Assignment 5 Part 1 Memo

Kate Rupar and Mark Thekkethala

Introduction

The goal of this report is to present an end-based study of our pmap method implementation and explore the impact of threading in general on the speed of programs. In this report we will demonstrate that our pmap design heavily outperforms map in runtime. In the following section we will present our benchmark tests to evaluate the speed of pmap in comparison to map, as well as describe our thought-process and design choices. We will also present our findings graphically and acknowledge potential sources of error.

Implementation

To provide some context, we will first discuss our DataFrame design. After we will introduce our solution for map and lastly describe our pmap implementation.

Brief Overview of our API Implementation

Columns

The Column class is extended by four classes: IntColumn, BoolColumn, FloatColumn, and StringColumn. Each column class contains an array of its type. Columns can return their elements and their type.

Rows

The Row class contains an Array of Object, which contains Integers, Floats, Bools, and Strings. Rows also have a size which represents the number of elements as well as an index which represents the position of a Row in a DataFrame. Rows can set elements and return elements. They are constructed using a Schema.

DataFrames

DataFrames contain a Schema, Array of Columns, and a number of Rows and columns. Aside from adding columns, you can also add Rows. The map and pmap methods exist inside the DataFrame class.

Schema

A Schema contains a String representing the types of a DataFrame (ex: “IBFS”). Schemas also contain the Row and column headers in arrays as well as have a length and width.

Rower

A Rower iterates through each Row of a DataFrame. Rowers have an accept method which determines how they interact with Rows. They also have a join_delete method which cleans up memory and is utilized when joining threads.

map

Our map function can be found in the DataFrame class in modified_dataframe.h. The map function takes in a Rower which calls accept on every Row of the dataframe. In this method we have to construct the Rows, since a DataFrame holds a list of Columns. Rowers can have multiple purposes depending on their design; it depends on the functionality of their accept method. For our benchmark test we designed two Rowers: sumRower and printStringRower which we will discuss in more detail in the later sections.

pmap

According to the given spec, pmap clones the given Rower and executes map in parallel. To clone the Rower, we added a method in the interface, clone(), which returns a Rower. In that method, the classes that inherit Rower will return a new instance of themselves.

Our team utilized the Thread interface (thread.h) provided by the instructors on Piazza. The Thread class uses the <thread> library and the class provides us with the basic functionality of the library. We created the class Thread_Map which inherits from Thread. Thread_Map is constructed by passing in an array of column* from a DataFrame, a Schema, a Rower which determines the functionality of the thread, and a beginning and ending index. The beginning and ending index are used so that multiple threads will not interfere with each other; in other words the indexes are used to split up the work. Other than the constructor, the only other method in Thread_Map is run; the rest are inherited from Thread.

In the pmap method, we create two threads, one for each Rower (the one passed to the method and its clone). We find the halfway point of the number of rows in this Dataframe, and give each thread half the Rows to accept. When we call start on the threads in the pmap method, the run method is triggered. Run creates Rows from the columns that Map_Thread was constructed with and passes each one to the Rower’s accept method. After all of the Rows from the beginning to ending index have been visited by the Rower, the two threads are joined and the results of the mapping are combined using the Rower method join_delete.

Description of the Analysis Performed

Generating Data

Since the requirement was to utilize a 100 MB data file with at least 10 columns, we manually created our data.txt file using a simple Python script, genData.py (included in the submission). The script creates a 10 column CSV file 6 million lines long (~100 MB) with three line variants. To create a representative data file, we included String, Float, and Integer types. data.txt is utilized in our bench.cpp file. The data.txt zip file is located at https://github.com/rupark/file. It is retrieved when “make run” is called and deleted after execution of our program.

Two Rower Subclasses

We designed two Rower subclasses for testing purposes: printStringRower and sumRower. printStringRower prints all of the Strings longer than 8 characters and sumRower finds the sum of all of the ints in the file. In join-delete, sumRower adds the other Rower’s sum to its own sum and deletes the other Rower.

