Ashley Teow and Justin Torre

CS4500: Assignment 5 [Part 1]: Concurrency

https://github.com/ashleyteow/sw-dev-a4

Introduction

The goal of this report is to evaluate the experiment we designed to measure the performance increase of using our DataFrame's `pmap()` function over its `map()` function - on executing our two custom Rower subclasses we created based off the dataset we chose to work with. Our DataFrame is a table composed of columns of equal length, where each column can only contain values of one type (Int, String, Bool, Float). The experiment's dataset is over 100MB in size with a total of 15 columns and is considered large enough to be able to get some reliable results from our experiment.

Additionally, this report will highlight how we chose to execute mapping in parallel for `pmap()` as this function is expected to perform as well or better than `map()`. The results for the experiment will be displayed in graphical form, followed by a comparison analysis of these results.

Implementation

Pmap is a mapping function that will map over our dataframe and apply a function defined by a Rower visitor. Pmap, is special because it uses any N number of threads to apply our rower on our dataframe. This function essentially chunks each of the dataframe number of rows by however many threads are defined by the program. We build a set of ranges for our threads to map over in our dataframe and at the end of our pmap we just wait for all the threads to finish. For any R number of rows we will split up the workers so that each thread will handle R/N rows. We essentially create an array of threads like this:

```
std::thread* threads[TC];
```

Where TC is the number of threads we need to spawn. We then call each thread like this:

```
    threads[i] = new std::thread([&r, s_i, e_i, this] {
      this->pmap_range ( s_i, e_i, r );
    });
```

After each thread is called, all the threads wait for each other to finish with the join function. The map changes the dataframe in place and is ever only concerned about the rows it is scheduled to handle, so data races should not be a concern.

Description of the analysis performed

Type of Data:

The bluebikes trip data includes:

- (Int) Trip duration in seconds
- (Int) Start time and date
- (Int) Stop time and date
- (Int) Start station id
- (String) Start station name
- (Float) Start station latitude
- (Float) Start station longitude
- (int) End station id
- (String) End station name
- (Float) End station latitude
- (Float) End station longitude
- (int) Bike id
- (String) Usertype
- (int) Birthyear
- (int) Gender

File size = 102 MB

This data was acquired from the BlueBikes system data and it provides information on every BlueBikes trip made by a user around BlueBikes stations locations all over Boston. More specifically, we took the dataset for January 2015 and February 2015 and combined them so that the final data file was over 100MB in size with a total of 15 columns. This specific subset of the dataset offers insight on the duration of BlueBikes trips along with their time and dates, the start and end locations of each trip, their bike ID, whether the user is a casual (one time / day pass) user or a member (monthly / annual) user, the user age by birth year, and the user's gender (self-reported).

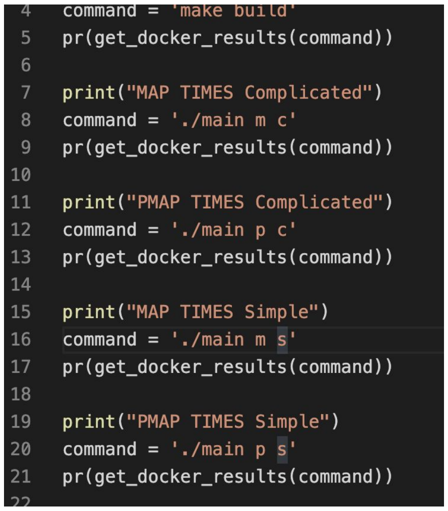
For our analysis, we defined two Rower subclasses; one that is a relatively simple inexpensive task to execute on the dataset, and the other is a far bit more expensive than the first one. Our SimpleTaskRower class rounds the latitude and longitude values in the dataset to the nearest whole integer. With some arithmetic, we simply implement the accept method to cast the floor into the int that we want to return if the column we are looking at is a float.
Our ComplicatedTaskRower is a little more involved in that changes the birthyear column to equal their age as of 2020, it converts the tripduration column to be represented in minutes instead of seconds, it adds 1 to each char for String values, in addition to rounding floats to the nearest whole number.

