Introduction:

Author: Xiaocheng Zhang | Tian Xia

This is a report about:

1. How we implement a dataframe which supported multiple threads running:
    - Implement a simple DataFrame follow the given API from CS4500 Assignment (Northeastern University).
    - Improve a method called *map(Rower &r)* which allows users to visit through the dataframe row by row.
    - Implement a new method called *pmap(Rower &r)* which allows users to visit through the large dataframe (contains many rows) more efficiently.
2. How dataframe use multiple threads to work:
    - The dataframe saved data in columns with different data type(String, int, float, bool).
    - A class type Rower is a visitor within some functions to do with dataframe.
    - *pmap(Rower &r)* will compute the length of dataframe and consider how many rows in a chunk it will be given to one thread.
    - Generate multiple threads and allocate jobs.
    - Collect all Rowers' data into original Rower &r by method *join_delete(Rower *other)*.
3. How much efficiency *pmap* can improve:
    - We ran the example mutilple times on different laptop.
    - Found that the more rows a dataframe has, the less time *pmap* used in contrust to *map*.

Implementation Description:

Code review of *pmap(Rower &r)*:

Part1:

```
/** This method clones the Rower and executes the map in parallel. Join is
  * used at the end to merge the results. */
 void pmap(Rower &r) {
   size_t row_num = nrows();
   size_t split_num = 0;
   size_t thread_num = 0;
```

> size_t row_num = nrows();

- Number of total rows in this dataframe
- Generated by method *nrows()*

```
/** The number of rows in the dataframe. */
  size_t nrows() { return schema_->length(); }
```

- `schema_` saved all format information, including a vector of each row's name.
- `length()` return the length of vector of row's name.

> size_t split_num = 0;

- Initialized as 0 (1 is ok).
- This number used to record the number of rows that will be placed into one thread.

> size_t thread_num = 0;

- Initialized the number of threads only.

Part2:

```
if (row_num > 7) {
  thread_num = 7;
  split_num = row_num / thread_num;
} else {
  split_num = 1;
  thread_num = row_num;
}
size_t offset = row_num - thread_num * split_num;
```

- If `row_num > 7`, in this case, we will use 8 threads to finish visiting.
- The reason that we set `thread_num = 7`:
    - We are using integer division which will give a value `split_num * thread_num <= row_num`.
    - In order to divide rows retionally, we deduced 1 on `thread_num` to get an `size_t offset = row_num - thread_num * split_num;`

Part3:

```
std::thread threads[thread_num + 1];
Rower **rower_array = new Rower *[thread_num + 1];
for (size_t i = 0; i <= thread_num; ++i) {
  size_t start_idx = i * split_num;
  size_t end_idx = 0;
  if (i != thread_num) {
    end_idx = (i + 1) * split_num - 1;
  } else {
    end_idx = start_idx + offset - 1;
  }
  rower_array[i] = dynamic_cast<Rower *>(r.clone());
  threads[i] = std::thread(thread_method, rower_array[i], table_, schema_,
                           start_idx, end_idx);
}
```

- Call `std::thread threads[thread_num + 1]` to initialize an array of threads

with size `thread_num + 1`. In the example above, it creates a `std::thread threads[8]` which start from index 0 - 7;

- Call `Rower **rower_array = new Rower *[thread_num + 1];` to initialize an array of Rower with size
- Call for loop to split threads.
- `if (i != thread_num)`
    1. if `i != 7`, this is not the final thread. So we don't need to consider about remain rows which saved as `offset`. `end_idx = (i + 1) * split_num - 1;` set end index of row for current thread.
    2. if `i == 7`, we are in last thread now, so we need to consider remain rows. `end_idx = start_idx + offset - 1`

```
rower_array[i] = dynamic_cast<Rower *>(r.clone());
      threads[i] = std::thread(thread_method, rower_array[i], table_, schema_,
                               start_idx, end_idx);
```

- set array of rower a new rower clone from passed in value &r from `pmap(Rower &r)`. Save rowers because we will call `join_delete(Rower *other)` to collect all rower's information soon.
- set array of threads with method `thread_method`, it will let rower to visit rows from `start_idx` to `end_idx`.

Part4:

```
    for (size_t i = 0; i <= thread_num; ++i) {
      threads[i].join();
      r.join_delete(rower_array[i]);
    }
    delete[] rower_array;
  }
```

- for loop to wait threads end and collect all data into original Rower.