Hardware

We utilized two devices to run our benchmark tests: a MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports) running macOS Catalina and a Snell Library iMac (Retina 5K, 27-inch, Late 2015) running macOS Mojave. To maintain consistency, the MacBook Pro was charging during every execution.

benchmark.cpp

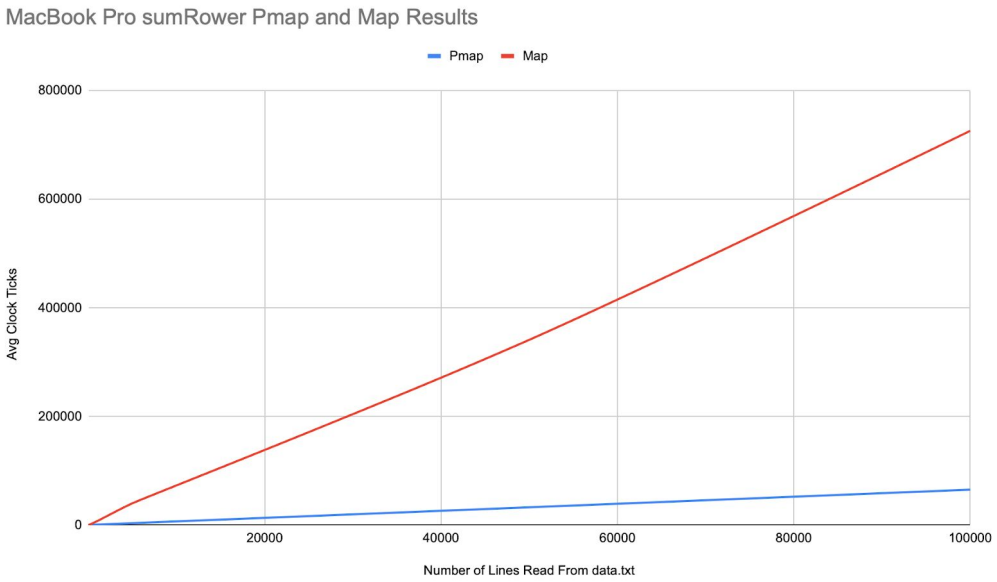
Our bench.cpp file is where we test the speed of map and pmap. To begin, we read in a certain number of lines from data.txt, break up each line by the delimiter (“,”), create Rows from the elements, and add the Rows to two identical DataFrames. The schema string we utilized was "SSSSIISFFI" in accordance with our data.txt file. After the DataFrames were created, we utilized the <time.h> library. We call the clock() method before and after running map and pmap. We then subtract the values to find the number of clock ticks that passed. We repeat this process for the two Rower subclasses and for different amounts of file lines. We averaged the runtime for each Rower/size combination over three executions to account for variations. To ensure that one Rower didn’t impact the performance of the other, we have two DataFrames, one for each Rower. After, our program prints out the number of clock ticks it took to execute pmap and map for sumRower and printStringRower.

