Jose Rodriguez 9/29/2021

Map Reduce Report

CS 4175 - Parallel Computing

During this assignment I implemented a parallel version of the word count program in order to count the instances of certain words over works of Shakespeare. I made use of the map reduce design pattern, where I first split off the work through some mapping operation, and I later combined the results from each thread. The specific way in which I implemented this is by assigning each thread a filename to compute, and counting the set of desired words from that file which maps a filename to a local dictionary of word counts. Afterwards, once each thread is done, I combine the counts from each different thread by adding them together.

The way in which I split off the work in a parallel way is through Python pymp's thread iterator while iterating on a list of filenames, something like this:

```
def compute_with_map_reduce(num_threads):
    shared_dict = pymp.shared.dict()

    with pymp.Parallel(num_threads) as p:
        sum_lock = p.lock
        for filename in p.iterate(files_to_search):
            # Mapping the filename from a string into a dictionary count
            current_counts = count_words_from_file(filename)

            # Reducing the multiple dictionaries into a single aggregate
            # dictionary
            sum_lock.acquire()
            add_dictionaries(current_counts, shared_dict)
            sum_lock.release()

    return dict(shared_dict)
```

Problems I encountered

During this assignment I didn't encounter many issues. I initially had some issues on counting the total number of words that exist on the word since I was having issues getting the number of words that was mentioned. Upon further review, I realized that I was splitting off the words which resulted in some of the words not being properly counted. For instance, "heart" may appear in multiple different ways as subparts of other words like "heartbreak". I think that

the goal counts also counts these words that contain other words like heartbreak. I was able to fix this by using str.count.

I also encountered issues attempting to add up the dictionaries together during the reduction stage, since in some cases I was struggling with a race condition. I was able to correct this issue by making use of a mutex lock so that only one thread was able to modify the dictionary at a given time.

Problems I was not able to overcome

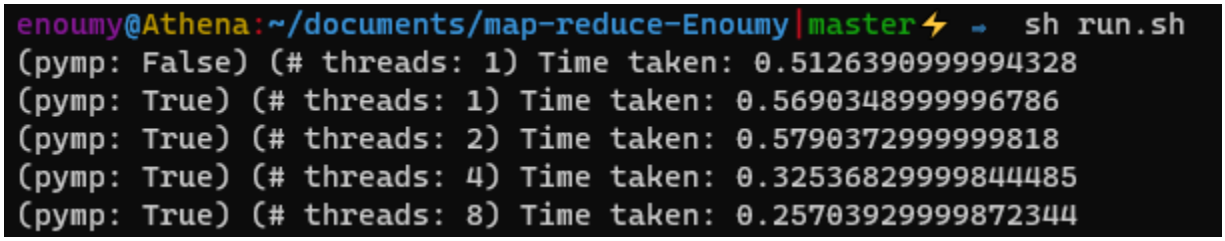
I was able to implement all of the functionality required by the assignment. I was able to overcome my problems.

How long the assignment took me

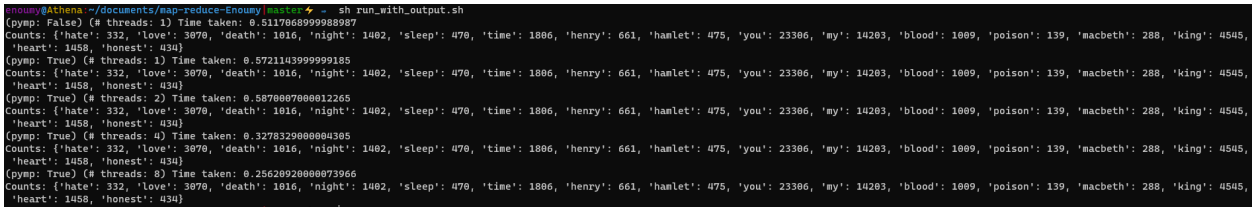
It took me around 4 total hours to complete this assignment. It took me like an hour to read the assignment + think about my implementation and what it would look like. Then I spent like 2 hours implementing the project itself. Furthermore, I spent around 1 hour writing the report of the assignment.

