STA250
HW2

Huajun Chai
998584845
hjchai@ucdavis.edu

Parallel computing

1. Methods used

In this assignment, we will explore parallel computing. In class, we learnt many techniques for parallel computing. They have different characteristics and implementations. According to the demand of this assignment, I used two different methods: Hadoop parallel computing and Java threads parallel computing.

In the following paragraph, I will introduce these two methods in details, and further analysis of the results are provided.

2. Method 1: Hadoop

2.1. Software requirement

Hadoop distributed file system (HDFS): is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly **fault-tolerant** and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject. The project URL is http://hadoop.apache.org/hdfs/ [1].

You can download the latest version of Hadoop from this website: http://hadoop.apache.org/releases.html . The instruction of installation is here: http://hadoop.apache.org/docs/current1/single_node_setup.html .

2.2. Instructions

First, we should load AirlineDelay data into Hadoop file system. But before that, we have to create directories on the hdfs first:

$hadoop fs –mkdir /user

$hadoop fs –mkdir /user/Huajun

$hadoop fs –mkdir /user/Huajun/Data

Then we use the following command to copy data into the directory on hdfs:

$hadoop fs –copyFromLocal ~/Data/Airlines/.csv /user/Huajun/Data/*

After that, we run “make” command in terminal under the Hadoop directory:

$make

This will compile the jar file: eg.jar.

After we get the file eg.jar, we run the following command to do the Hadoop computing:

$hadoop jar eg.jar DelaysFrequencyTable Data Out

But if we want to run Hadoop command in a R script as a source file, we might need the following line of command:

$hadoop dfsadmin -safemode leave

Sometimes Hadoop will lock the node in a safemode. If we use "source" command to run the whole R scripts, it may run into the following error-" name node is in safe mode".

A frequency table is established on the hdfs in the /user/Huajun/Out directory. We can use the following command to copy the frequency table to the local machine:

$hadoop fs –copyToLocal /user/Huajun/Out/ ~/Desktop/STA250/HW2/Out/*

Then we can use the R codes from the first assignment to compute the statistics. I am not going to discuss the details again. Please refer to the first assignment if needed.

2.3. Result

Mean	6.566504
Median	0
Std	31.556326
Total running time(s)	895.559

3. Method 2: Java threads

3.1. Programming language used

In this method, I use java together with R to compute the statistics. A multiple threads technique of Java is used. Java code aims to obtain the frequency table. And R scripts is used to compute statistics based on the frequency table we get from Java code (For this part, we use the R scripts of HW1).

In order to gain the advantage of parallel computing, we use threads technique. We set up multiple threads to deal with the cvs files. Each thread is dedicated to one cvs file. Theoretically, the time of threads running is determined by the longest computation time of the threads. But if there exists the following relation remains to be explored.

3.2. Introduction

In Java, there are multiple ways to implement threads applications. The one that Professor gave us uses “Runnable” functions. There is also one called “callable”. The main difference between these two implementation is that the latter can have return values, while the former has void return.

- **How to synchronize data which is accessed by different threads at the same time?**

During implementation, the biggest issue is that at one time, there may be more than one threads trying to modify some global variables. This can cause some fatal errors towards correct result. For example, thread A and thread B both are trying to update the value stored in Delays[0] at the same time, which has an initial value of 1000. For thread A, what it wants to do is: Delays[0] = Delays[0] + 20; For thread B, what it wants to do is Delays[0] = Delays[0] + 30. However, the program won't know what initial value is for Delays[0] for each thread as they are accessing the same variable at the same time.

One way to solve this problem is to use locker to lock the variable when one thread is trying to access the variable. During the locking time, any other variables have no right of access to that variable. There is a keyword “**synchronized**” which is designed for this use.

However, locking the variable has a large overhead in terms of computation time. Since other threads has no right of access the variable being locked, they have to wait till the current thread has done with its access. This would definitely introduce some amount of extra time. So the efficiency is not that good.

There is way to work around that. We can use “Callable()” function. Callable() allows threads to have a return value. Now we don't need to

have a global variable named Delays, but a variable named delays for every thread. After all threads get their own delays, then we combine them into one “Delays” array. This is quite straightforward. And the implementation of my JavaThreads is based on this callable function.

- **Short introduction to my code**

My java code is modified from the code provided by Professor Duncan. Many thanks to Duncan.

First we feed in filenames. Then we use buf.readline() to skip the header of each csv file. A readRecords() function is used to read the rest data. Function getDelay() tries to pull out the right column of data, for the yearly data it is the 15^{th} column, and 45^{th} for the monthly data. And storeDelay() stored the delay data into a local variable “delays”. And the callable function call() returns this local variable “delays” to the calling function which is the main() function.

In the main() function, a List<Future<int[]>> is used to store delays data for each thread. From the structure of this list we can tell that each element of the list is an int array. “Future” here is some keyword that can allow us to catch the return value of the callable function. We don’t need to know too much about that.

After the list is constructed, we further combine delays data into a variable named “dd” stands for DelayData. In the meanwhile, we write it into a txt file named “delayData.txt”. This txt file will be fed into R scripts to compute statistics.

