Assignment 5 – Part 1. Concurrency

By Divit Koradia, Ethan Griffee

Introduction

In the context of computer science, concurrency is the ability for a program to be decomposed into parts such that they can run independently of each other. This means that tasks can be executed out of order and the results would still be the same if they are executed in order. Thus, one of the key benefits of implementing concurrency is to reduce the time it takes for running and executing a program.

To test this claim, we designed an experiment to measure the performance increase of **pmap** over **map**– using the performance on map function as a baseline. Specifically, we are trying to test if the **pmap** function, in general, is faster at executing calculations on the rows of a data frame in comparison to the **map** function.

Implementation

To implement pmap, two separate classes were created which extended the thread.h file that was given. These classes, appropriately named DataFrameThreadOne and DataFrameThreadTwo both take in a reference to a data frame and a rower. In the run function of the first class, accept is called on the first nrows/2 rows, and the second class handles the next half. Pmap assumes the rower input has appropriately overridden the clone function of Object, and thus clones the rower input. It then creates one object of each of these classes using a different rower each. The start function is then called on each of the two classes. When both are finished the join function is called on the original rower with the copy as an argument.

Analysis performed

To test performance differences between **map** and **pmap,** we did the following:

1. Used a for loop to create a synthetic datafile.txt with approximately 3.5 million rows and 11 columns.
2. Implemented 4500NE teams' SoReR by replacing their columns with the columns in our data frame so that we could parse the datafile.txt and populate our data frame.
3. Once we had populated the data frame with information, we created two different Rower subclasses to test the behavior and implementation of our **map** and **pmap** function.
    a. Rower 1 – is a rower which counts up to 10,000 for every row it parses and then returns the sum of counts for every row in the data frame.
    b. Rower 2 – is a rower which reads in a row and adds each field of that row to an array of its datatype. For instance, if it encounters a string, it will add it to a StringArray or if it is an int it will add it to an IntArray and so forth.

4. For testing purposes, we choose to run each of the rower subclasses with our **map** and **pmap** function for three different values of length (number of rows we read from the datafile.txt) to determine how long a given code takes to run.
5. The different values of length we used were:
    a. max_len: the data frame reads in all the 3.5 million rows
    b. half_len: the data frame reads in half of the max_len i.e. 1.75 million rows
    c. one_tenth_len: the data frame reads around $1/10^{th}$ the max_len i.e. 350,000 rows
6. We use the Linux "*time*" command which prints out three different values of time:
    a. real or total – elapsed real time between invocation and termination
    b. user – the user CPU time
    c. system (sys) – the system CPU time

    For this assignment, we concentrate on the real time because it the best representation of how fast the functions will run with respect to a wall clock.
7. We repeat step 4 three times for accuracy so that we can plot the averages of the 3 different trials in our graphs.
8. Finally, we run the same test on different computers for a holistic viewpoint of how our functions perform across different software and hardware specifications.
9. We used our laptops and a Windows PC as test benches. Both of us had the latest version of macOS Catalina (v 10.15.3) while the PC was running on the latest version Windows 10. Our hardware configurations were different.
    a. Ethan's laptop is running a 2015 2.2 GHz Quad-Core Intel Core i7 with 16GB of ram.
    b. Divit's laptop is running a 2019 2.4 GHz Quad-Core Intel Core i5 with 8GB of ram.
    c. Divit's PC is running a 2019 3.6 GHz 8-Core Intel Core i9 with 16GB of ram.

Findings and Results

After collecting the data and reviewing it, we can confidently say that on average, our **pmap** function executes faster than our **map** function. If you look at figure 1, 2 and 3 you will notice that across all the three machine we used to test Rower 1 on, our **pmap** function consistently outperformed the **map** function for different input lengths by an average of 40%.

