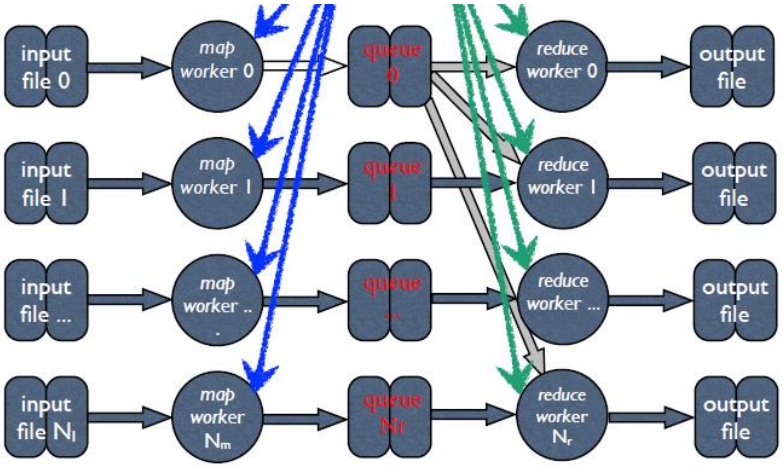
Even though this shows one input file for mappers and reducers, there should be more than one.

Using OpenMP, each mapper thread will count the word locally and create a temporary file for each word, as described by the above "scratch file". These files are enqueued and then read by each assigned reducer. We run with 7 threads for the given txt files. The running results are 265s, as given in the folder: "Version1_RunResult_265s".

Reasons for the slow speed of version 1: We test different thread numbers but find small running time difference. Then, the timing of each procedure shows that the major bottleneck is the I/O for the large number of temporary files from mappers to reducers.

2.2. Version 2: replace I/O of temporary files by Queues

Above bottleneck analysis to version 1 leads to our second version here, where we no longer generate a lot of temporary files. In this version, we directly enqueue the local word count from each mapper to the unique queue of each reducer. The control flow diagram is shown below. Getting rid of I/O, the same running results for the given txt files in 7 threads only cost 1.13s. The results are in the folder: "Version2_DeleteTempIO_1.13s".



Evaluations: Compare to version 1, the second version improves the running time from 265s to 1.13s, a big enhancement. However, we still some part might be further improved:

a) The mappers and reducers are not parallelized in current version. After all mappers finish the reading files and local count, the reducers then dequeue the word counts and combine the total counts of each word. Since reading from files and locally counting are much slower, and the generated queue may require a large cache memory, we could let the reducers simultaneously dequeue these queues and free the queue memory. This is done in version 3

2.3. Version 3: parallelize the mappers and reducers

With OpenMP, we can group the thread to execute different tasks (mapping and reducing) simultaneously, which leads to the version 3 here. (Reference: https://docs.oracle.com/cd/E19205-01/819-5270/aewbc/index.html)

Since the two mapper and reducer tasks are executed at the same time, chances are that the queue are empty when the reducers dequeue faster than the enqueue. So, we use a wait and notify mechanism to ensure that reducer threads wait until all mappers have finished before considering themselves complete. The key steps are:

1) After loading all the file names, we enqueue one more special file name at the end of the queue.
2) When any mapper thread reads this special file name, we change the flag indicating all mappers have finished their job.
3) The reducers are reducing the queue simultaneously with the mappers, but when the flag changes, the reducers will move forward to another reducing loop:

```
while (flag_reading_file_finished == 0) {
    if (inter_file_queue[reducer_id].try_dequeue(wordcount) == 1) {
        WordCount[wordcount.word] += wordcount.count;
    }
}
while (inter_file_queue[reducer_id].try_dequeue(wordcount) == 1 ) {
WordCount[wordcount.word] += wordcount.count;
}
```

We run the version 3 on 7 threads for the given files:

	Version 1: Generate temporary files	Version 2: 7 mappers then 7 reducers	Version 3: 4 mappers and 3 reducers	Version 3: 5 mappers and 2 reducers
Run time:	265s	1.13s	0.78s	0.70s

The results are in the folder: “Version3_ParallelMapperReducer”. Version 3 basically can achieve a 1.6X speedup compared to version 2.

3. Speedup tests

All these versions give the same counts. We tried some small examples and search the total counts using Microsoft Word, and we get the same number of word count. These results validate the accuracy of our codes.

For speedup analysis, we write a sequential code and the sequential running time is 1.9s. Then for the above version 3 with number of threads p = 7, parallel running time tp = 0.70s, some speedup metrices are:

1) Speedup $\Psi = t_{seq} / t_p = 1.9s / 0.7s = 2.7$
2) Efficiency $E = \Psi / p = 2.7 / 7 = 0.39$
3) Karp-Flatt metric $e = \frac{1/\Psi - 1/p}{1 - 1/p} = \frac{1/2.7 - 1/7}{1 - 1/7} = 0.265$

These metrices show our code has a fairly good performance on the given 17 loading files.

4. Possible improvements:

Limited by our time, we only solve a few bottlenecks as in above 3 versions. Facing large scale applications, we feel there are still some possible improvements:

1) In this version, we combine the reader and mapper at the same thread. We also make the reducer be the writer. These can all be adjusted for future large data application. Here, for our case, we feel the major time consuming is on IO, so, we mainly parallelize mappers and reducers.
2) The hash map function in this version limit us to have up to 26 reducers, we also need to use different hash map functions for larger number of threads.