Performance measurements

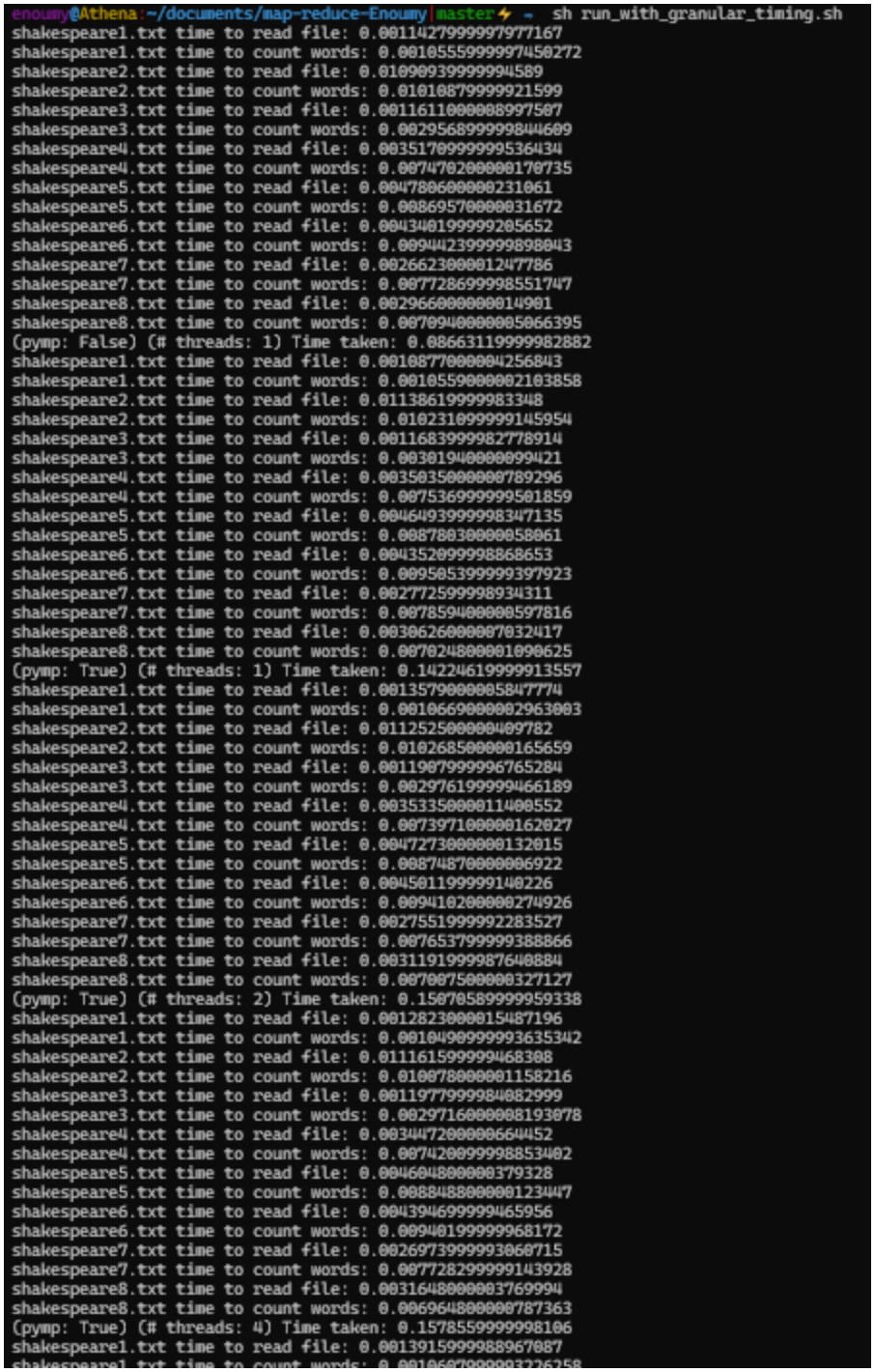
Performance measurements with only timing:



Performance measurements with output:



Performance measurements on the reading time taken and the counting time taken for each file:



Furthermore, the results of the map reduce operation running on different threads is tabulated in the following manner:

Test	**Time taken**	**Improvement**
No parallelism	0.5126 seconds	

Single thread	0.5690 seconds	0.9008
Two threads	0.5790 seconds	0.9827
Four threads	0.3254 seconds	1.7793
Eight threads	0.2570 seconds	1.2661

Analysis

Something worth noting is that when the program goes from not running with any parallelism to parallelism but with a single thread there is a performance hit of around 10%. This can probably be explained due to the extra overhead necessary to get pymp working even though it only uses one thread.

When the program goes from using a single thread to two threads the performance does not improve either, in fact, it goes down by around 2%. This can be explained due to a variety of factors that cancel out each other. In addition to the larger overhead necessary to run multiple threads in parallel, there is now the overhead of having a lock, meaning that threads would have to wait on each other. This is not as big as a performance hit since there are also performance gains for performing the counting of the words in parallel.

When the program goes from 2 threads to 4 threads, the program finally speeds up by running around 1.77 faster. This is still not as ideal as a doubling in speed from a doubling in threads still. This can be explained by the extra overhead in the reduction step and from the extra overhead from running more threads/improper balance of workload in the threads.

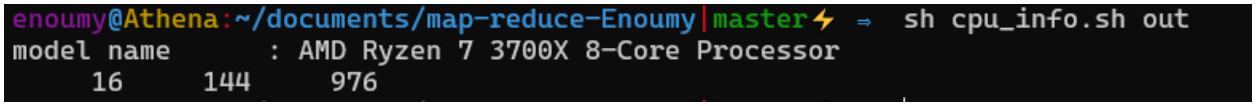
When the program moves from four threads to 8 there is now a speed increase of 1.2661 which is less than before. This can be explained since there are still some sequential steps that need to occur. The more threads we add, the less of a speed up we get which leads to diminishing returns.

Observations and comments

I enjoyed working on this assignment. I had used map reduce libraries that abstracted away a lot of the underlying work in the past. I really like learning about how map reduce works at a lower level.

Output of CPU info

This section contains the output of when I run CPU info.


enoumy@Athena:~/documents/map-reduce-Enoumy master ⇒ sh cpu_info.sh out
model name : AMD Ryzen 7 3700X 8-Core Processor
16 144 976