The codes are all annotated. Please refer to the codes if you have problems understanding this part.

3.3. Result

Mean	6.566436
Median	0
Std	31.553639
Total running time(s)	1306.616

4. Comparison

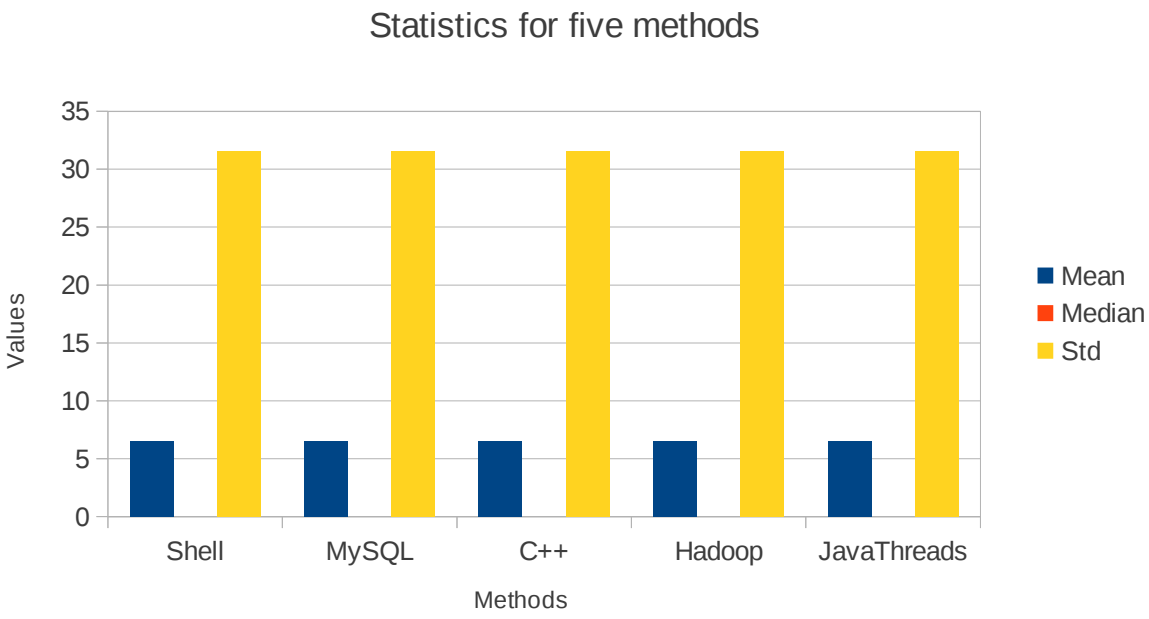
Here we provide a comparison of both statistics calculated and total running time among the two parallel computing methods and the three sequential computing method used in the first assignment.

4.1. Statistics

	Sequential			Parallel	
	Shell+R	MySQL+R	C++&R	Hadoop+R	Java+R
Mean	6.566504	6.566504	6.566504	6.566504	6.566436
Median	0	0	0	0	0
Std	31.556326	31.556326	31.556326	31.556326	31.553639

From the table above we can tell that all the statistics we computed for the five methods are all the same. There is a tiny difference for JavaThreads method. This is because we set a upper and lower bound for the range of the arrival delay, which is -2250 and 2250. There are some value that are out of this range are discarded. So it is not surprising that the mean and standard deviation will be a little different from the rest methods.

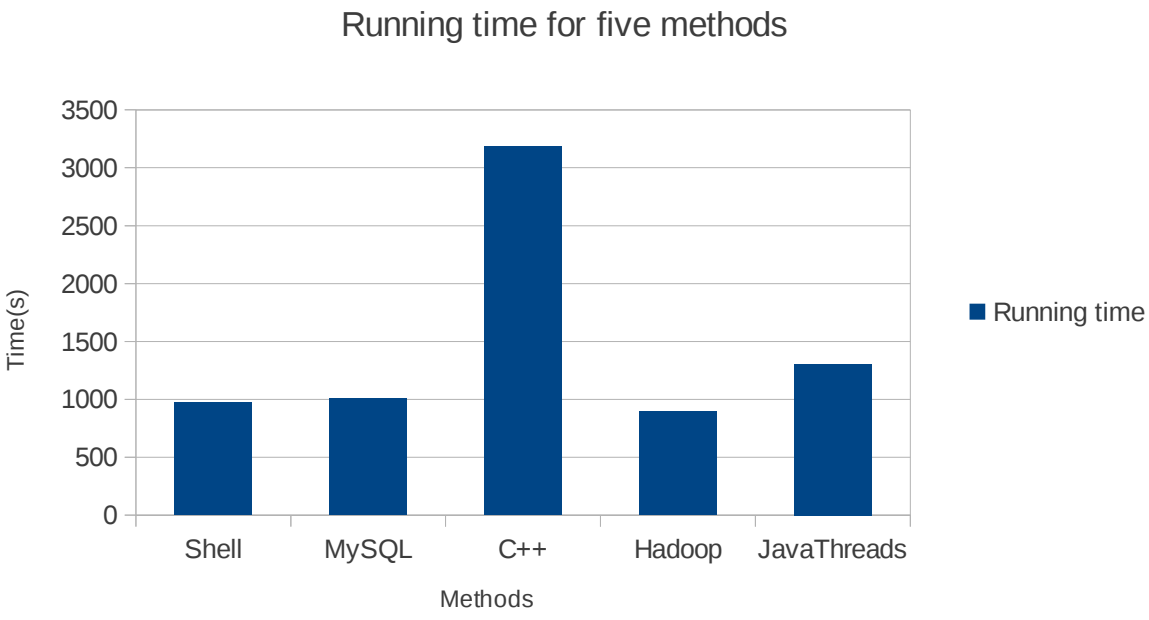
So in this sense, we can say that our codes works fine and the results are correct.



