# MEMO

## Introduction:

The goal of this report is to access the performance increase of running multiple threads over single thread when parsing huge data file. To do this, we implemented pmap as multiple thread version and map as single thread version and compared their results by executing two self-made Rower subclasses. We decided that the performance of these two functions is the measured output of time to execute, which suggests that faster running time will have better performance. To reduce errors and bias, we decided to run tests on two different laptops but only compared the results which come from the same machine to uphold impartiality. The first machine we used is a Microsoft Surface pro which has 2.6GHz Intel Core i7 CPU with 8GB RAM, and it runs code with a Ubuntu VMware workstation; the second machine is a MacBook Pro of 2.6GHz Intel Core i5 with 16GB RAM, which runs code using PyCharm. We kept both machines' chargers plugged during testing to make sure the performance is optimal.

## Implementation:

Our basic idea of pmap is to initialize multiple threads and execute part of map function in each thread, then combine these results. Since our map has a for loop which goes through every row and accepts it, it will be relatively slow for parsing huge number of rows (e.g. more than 1million). So, we considered to split these rows equally into each thread and run these threads concurrently to reduce overall time. Our machines have 4 CPU cores which can support a maximum of 4 threads at the same time, but we chose to use only 2 threads as it could eliminate the cost of managing multiple threads as much as possible, and should be enough to show the increase in performance. We implemented pmap by initializing an array of threads which has a size of 2 and each thread which takes in its index (which represent the part of rows it deals with) to call map_helper. We made map_helper which maps the corresponding half of rows by the given index so that these 2 threads can map through all the rows. After that, we call thread.join on each thread to combine the result and end pmap. We did all implementations inside DataFrame class, including initializing threads and creating the function they take in. So the main challenge was to figure out the correct representation of parameters for new std::thread, as it takes in an extra object class "this" and requires to make rower as a standard reference.

## Description of the analysis:

The first step is experimental design. As the assignment specification required, we made two rower subclasses which were different in level of expensive operations. The first rower subclass is Fibonacci which calculates the Fibonacci numbers from the third

element to the end in a row, which was provided with the first two integers. This rower class only deals with simple sum operations, so we made it as the less expensive one.

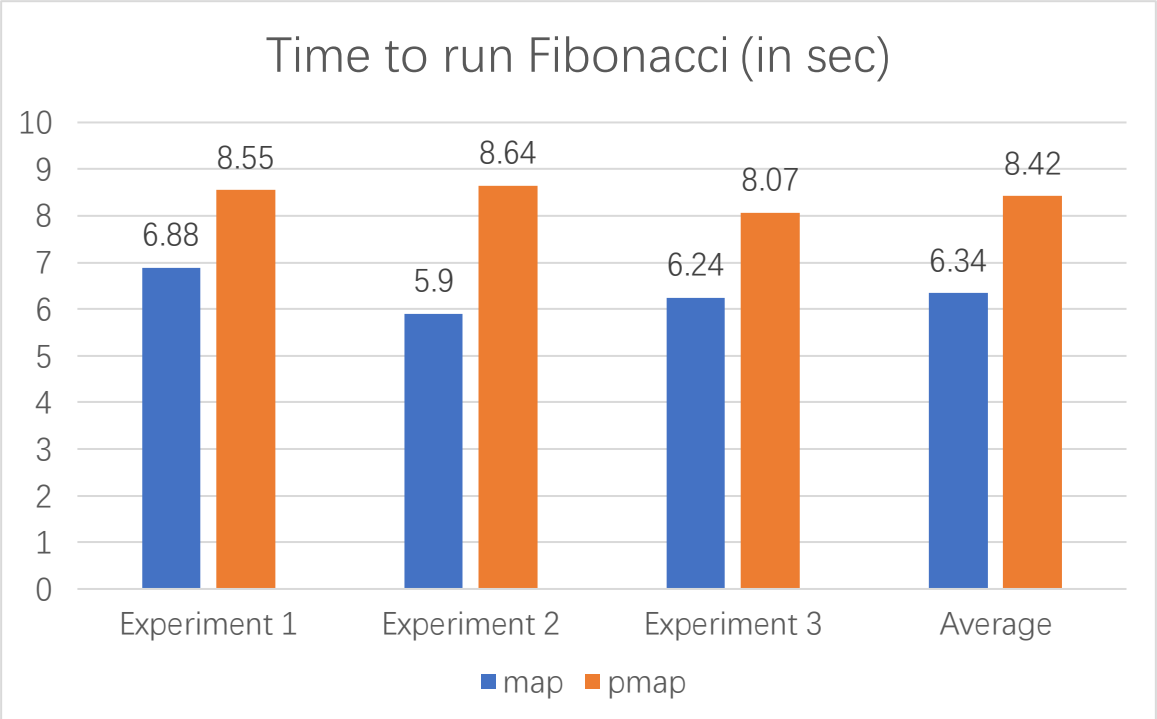
The second rower subclass is RCG which randomly generates a character for each element and allocate 2 bytes of memory for it. We used malloc for memory allocation and limit the total allocation size around 100MB, which indicates a dataframe with 10 columns and 5,000,000 rows in maximum. As this rower class includes large memory allocation which is far more expensive than sum operations in C++, we made it as the more expensive one which could hopefully evaluate the performance better than the first one.

