

# 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 pypm's thread iterator while iterating on a list of filenames, something like this:

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

    with pypm.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:

```
enoumy@Athena:~/documents/map-reduce-Enoumy|master ⚡ - sh run.sh
(pymp: False) (# threads: 1) Time taken: 0.51263909999994328
(pymp: True) (# threads: 1) Time taken: 0.56903489999996786
(pymp: True) (# threads: 2) Time taken: 0.5790372999999818
(pymp: True) (# threads: 4) Time taken: 0.325368299999844485
(pymp: True) (# threads: 8) Time taken: 0.257039299999872344
```

Performance measurements with output:

```
enoumy@Athena:~/documents/map-reduce-Enoumy|master ⚡ - sh run_with_output.sh
(pymp: False) (# threads: 1) Time taken: 0.51263909999994328
Counts: {'hate': 332, 'love': 3070, 'death': 1016, 'night': 1402, 'sleep': 470, 'time': 1886, 'henry': 661, 'hamlet': 475, 'you': 23366, 'my': 14283, 'blood': 1009, 'poison': 139, 'macbeth': 288, 'king': 4545, 'heart': 1458, 'honest': 434}
(pymp: True) (# threads: 1) Time taken: 0.57211439999999185
Counts: {'hate': 332, 'love': 3070, 'death': 1016, 'night': 1402, 'sleep': 470, 'time': 1886, 'henry': 661, 'hamlet': 475, 'you': 23366, 'my': 14283, 'blood': 1009, 'poison': 139, 'macbeth': 288, 'king': 4545, 'heart': 1458, 'honest': 434}
(pymp: True) (# threads: 2) Time taken: 0.58700070000012265
Counts: {'hate': 332, 'love': 3070, 'death': 1016, 'night': 1402, 'sleep': 470, 'time': 1886, 'henry': 661, 'hamlet': 475, 'you': 23366, 'my': 14283, 'blood': 1009, 'poison': 139, 'macbeth': 288, 'king': 4545, 'heart': 1458, 'honest': 434}
(pymp: True) (# threads: 4) Time taken: 0.32783290000004385
Counts: {'hate': 332, 'love': 3070, 'death': 1016, 'night': 1402, 'sleep': 470, 'time': 1886, 'henry': 661, 'hamlet': 475, 'you': 23366, 'my': 14283, 'blood': 1009, 'poison': 139, 'macbeth': 288, 'king': 4545, 'heart': 1458, 'honest': 434}
(pymp: True) (# threads: 8) Time taken: 0.256209200000073966
Counts: {'hate': 332, 'love': 3070, 'death': 1016, 'night': 1402, 'sleep': 470, 'time': 1886, 'henry': 661, 'hamlet': 475, 'you': 23366, 'my': 14283, 'blood': 1009, 'poison': 139, 'macbeth': 288, 'king': 4545, 'heart': 1458, 'honest': 434}
```

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

```

enoumy@Athena: ~/documents/map-reduce-Enoumy master ✚ - sh run_with_granular_timing.sh
shakespeare1.txt time to read file: 0.0011427999997977167
shakespeare1.txt time to count words: 0.0010555999997450272
shakespeare2.txt time to read file: 0.01090939999994589
shakespeare2.txt time to count words: 0.010108799999921599
shakespeare3.txt time to read file: 0.0011611000008997507
shakespeare3.txt time to count words: 0.002956899999844609
shakespeare4.txt time to read file: 0.00351709999999536434
shakespeare4.txt time to count words: 0.007470200000170735
shakespeare5.txt time to read file: 0.004780600000231061
shakespeare5.txt time to count words: 0.00869570000031672
shakespeare6.txt time to read file: 0.004340199999205652
shakespeare6.txt time to count words: 0.009442399999898043
shakespeare7.txt time to read file: 0.002662300001247786
shakespeare7.txt time to count words: 0.0077286999998551747
shakespeare8.txt time to read file: 0.002966000000014901
shakespeare8.txt time to count words: 0.0070940000005066395
(pymp: False) (# threads: 1) Time taken: 0.08663119999982882
shakespeare1.txt time to read file: 0.0010877000004256843
shakespeare1.txt time to count words: 0.0010559000002103858
shakespeare2.txt time to read file: 0.01138619999983348
shakespeare2.txt time to count words: 0.010231099999145954
shakespeare3.txt time to read file: 0.00116839999982778914
shakespeare3.txt time to count words: 0.003019400000099421
shakespeare4.txt time to read file: 0.0035035000000789296
shakespeare4.txt time to count words: 0.007536999999501859
shakespeare5.txt time to read file: 0.0046493999998347135
shakespeare5.txt time to count words: 0.00878030000058061
shakespeare6.txt time to read file: 0.0043520999998868653
shakespeare6.txt time to count words: 0.009505399999397923
shakespeare7.txt time to read file: 0.0027725999998934311
shakespeare7.txt time to count words: 0.007859400000597816
shakespeare8.txt time to read file: 0.0030626000007032417
shakespeare8.txt time to count words: 0.007024800001090625
(pymp: True) (# threads: 1) Time taken: 0.14224619999913557
shakespeare1.txt time to read file: 0.0013579000005847774
shakespeare1.txt time to count words: 0.0010669000002963003
shakespeare2.txt time to read file: 0.011252500000409782
shakespeare2.txt time to count words: 0.010268500000165659
shakespeare3.txt time to read file: 0.0011907999996765284
shakespeare3.txt time to count words: 0.002976199999466189
shakespeare4.txt time to read file: 0.0035335000011400552
shakespeare4.txt time to count words: 0.007397100000162027
shakespeare5.txt time to read file: 0.0047273000000132015
shakespeare5.txt time to count words: 0.00874870000006922
shakespeare6.txt time to read file: 0.004501199999140226
shakespeare6.txt time to count words: 0.009410200000274926
shakespeare7.txt time to read file: 0.0027551999992283527
shakespeare7.txt time to count words: 0.007653799999388866
shakespeare8.txt time to read file: 0.0031191999987640884
shakespeare8.txt time to count words: 0.007007500000327127
(pymp: True) (# threads: 2) Time taken: 0.15070589999959338
shakespeare1.txt time to read file: 0.0012823000015487196
shakespeare1.txt time to count words: 0.0010490999993635342
shakespeare2.txt time to read file: 0.011161599999468308
shakespeare2.txt time to count words: 0.010078000001158216
shakespeare3.txt time to read file: 0.0011977999984882999
shakespeare3.txt time to count words: 0.0029716000008193078
shakespeare4.txt time to read file: 0.003447200000664452
shakespeare4.txt time to count words: 0.007420099998853402
shakespeare5.txt time to read file: 0.004604800000379328
shakespeare5.txt time to count words: 0.008848800000123447
shakespeare6.txt time to read file: 0.004394699999465956
shakespeare6.txt time to count words: 0.00940199999968172
shakespeare7.txt time to read file: 0.0026973999993060715
shakespeare7.txt time to count words: 0.007728299999143928
shakespeare8.txt time to read file: 0.0031648000003769994
shakespeare8.txt time to count words: 0.006964800000787363
(pymp: True) (# threads: 4) Time taken: 0.1578559999998106
shakespeare1.txt time to read file: 0.0013915999988967087
shakespeare1.txt time to count words: 0.0010667000003276258

```

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	

<b>Single thread</b>	0.5690 seconds	0.9008
<b>Two threads</b>	0.5790 seconds	0.9827
<b>Four threads</b>	0.3254 seconds	1.7793
<b>Eight threads</b>	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 pypm 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
```