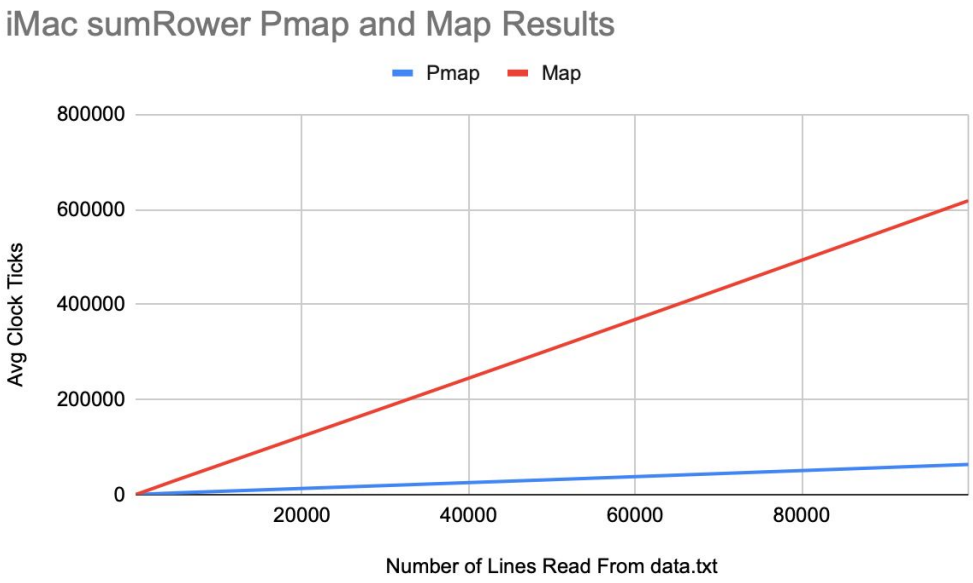
Comparison of Experimental Results

Results

sumRower

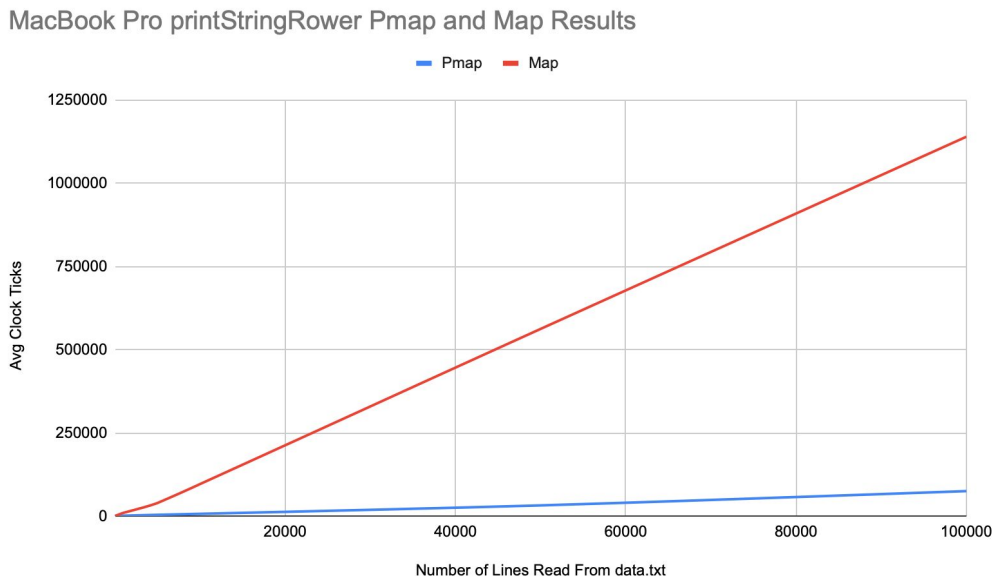
The following are the results from running pmap and map using a sumRower on Macbook Pro and iMac.