Data recorded for Rower 2 (figure 4, 5 and 6) seems to follow a similar trend, however, the percentage change in time between **map** and **pmap** function is considerably lower at around 8-10%. Furthermore, the margin of returns diminish as the input length increases in size. For instance, on Ethan's laptop the percentage change in time between **pmap** and **map** executing Rower 2 decreases as the length is increased from one_tenth to max_len. We believe this behavior is caused by our join_delete function for Rower 2 as it uses an add_all() function from the array class to join the arrays stored in the two different threads. Once they are done executing the add all method basically redoes the work done by thread 2 initially. Hence, we believe the cost of joining is a source of nondeterministic behavior as it seems to cost more in Rower2 with longer inputs.

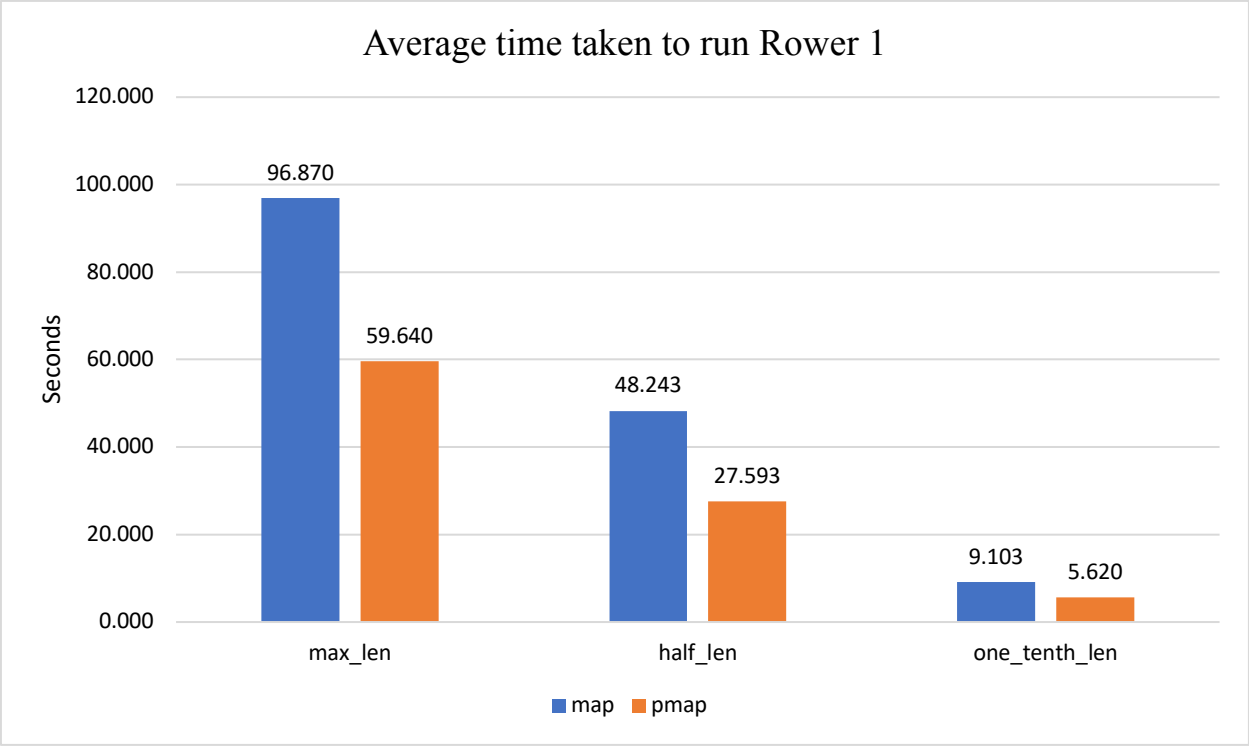

Figure 1. Average time taken to run Rower 1 on Ethan's Laptop.