To test with impartiality and accuracy, we performed a single-build and measured multiple runs which test the time of map and pmap on each machine. The data we used for test was generated in our test function inside bench.cpp by a for loop which adds approximately 1 million of rows to an empty dataframe. Due to different schemas of each rower subclass, the total number of rows also varied. But in general, every dataframe we used for test has 10 columns and size of approximately 100MB.

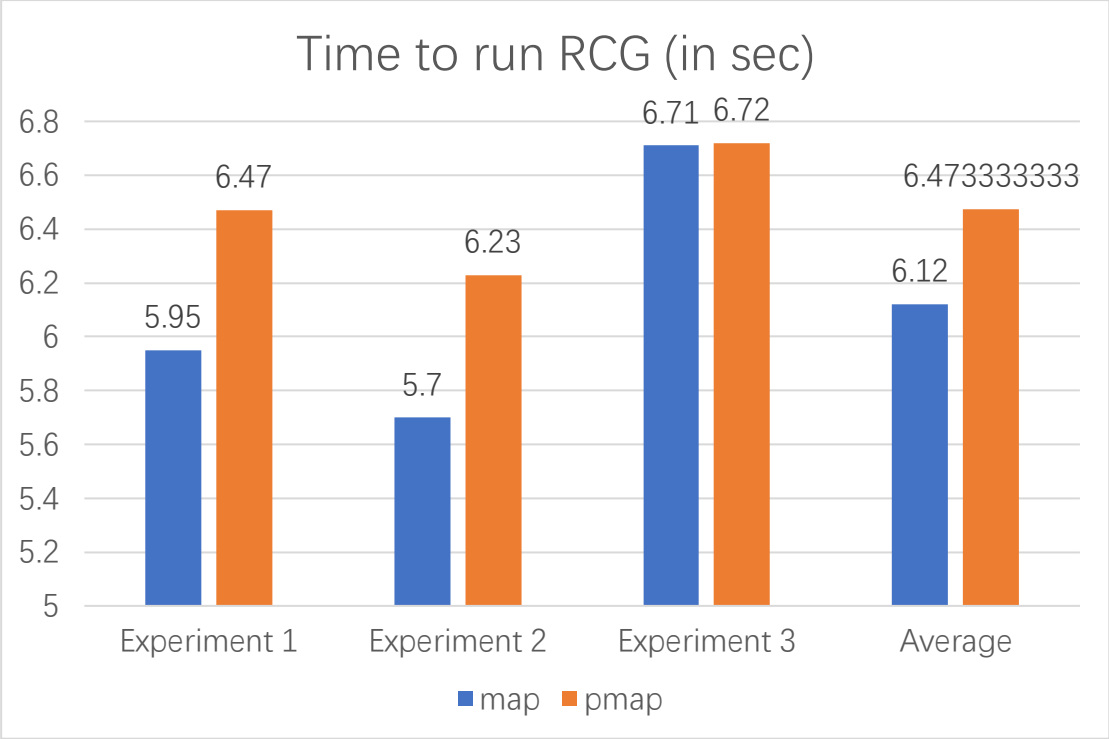
To test the runtime of map and pmap, we used standard chrono clock in <chrono> to record the start and end time of execution in microseconds and calculated their difference to get the result. Due to huge memory usage when parsing large data, the program could be killed if execute map or pmap multiple times in a single run. To avoid this from happening, we made everything the same except the map function we called inside test functions for the same rower and called one test function for each run. We repeated the experiment three times for large data input.