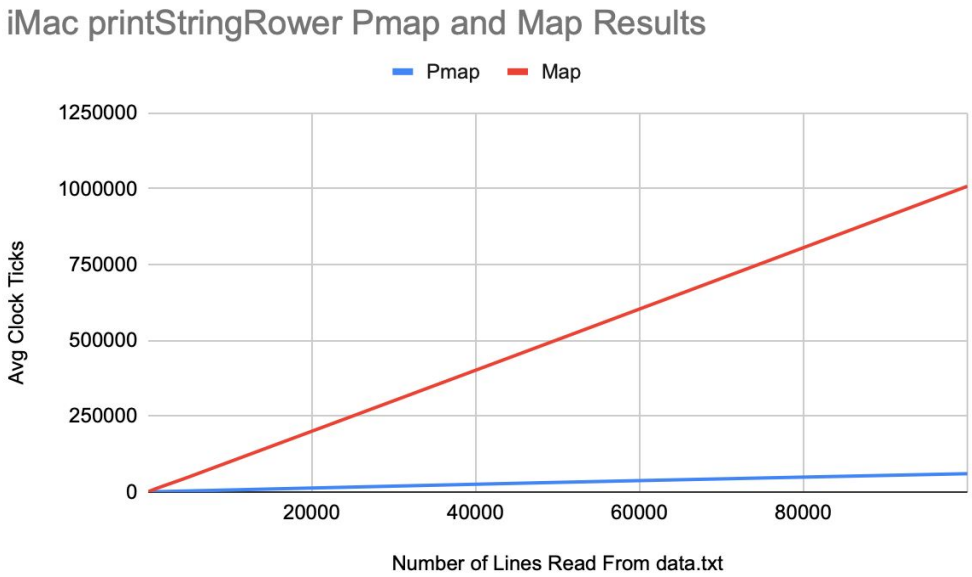




printStringRower

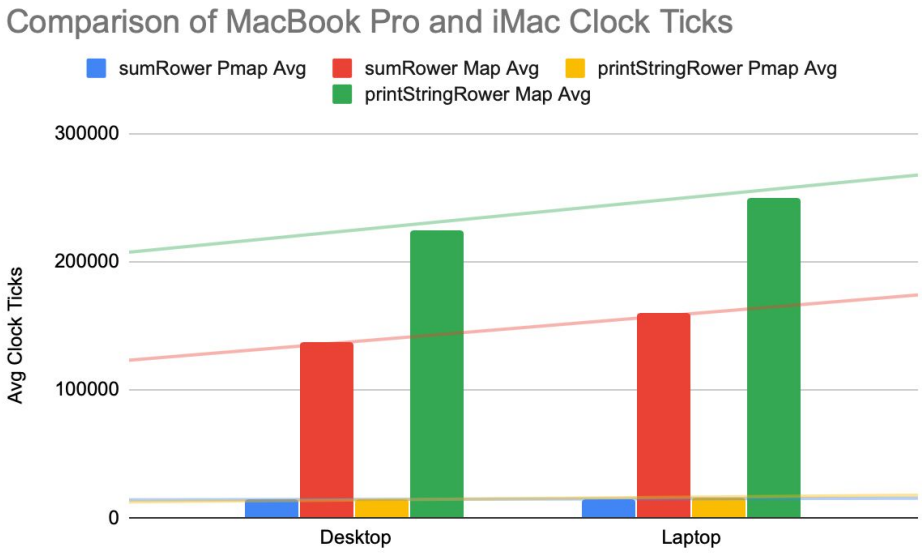
The following are the results from running pmap and map using a printStringRower on Macbook Pro and iMac.





Hardware Comparison

The following graph shows the averages for each Rower and method split by device.



Discussion

It is evident from viewing these graphs that pmap is able to execute the work in a fraction of the time map can. In every instance, the slope of the map line is significantly greater than the pmap line. This is because threading enables parallel execution, and two Rowers can simultaneously visit the DataFrame Rows reducing the runtime. From viewing these graphs, it is also apparent that the MacBook takes slightly more time to execute than the iMac. This is unsurprising, since laptops have smaller components which don't usually perform as well as larger versions. Laptops are also designed for mobility, so processors are typically designed for power-saving causing them to run more slowly. From our findings, it is also clear that printStringRower takes a lot longer to execute than sumRower. This is because printStringRower calls cout every time it finds a string longer than 8 characters in its accept method. This causes printStringRower to have a longer execution time than sumRower. This distinction between Rower performance is best seen in the final graph of the previous section, "Comparison of Macbook Pro and iMac Clock Ticks".

Potential Sources of Non-Determinism

The runtime environment and runtime elements are all sources of nondeterminism. The processor management system affects the timing recorded. Resources such as memory and the CPU are also utilized by other OS processes which may have an impact on the performance of the program. Multi-threaded programs are non-deterministic since the execution of a thread relies on environmental conditions.

Threats to Validity

There are a few threats to the validity of our experiments. For one, the type of OS and device could impact the runtime of programs since we only used macOS in our tests. Another factor is that the programs may run differently when a laptop is charging than when it is not. To mitigate this, we only executed the program when our MacBook Pro was connected to power.

Conclusions

In this memo we have demonstrated the effectiveness of threaded programs. After averaging all of the tasks, we found that utilizing pmap reduces the average number of clock ticks by approximately 75%, which is a significant amount. The success of pmap persisted across both of our tested devices. From this study we can conclude that threading is important in order to develop time efficient programs.