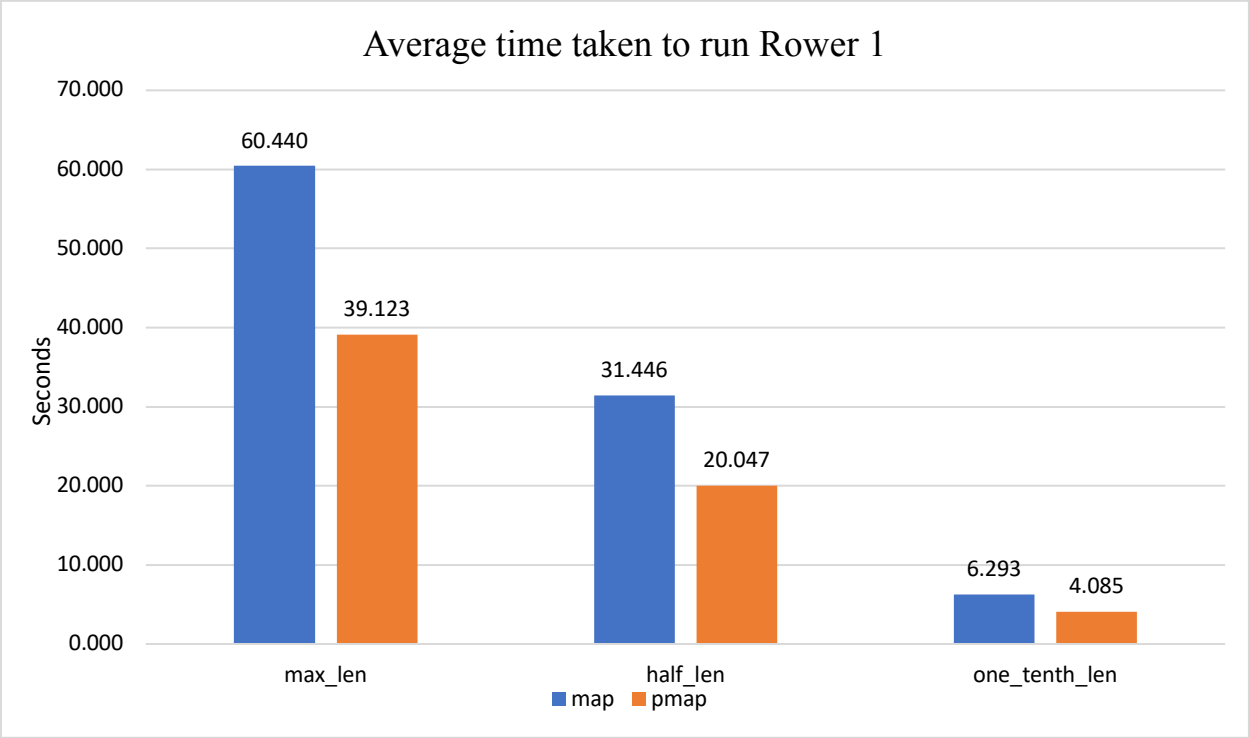

Figure 2. Average time taken to run Rower 1 on Divit's Laptop.