We decided to use user time, real time, system time and memory usage as benchmarks to measure the performance of these two functions on our two Rower subclasses as these are the most useful measurements when comparing two implementations, to find out which is fastest.

Comparison of experimental results

In order to generate these two bar charts in *Figure 1* and *Figure 2,* we ran a python script in our `docker_runner.py` file that executed these functions within a docker container and is built through the Makefile. Something to note is that these times are end-to end, meaning that it includes the time taken to unzip and read the csv file as well. We tried to account for this and used our system time variables to measure the performance before and after calling our map functions.

Our test suite for this benchmark analysis consisted of the following commands to print the results we wanted for these 4 runs:



```
command = 'make build'
pr(get_docker_results(command))

print("MAP TIMES Complicated")
command = './main m c'
pr(get_docker_results(command))

print("PMAP TIMES Complicated")
command = './main p c'
pr(get_docker_results(command))

print("MAP TIMES Simple")
command = './main m s'
pr(get_docker_results(command))

print("PMAP TIMES Simple")
command = './main p s'
pr(get_docker_results(command))
```

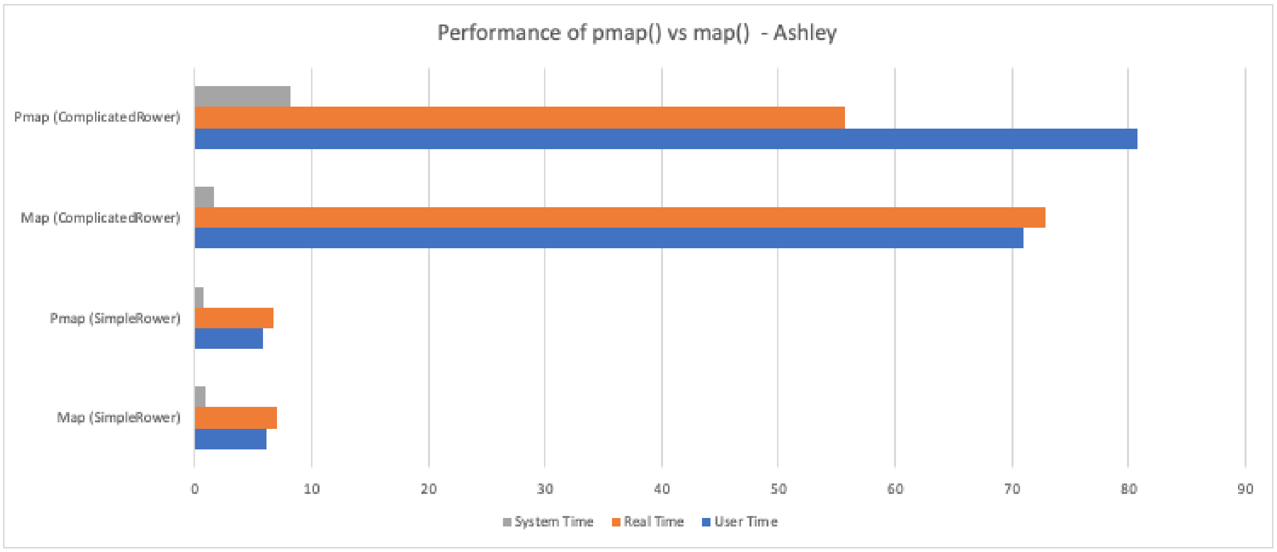


Figure 1: Performance Analysis using user time, real time, and system time. This analysis was run on Ashley's computer inside a docker container. [CPU = 1.6 GHz Dual-Core Intel Core i5, Memory=16 GB 2133 MHz LPDDR3, OS=10.15.1 macOS Catalina, Docker = 2.1.0.5]

