Assignment 5 report:

Introduction:

Our goal is to test the difference in performance between using multi-threaded, and single-threaded programs. Our expected result is that multi-threaded will perform better when the data size is larger (around 10000 rows), as well as when the computation for each row is more complicated. Most of the difference between multi-threaded and single-threaded comes from the startup costs. That is, for multi-threaded programs to be advantageous the time spent running the equation must be more of a cost than the time to allocate the memory and launch the threads. To test this we created an experiment using the modified data frame from last assignment, and compared the performance between using map (single-threaded) and pmap (multi-threaded). For each experiment, map or pmap will take a same rower to use on the same data.

Implementation of pmap():

The design behind pmap() is simple. Since multi-threaded operations cannot depend on sequential access, it can make the assumption that each Rower can cover a different part of the dataframe. First it decides how many threads to use (if the dataframe is too small it instead runs it as a single threaded operation), up to a maximum of 8 threads. This number is fairly arbitrary, and depends on computer. In this case, it was tested on a 4-core intel laptop with 8 virtual cores, so 8 threads was picked as a safe maximum thread value. It was also tested on the CCIS linux servers, which have significantly more cores. More testing is needed to truly choose this value, and it will depend on machine. It then generates a rower for each thread using clone(). Next it divides up the dataframe in about equal amounts per thread, and launches a new thread to parse that number of rows in the database (using _pmap_helper()). Following that, it joins the threads, combines the rowers, and then cleans up the memory it allocated. It attempts to not use too many threads for too small a database, while still seeing a speed-up. More testing it needed to determine the true number of threads that should be used, and the amount of rows that each thread should cover at a minimum to see a substantial speed increase.

Description of the analysis performed:

Neil's computer configuration:
Intel i5-6300U CPU@2.4GHz, 64 bits
Kernel: 5.51- arch1-1
RAM: 8GB

Lab computer configuration:
Intel® Xeon® Gold 5118 CPU @ 2.30GHz
2 sockets,

12 cores per socket.

Datafile.txt:

Data url: https://www.kaggle.com/sobhanmoosavi/us-accidents

Data is from a Kaggle competition and informed the traffic accidents which covers 49 states of the U.S. The data is collected from February 2016 to December 2019, using several data providers, including two APIs that provide streaming traffic incident data. These APIs broadcast traffic data captured by a variety of entities, such as the US and state departments of transportation, law enforcement agencies, traffic cameras, and traffic sensors within the road-networks. Currently, there are about 3.0 million accident records in this dataset.

Size of the file: 102.6MB
Data shape: 49 columns, 297432 rows

Columns information:

ID This is a unique identifier of the accident record.
SourceIndicates source of the accident report (i.e. the API which reported the accident.).
TMCA traffic accident may have a Traffic Message Channel (TMC) code which provides more detailed description of the event.
SeverityShows the severity of the accident, a number between 1 and 4, where 1 indicates the least impact on traffic (i.e., short delay as a result of the accident) and 4 indicates a significant impact on traffic (i.e., long delay).
Start_TimeShows start time of the accident in local time zone.
End_TimeShows end time of the accident in local time zone.
Start_LatShows latitude in GPS coordinate of the start point.
Start_LngShows longitude in GPS coordinate of the start point.
End_LatShows latitude in GPS coordinate of the end point.
End_LngShows longitude in GPS coordinate of the end point.
Distance(mi)The length of the road extent affected by the accident.
DescriptionShows natural language description of the accident.
NumberShows the street number in address field.
StreetShows the street name in address field.
SideShows the relative side of the street (Right/Left) in address field.
CityShows the city in address field.
CountyShows the county in address field.
StateShows the state in address field.
ZipcodeShows the zipcode in address field.
CountryShows the country in address field.
TimezoneShows timezone based on the location of the accident (eastern, central, etc.).

Airport_CodeDenotes an airport-based weather station which is the closest one to location of the accident.
Weather_TimestampShows the time-stamp of weather observation record (in local time).
Temperature(F)Shows the temperature (in Fahrenheit).
Wind_Chill(F)Shows the wind chill (in Fahrenheit).
Humidity(%)Shows the humidity (in percentage).
Pressure(in)Shows the air pressure (in inches).
Visibility(mi)Shows visibility (in miles).
Wind_DirectionShows wind direction.
Wind_Speed(mph)Shows wind speed (in miles per hour).
Precipitation(in)Shows precipitation amount in inches, if there is any.
Weather_ConditionShows the weather condition (rain, snow, thunderstorm, fog, etc.).
AmenityA Point-Of-Interest (POI) annotation which indicates presence of amenity in a nearby location.
BumpA POI annotation which indicates presence of speed bump or hump in a nearby location.
CrossingA POI annotation which indicates presence of crossing in a nearby location.
Give_WayA POI annotation which indicates presence of give_way sign in a nearby location.
JunctionA POI annotation which indicates presence of junction in a nearby location.
No_ExitA POI annotation which indicates presence of no_exit sign in a nearby location.
RailwayA POI annotation which indicates presence of railway in a nearby location.
RoundaboutA POI annotation which indicates presence of roundabout in a nearby location.
StationA POI annotation which indicates presence of station (bus, train, etc.) in a nearby location.
StopA POI annotation which indicates presence of stop sign in a nearby location.
Traffic_CalmingA POI annotation which indicates presence of traffic_calming means in a nearby location.
Traffic_SignalA POI annotation which indicates presence of traffic_signal in a nearby location.
Turning_LoopA POI annotation which indicates presence of turning_loop in a nearby location.
Sunrise_SunsetShows the period of day (i.e. day or night) based on sunrise/sunset.
Civil_TwilightShows the period of day (i.e. day or night) based on civil twilight.
Nautical_TwilightShows the period of day (i.e. day or night) based on nautical twilight.
Astronomical_TwilightShows the period of day (i.e. day or night) based on astronomical twilight.