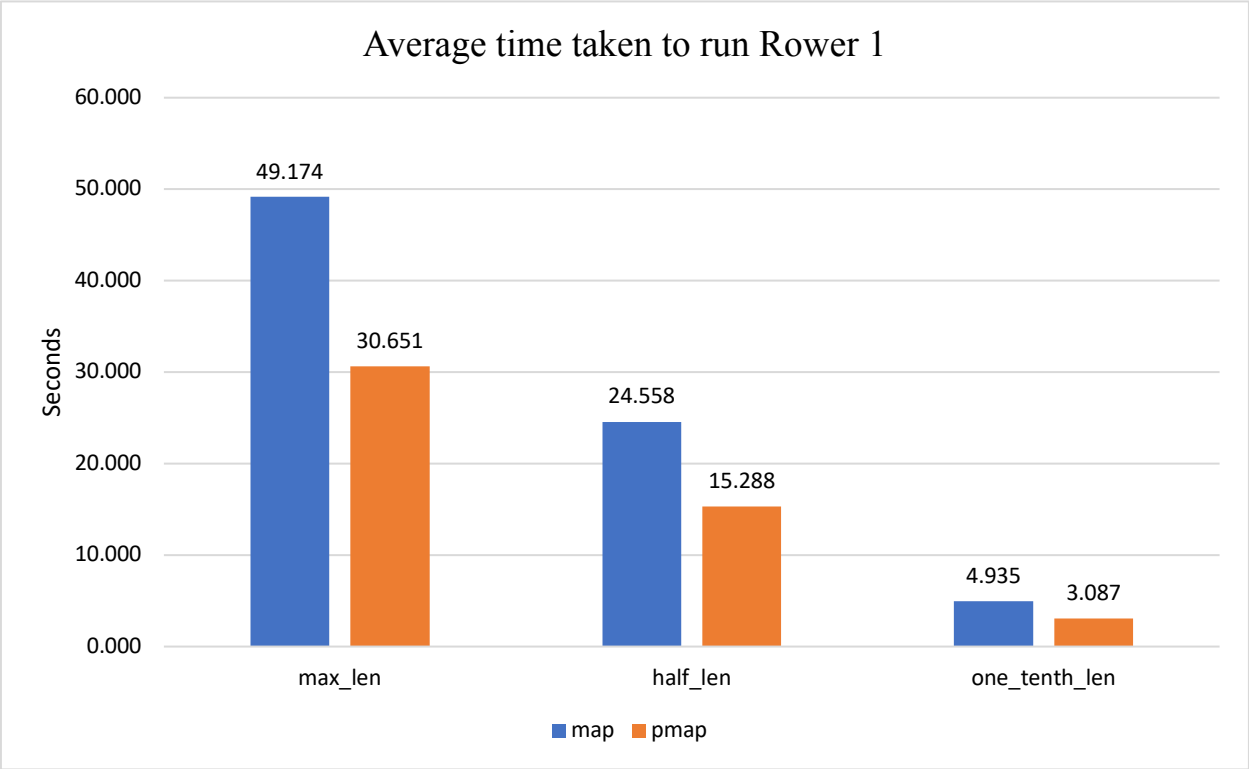

Figure 3. Average time taken to run Rower 1 on Divit's PC.