4.2. Running time

	Sequential			Parallel	
	Shell+R	MySQL+R	C++&R	Hadoop+R	Java+R
Running time	971.110	1004.737	3180.432	895.559	1306.616

The running time for these five methods varies a lot. The fastest is hadoop, and the slowest one is C++ looping. Shell and MySQL is quite efficient compared to JavaThreads. Why java threads is not as fast as shell or hadoop or MySQL? I think this is because the limitation of cores of CPUs. Most machines use 4 or 8 cores. But our threads are more than that. There are going to be some jobs waiting while some are being processed. For the evaluation of the effect of # of cores on the computation time, I will further use virtual machine to set a different number of cores to test this.

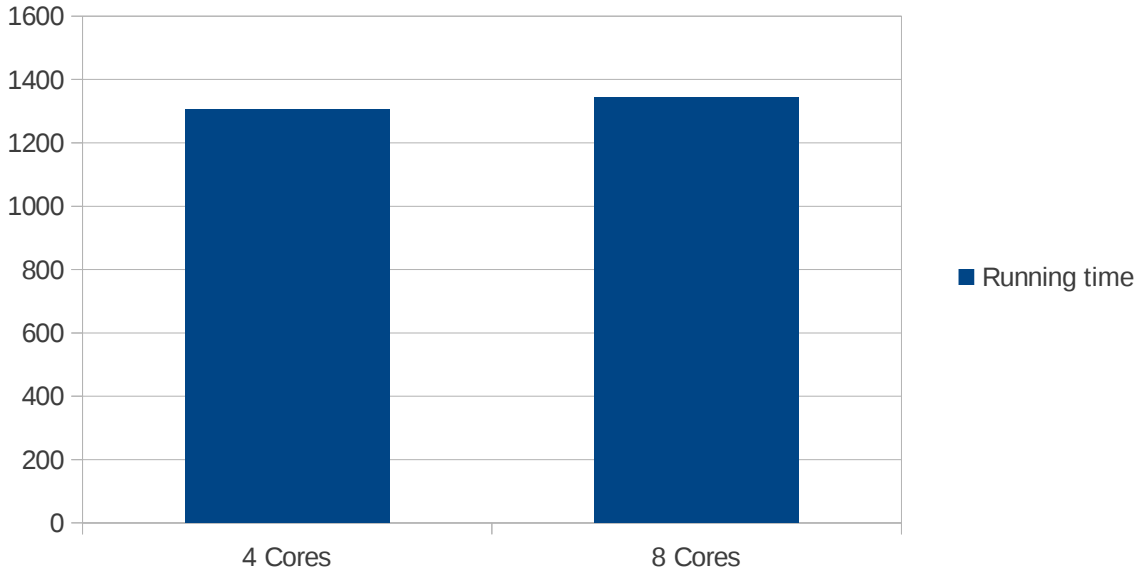


4.3. # of cores vs. running time

I set the # of cores of CPUs from 4 to 8, and run the JavaThreads codes again. The following chart shows the results of two separated runs.

The 8 cores run takes 1344.0s to end. It is very close to 4 cores run, which takes 1306.616s. So the number of cores is not the determining factor of the total running time. And compare to “ecut”, we can tell that javathreads doesn't have huge advantage over merely sequential run. I think there are several explanations for this.

1. The maximum threads that the CPU can process at one time equals to the cores available. If there are more threads than the # of cores, then some of the threads have to wait. Ending old threads and starting new threads can have a great computation cost. Sometimes, this overhead may exceed the benefit from threads running.
2. Maybe the virtual machine is limited by the performance of the physical machine. When we set the cores of the virtual machine to be 8, then the physical machine has little CPU resource to use. This is in turn affect the performance of the virtual machine because virtual machine depends on the physical machine.



5. Conclusion

1. hadoop: hadoop is a good parallel computing technique. The only thing we need to care is the robustness of the program. It has some advantages over JavaThreads.
2. The relation between parallel and sequential computing is not simply linear. There are some complicated relations between them. This may due to the computation limitation, code efficiency and so on.

   So the relation equation in the previous is not valid. There is not such simple relation.
3. All the statistics are correct. Mean is 6.57, median is 0 and standard deviation is 31.56.

Reference:

[1] http://hadoop.apache.org/docs/stable1/hdfs_design.html