## Comparison of experimental results:

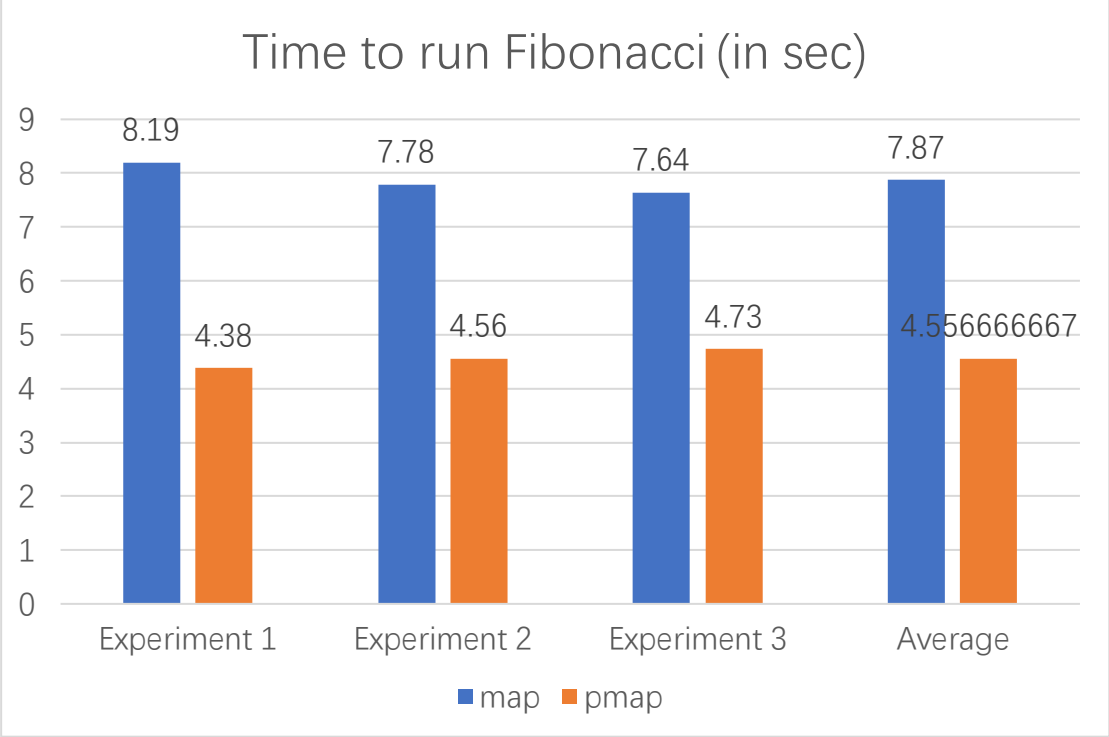
On Surface Pro, Testing Dataframe with 10 columns, and 1200*1000 rows.



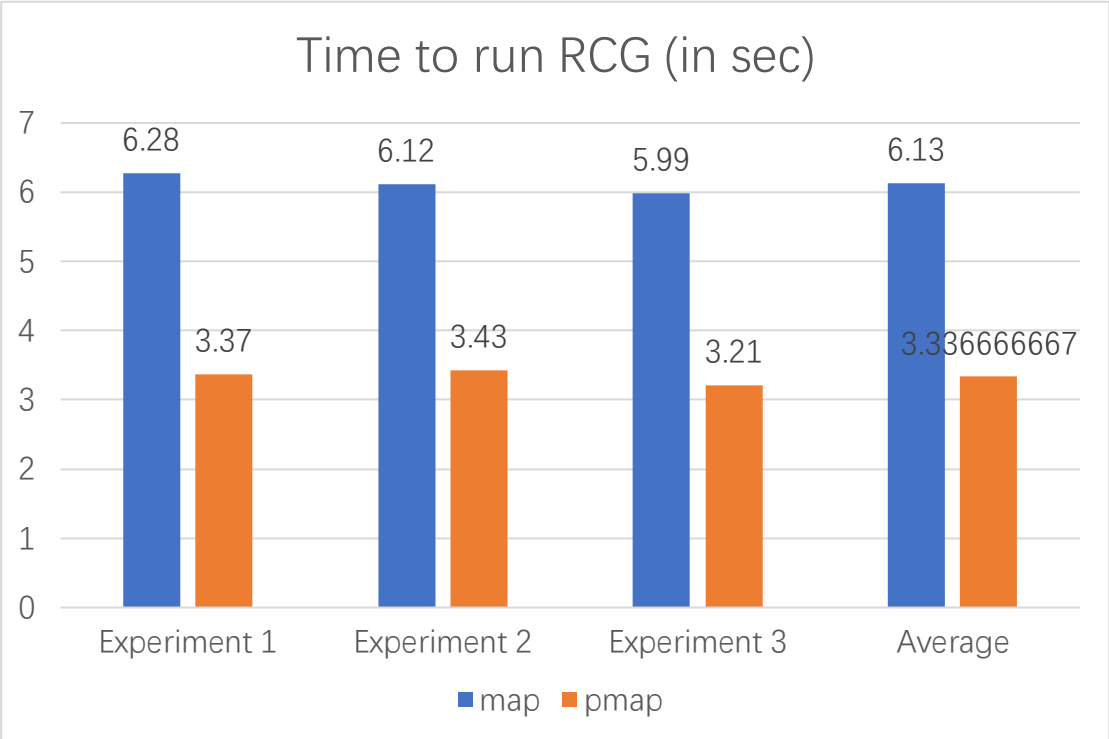
On Surface Pro, Testing Dataframe with 10 columns, and 600*1000 rows.