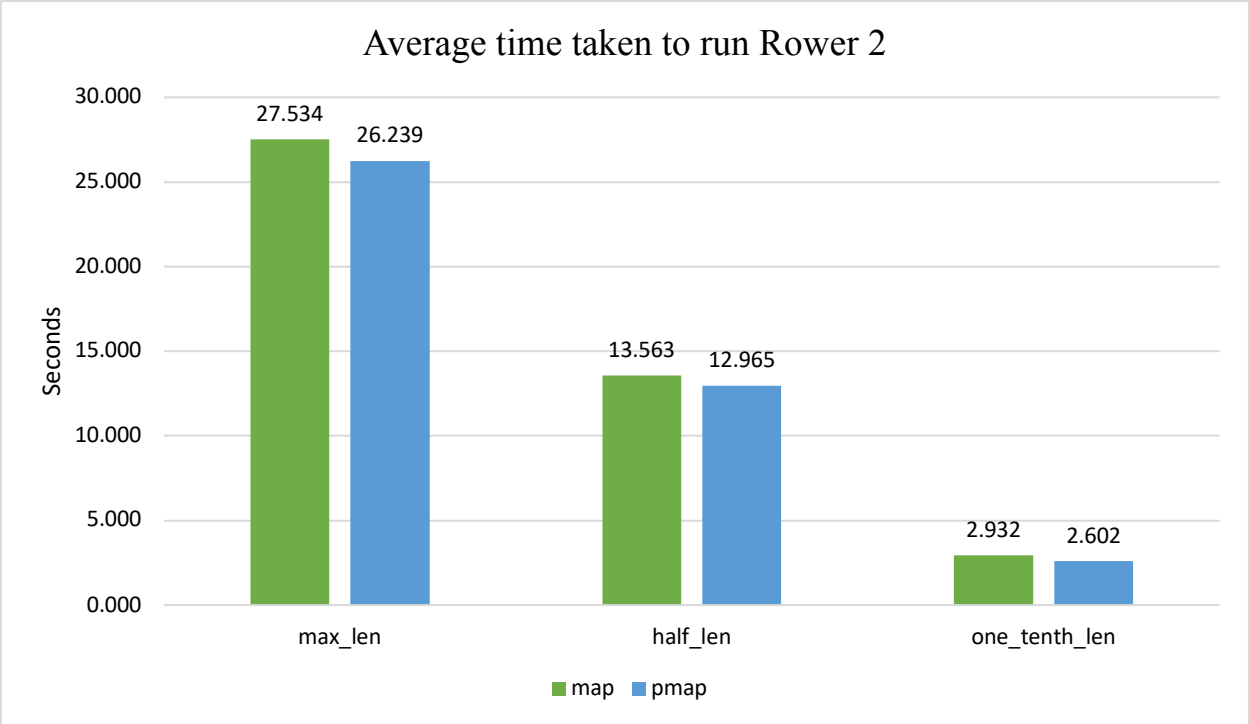

Figure 4. Average time taken to run Rower 2 on Ethan's Laptop.

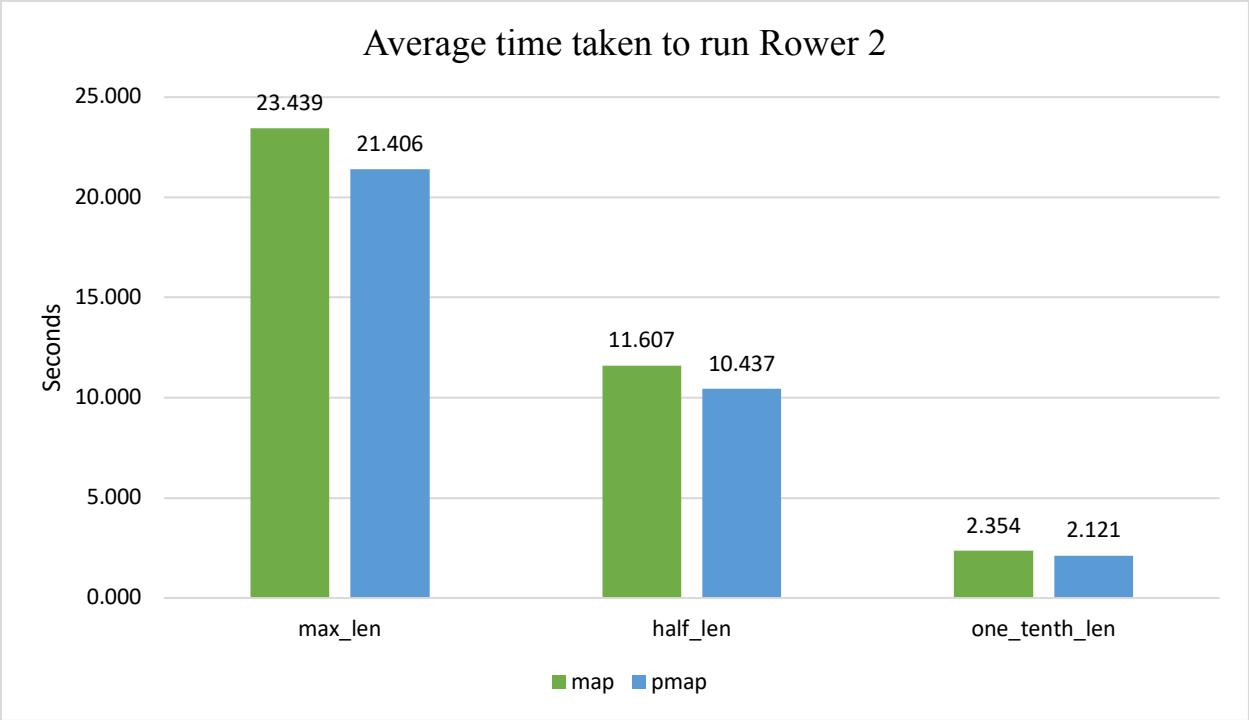

Figure 5. Average time taken to run Rower 2 on Divit's Laptop.