Description of the analysis performed:

The data where we perform our analysis is the HighD dataset. It is published at IEEE ITSC 2018. The data in this dataset is collected by drone in different sections of a German highway. The dataset records some driving behavior such as each vehicles x and y position, longitudinal and lateral velocity, accelerations, and its surrounding vehicle's information. The data type used in this dataset is mainly float.

dataframe looks like this:

frame	id	x	y	width	height	xVelocity	yVelocity	xAcceler
1	1	55.63	12.61	4.35	1.92	-35.08	-0.10	-0.26
2	1	54.39	12.61	4.35	1.92	-35.09	-0.10	-0.26
3	1	53.01	12.60	4.35	1.92	-35.10	-0.10	-0.26

Our program will go through each row by calling function object rower in `rower&fielder.h`:

```
class AdvanceRower;
//too long to display them...
class TestRower;
```

AdvanceRower will save all strings in this row and form a dataframe (This rower has been tested by trivial initialized dataframe, we used two for loop to create a full dataframe with four different types).
`Rower.join_delete` method will collect all saved string colomn into a dataframe.

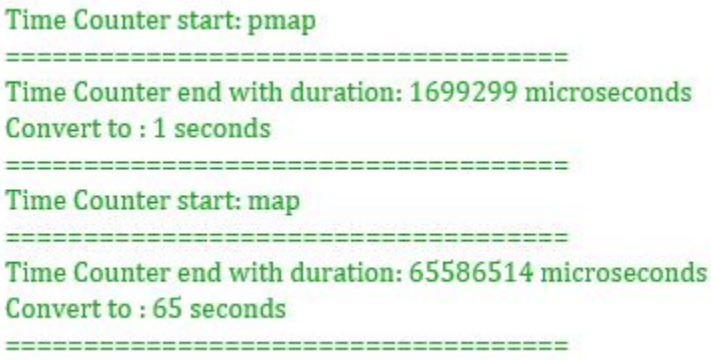
TestRower will save all types of data into four types vectors with some small differences (such as some computation during the collection).

In `bench.cpp`, we tested output of Rower that be passed in by `pmap` and `map`. Used `t_true` method to check whether two method provide the same answer.

Comparison of experimental results:

Record	Laptop 1	Laptop 2
Processor Name	Intel Core i7	Intel Core i5
Processor Speed	2.2 GHz	2.6 GHz
Number of Processors	1	1
Total Number of Cores	4	2
Memory	16 GB	8 GB
Speed of pmap (average)	1.646387 s	4.1969805 s
Speed of map (average)	65.370497 s	121.148712 s

We used two different laptops with different specifications to test the speed of `pmap` and `map`.
The deffences is clear from the form.
This is a screen shot of speed selected from 5 times running by Laptop1:



- `pmap` with 8 threads in this laptop spent nearly 1 second, 1,699,299 microseconds.
- `map` with 1 thread in this laptop spent nearly 54 second, 65,586,514 microseconds.

This is a screen shot of speed selected from 5 times running by Laptop2:

```
unzip CS4500-as5-only-/datafile.zip
Archive:  CS4500-as5-only-/datafile.zip
  inflating: datafile.txt
./bench datafile.txt
Time Counter start: pmap
====================================
Time Counter end with duration: 4210568 microseconds
Convert to : 4 seconds
====================================
Time Counter start: map
====================================
Time Counter end with duration: 119277327 microseconds
Convert to : 119 seconds
```

- `pmap` with 8 threads in this laptop spent nearly 4 second, 4,210,568 microseconds.
- `map` with 1 thread in this laptop spent nearly 119 second, 119,277,327 microseconds.
- Conclusion: Laptop1 is nearly 4 times speed fast to Laptop2.

Threats to validity:

The data's restriction:

1. Our *bench.cpp* only support read *.txt file with **float** data, because we did not combine this assignment with assignment1.
2. Since we did not handle so many cases from assgnment1, during the process of read file, it will automatically **read through all the data** and form dataframe at the same time. So the datafile's size should not be too large (100 MB is ok), or it will cause a Memory overflow.
3. Besides, the first line will be read as **title of column** automatically.

Threads restriction:

1. In order to speed up its normal speed, each Rower is treated as independent. If Rowers are dependent (one rower will based on previous

data), it will cause problems such as data lost (Since we did not use Mutex to restrict rower or threads).

Rower restriction:

1. Rower must provided with the clone method. This is the only way we copy the original Rower, or the program will crash with Segmentational fault.

Conclusions:

To summary, our pmap perform great since we don't use dependent Rower. When there are more rows, `pmap` looks more efficient than `map`.
There are also some cases that better to choose regular `map` rather than `pmap`. Because allocating threads also needs time, if the number of rows in the dataframe is not quite large, it is inefficient to use `pmap`.
To make sure the program runs, we need to check the file before we map it, and the specific rower is also required.