On MacBook Pro, Testing Dataframe with 10 columns, and 1200*1000 rows.



On MacBook Pro, Testing Dataframe with 10 columns, and 600*1000 rows.




For tests on surface, the results were quite unexpected, as pmap runs slower than map in both cases, which is opposite to our assumption. For comparison in detail, pmap also had worse performance in running Fibonacci than RCG regarding to map.

Despite the unexpected result, we could still find that multithreading performs better in operations which are expensive and costly, as pmap performs better relatively to map in RCG than Fibonacci. If we could design our rower subclasses to be more expensive, hopefully pmap would have less runtime than map.

As for tests on MacBook, pmap runs significantly faster than map in both cases as expected, especially for RCG. The runtime of pmap for RCG is approximately half of the corresponding runtime of map, which is very close to our optimal predicated result of 1/n (n equals number of threads we used) of time for map.

Although we got totally different results in these two machines, we didn't think any of them should be discarded as testing errors. On the other hand, both were useful for analyzing the performance of pmap under different operations. The slow runtime of pmap in surface pro could probably be affected by the age of the system (about 7 years) and excessive usage of CPU which increased the time of managing multiple threads. However, there are still potential reasons from our designs:

1. Our rower subclasses don't have operations which are expensive enough to require multithreading. Since Fibonacci is only about simple sum operation, the runtime processed by a single thread could be very fast so there is little difference between using different number of threads to execute. Although pmap has better performance in RCG, this increase of performance still can't cover its side-effects, such as extra time consumed by initializing multiple threads and joining them.

2. The joining part in our pmap consumes most of the time in our test. It has shown that more threads we used, more time we took for running pmap. However, the time for calling parallel mapping (accepting all rows) is much faster than map, as we tested it by commenting out the joining part. This is related to reason one, since we don't have any expensive operation which can cover the time of joining. But on the other hand, our joining method is also costly which makes multithreading inefficient.

## Threats to validity

There are several threats to validity:

1. We didn't test map and pmap in the same test function. This may cause errors in measurements as the usage and performance of CPU could vary when initializing and parsing dataframe before calling map and pmap. Thus, there would be more factors affecting the time performance out of control.
2. Our map function only visits rows with rower but doesn't make change to the dataframe. This means we are not calling filter in map, so we don't need to consider rows to be dropped if there is any. Since we don't deal with these incorrect rows, the runtime could be slightly faster.
3. The dataframes we used for testing are very closed to the limit which could kill the program. This is likely to hurt the performance when running on other machines or languages with more heavy runtime systems.
4. We used thread.join instead of self-defined joining method, and we didn't do any cleaning stuff after the joining. This could possibly result in larger usage of memory which took more time to run pmap.

## Conclusions

Based on the above analysis, we found that pmap has better performance than map, especially in mapping expensive and costly operation. Besides, the parallel mapping, which refers to splitting rows and equally assigning each thread to accept them, is much more efficient than mapping with a single thread. However, it is possible for the total runtime of pmap to be slower than map for some reasons, such as old system life, high CPU usage and easy operations. This suggests that multithreading could be inefficient than single threading when some special conditions are meet. As a result, multithreading can improve performance when dealing with huge data input than single thread in general, and it is more efficient in running expensive and costly operations in C++.