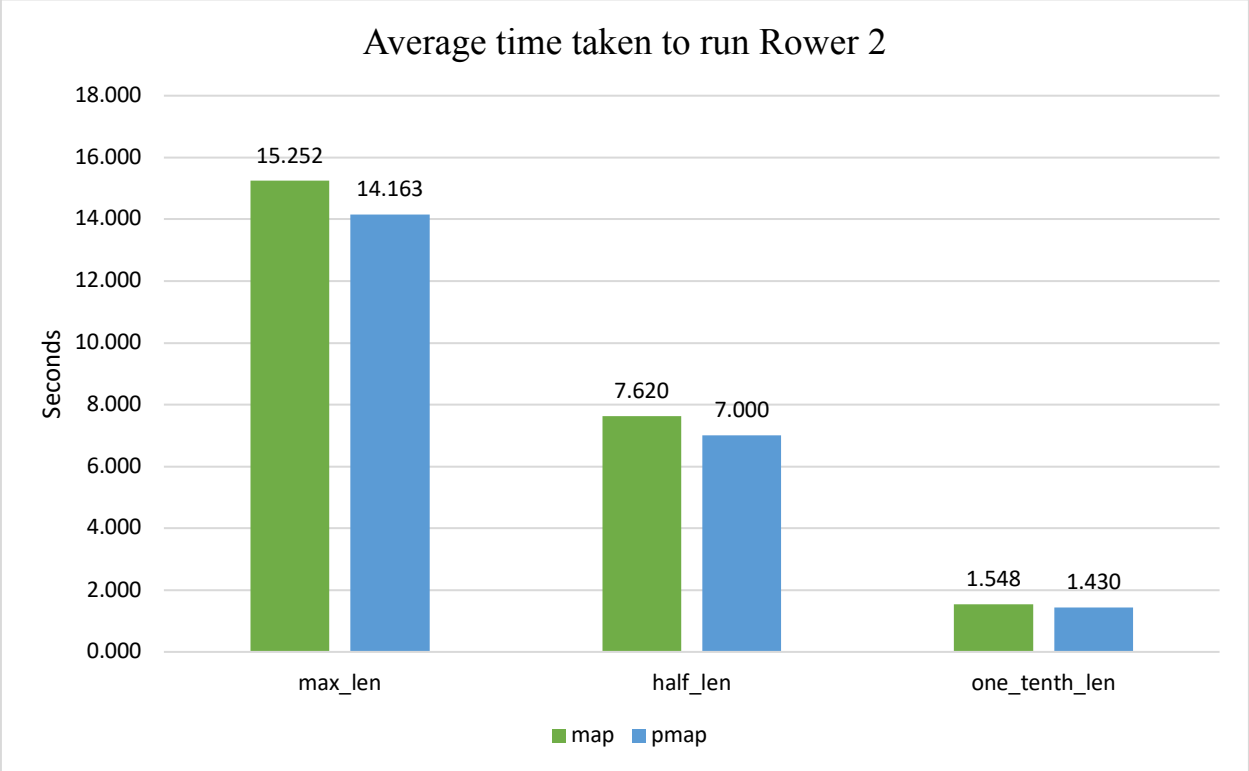

Figure 6. Average time taken to run Rower 2 on Divit's PC.

Conclusion

Our test result verified what we already hypothesized about **map** and **pmap** function. In general, across different software and hardware configurations, the **pmap** function will consistently perform better. However, we did notice that when the Rower subclass has an expensive join_delete function, the performance of our **pmap** function decreases significantly due to the way we implement it. Since performance is not greatly affect, this would be a good reason to consider using **map** over **pmap** with expensive joins because the CPU cost is sometimes not worth the marginal speed up.