Tasks:

SimpleRower:

Simple rower chooses four columns(different type of column, one for each) from the one of the row in the dataframe and does the following work:

1. Severity(int): adds one.
2. Distance(float): multiplies to convert from miles to kilometers.
3. Amenity(bool): updates a Boolean.
4. Description(String): stores it to a variable.

ComplexRower：

The complex rower will loop through all of the fields in the row and bring every field to a fielder to test if there is any change should be made. There are four edited fielder as the following:
First fielder: keep storing the sum of all the integer columns and float columns.
Second fielder: storing how many floats are related to the information of weather.
Third fielder: storing the record of place of interest (POI). if there is a related POI return true, otherwise false.
Fourth fielder: storing all of the strings in a string array.
After we loop through all of the fields, the information we get from the fielder is stored in members of the class.

Experiment:

We created a new bench.cpp with running the data, and we selected three benchmarks as the number of rows to run the data we want to test. The max rows of the datafile.txt is 297,432 lines. Our bench mark is the following:

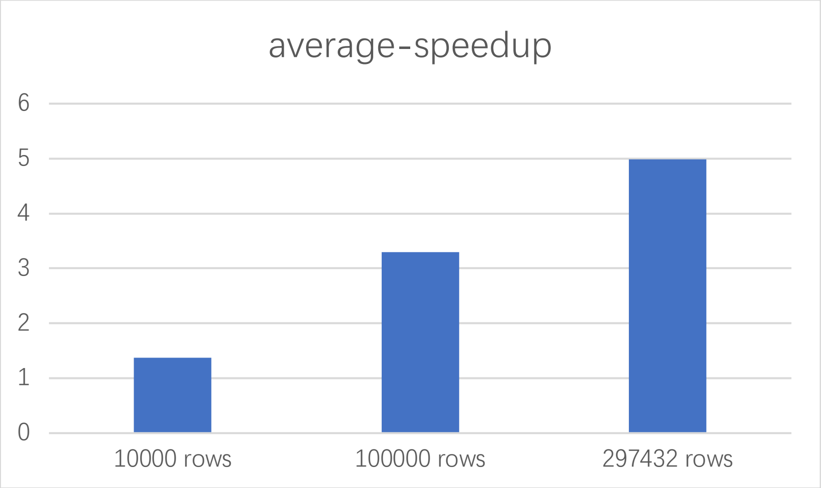
```
#define SIZE_ONE    10000
#define SIZE_TWO    100000
#define SIZE_THREE   297432
```

And for each experience in each rower we do 100 times loop and record the time when we use map() and pmap().

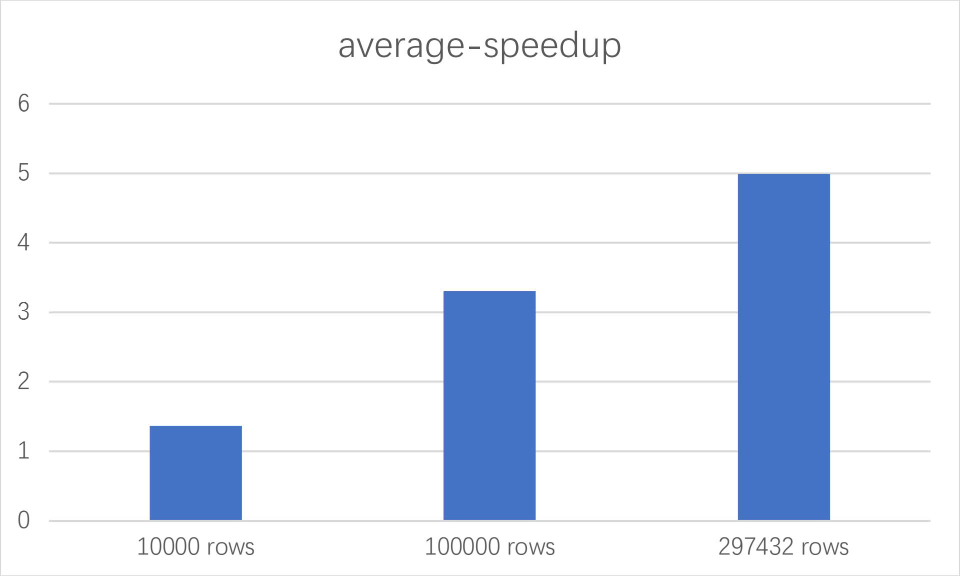
Comparison of experimental results:

We uses average speedup to measured the performance which is:
[Single-threaded performance / multi-threaded performance] / total-loop numbers

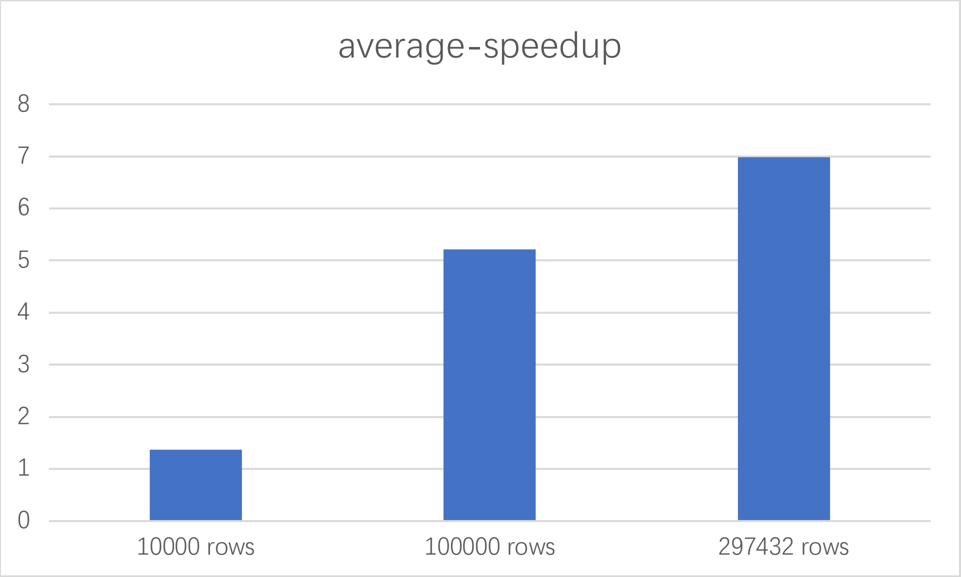
SimpleRower Comparison result:
On Neil’s laptop:



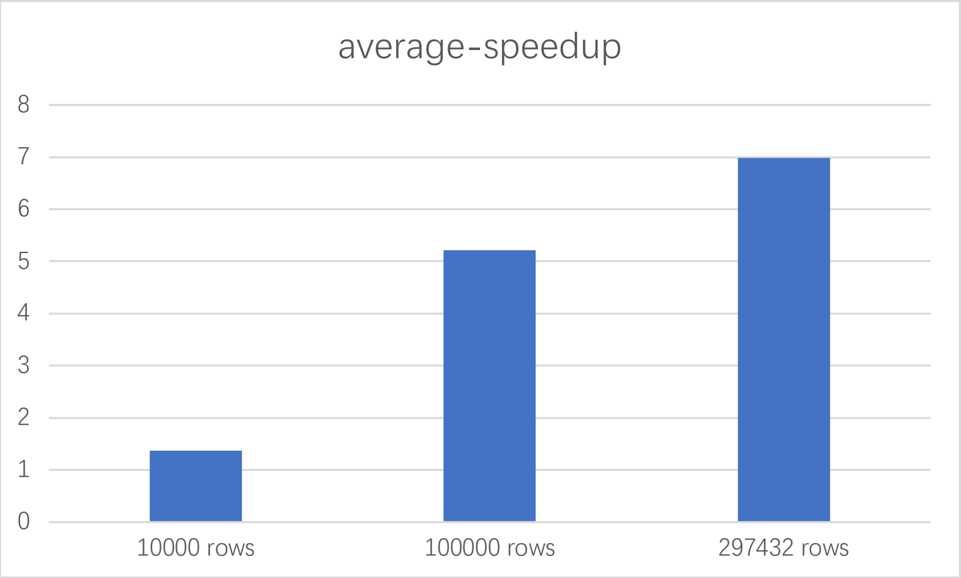
On lab's computer:



ComplexRower Comparison result:
On Neil's laptop:



On lab’s computer:



Threats to validity:

The factors that could cause different results are listed below:

1. Different hardware could cause the different of performance. A newer and a better CPU could perform better.
2. Different systems could vary the performance as well, include standard libraries, memory allocator implementation. It could also depends on Linux, Mac OSX, and Microsoft windows.

Conclusion:

The result outputs very obvious, pmap performs better than map for almost every case, and when it runs all rows of the dataset (297432 rows), we see a speed-up of around 2x.