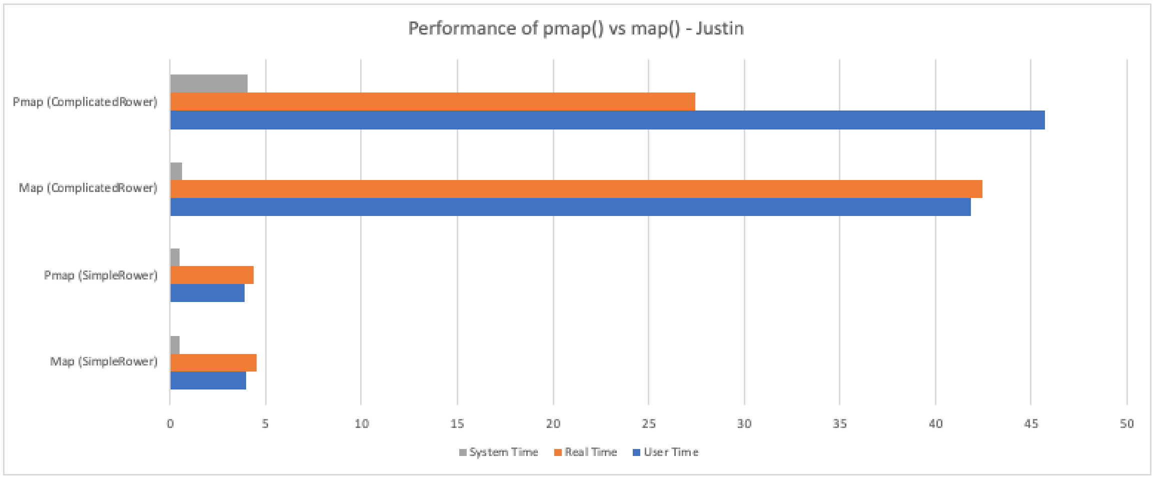


Figure 2: Performance Analysis using user time, real time, and system time. This analysis was run on Justin's computer inside a docker container. [CPU = 2.6 GHz Intel Core i7, Memory=16 GB DDR3, OS=10.14.3 macOS Mojave. Docker = 2.1.0.5]

The results from both tests on our computers show that our map() function is actually faster than our pmap() function for both custom Rower subclasses. Justin's was faster at executing these functions than the run on Ashley's computer but the general pattern still follows.

There was some manual testing done using `clock`. In this testing we printed the clock time before and after our maps and found the following results:

	Time in milliseconds
pmap Simple	962445
map Simple	667445
pmap Complicated	10083174
map Complicated	8034112

Figure 2: Performance Analysis using user time, real time, and system time. This analysis was run on Justin's computer. [CPU = 2.6 GHz Intel Core i7, Memory=16 GB DDR3, OS=10.14.3 macOS Mojave]

As you can see from the table above, pmap performs worse than map. This does not seem to make sense, we may however want to use another timing method in the future.

Threats to validity

There are several ways our results could differ from someone else's. For instance, there could be a misinterpretation of the API in which these results may not make sense. Additionally, we gathered from these results that pmap() was slower, which was an unintended result and could indicate that its implementation is slightly too inefficient. Another threat to validity is whether or not using `clock` is a valid method for measuring our benchmarks. Additionally in our test we run our timing on a docker container and docker may throw in some variability with thread management that we are unaware of.

Conclusions

After measuring the performance increase of using our DataFrame's `pmap()` function over its `map()` function on executing our two custom Rower subclasses we designed for our Bluebikes dataset, we have concluded that our pmap() function is not as efficient as our map() function. Our implementation of pmap() is still too slow to be able to overtake map() in the long run. Due to our time constraints we were not able to debug this issue, and pmap() should clearly be faster than map(). In the future we plan on debugging this issue, but we are confident in our testing pipeline to ensure reliable test driven development for future iterations of pmap().