Amer Sulieman
10/18/2018
CSCI 4401
HW1 Report

Running on : Macbook pro with a 3.1 GHz Intel Core i5 processor.

Part (A):

Sequential Run:

I compute the matrices multiplication as they were placed in 2D arrays. I used 3 nested for loops to compute the multiplication and storing the result in a final 2D array. The following algorithm will be ran if the user chooses 1 as number of the threads, which means they want it to run sequential.

The algorithm goes as follow:

variables columnSpot and rowSpot to be on:
loop matrix1Row=0 as long as the number of rows haven't been met
    start columnSpot at 0
    loop matrix2Column=0 as long as the numbers of columns haven't been met
        variable result=0
        loop iterator variable at 0 while iterator < matrix 1 columns number
            result +=matrix1[matrix1Row][iterator] * matrix2[iterator][matrix2column]
        resultArray[matrix1row][columnSpot]=result;
        columnSpot++;

Parallel Run:

This case the whole algorithm is different because there will be job distribution for each thread and edge cars such that each thread might not get equal number of jobs to do. When the user inputs any number higher than 1 and less than the number of jobs that the file contain, then the parallel algorithm runs.

To solve parallel multiplication, series of steps are taken:
I create 2 queues for. 1 queue that will contain the row spot for that thread to work on and the second queue contains the column spot correlated to the row that the queue will work on. I use a thread array that will create number of threads given when the program is ran. To calculate the number of jobs each thread should do, I use a separate method that will return an array of number of jobs corresponding to the index which equals the thread number.

DistributeJobs() method takes in three parameters: # of jobs, #threads, boolean value if it is fair distribution or not. As an example: 15 jobs / 7 threads that is not a fair distribution.

DistributeJobs(jobs,threads,fairJobs){

First:
method checks if the booleans value "fairJobs" is true, if it is then it just divides the number of jobs by number threads and place them in array corresponding to threads numbers and return it.

If the jobs are not fairly distributed then:
Loop for each thread in the array of jobs, the idea of the algorithm is as follow, I divide the jobs by the number of threads, if it is capable of being rounded up then I round up the load number and assign the to be the number of jobs to that thread, when the next thread job is calculated, if the previous one was rounded up, then I subtract the amount rounded up for the previous thread from the number of load that the current thread is calculated at, if it is eligible to be

rounded up as well then the same happen as the first thread. If in case a thread load distribution was not able to be rounded up then I round down and keep track how much was rounded down so the next thread gets that amount rounded down from the previous thread.

}

Example:
15 jobs , 7 threads.

First thread load = 15/7 = 2.333333
I can not round that up so I round down and save the .333333, so thread one is 2 jobs loads.
Next thread load = 15/7 = 2.333333
I add the .333333 from the previous round down the current thread, which it will make its load equal to 2.666666 then that is able to be rounded up, so this thread job load is 3. I also save how much I rounded up by which is .33333 something.
Next thread load = 15/7 = 2.333333
which is not able to be rounded up so I subtract the value that I rounded up by earlier which is .33333 now this thread job is 2.33333-.33333=2.
I do this process for all the threads wants to allocate. This way, not a single thread is being overworked so all get fair distribution algorithm.

Since now I have an array with the threads knowing how much work each needs to do, now it is time to to start the multiplication work. Since some threads have more jobs than others, I create a queue for the rows that the thread needs to be on and a queue for the columns that the thread needs to be on.

Adding positions to the queue:
I loop each thread and create those two queues. For every thread, I loop the following as long as I haven't exceeded the number of jobs it has to do.

Column = 0, row = 0;
LOOP:
If the column = number of columns in matrix 2, it means we reached end of a row, so I increment the row variable and reset the column variable to 0. Because every time I reach end of row, it means I need to start on new row. If I didn't reach end of the row then I enqueue that column to the column queue enqueue that row to the row queue. Then increment the column variable and loop again as long as there are more jobs to do.
After I have the queue for the rows and columns that thread needs to do then I create the thread and pass in the id, the row-queue , column-queue, matrix1, matrix2 and the result matrix array to store data. Then I start the thread work.
Thread work goes as follow:
loop:
As long the queues for rows and columns are not empty, that means there are more jobs to do for that thread so keep on looping, so I dequeue [row][column] to work on, I will have them in variables
multiplication result variable =0;
loop:
start iterator at 0 and loop until number of elements to multiply is met.
multiplication result+=matrix1[row][iterator] * matrix2[iterator][column]
resultArray[row][column]=multiplication result;
After all the threads have been looped, I use a barrier so they all can meet and then I print the result.

When my algorithms are ran, they can produce output similar to the following, it may vary for the parallel part because each time threads can finish at different timing.

Sequential Run:

```
                Matrix1
 3.450000, -5.120000, 2.056000, -19.100000
 1.230000, 0.900000, 6.200000, 10.000000
 9.570000, 9.010000, -2.056000, 2.390000
                Matrix2
 5.430000, -12.500000, 65.020000, -1.100000, 6.200000
 3.210000, -9.020000, 4.560000, -11.220000, 4.010000
 5.820000, 1.080000, -2.056000, -23.450000, -13.720000
 83.470000, 56.100000, 4.283200, 42.200000, -2.340000

The job is ran sequentially because you wanted 1 thread!!!!

        Processing in sequential........
Multiplying for result array cell[0][0]..
Multiplying for result array cell[0][1]..
Multiplying for result array cell[0][2]..
Multiplying for result array cell[0][3]..
Multiplying for result array cell[0][4]..
Multiplying for result array cell[1][0]..
Multiplying for result array cell[1][1]..
Multiplying for result array cell[1][2]..
Multiplying for result array cell[1][3]..
Multiplying for result array cell[1][4]..
Multiplying for result array cell[2][0]..
Multiplying for result array cell[2][1]..
Multiplying for result array cell[2][2]..
Multiplying for result array cell[2][3]..
Multiplying for result array cell[2][4]..

        Result of multiplying Matrix 1 * Matrix 2
 -1580.012780, -1066.232120, 114.935544, -800.581800, 17.344480
 880.351900, 544.203000, 114.163400, 265.159000, -97.229000
 268.414580, -69.036680, 677.790984, 37.452000, 118.079820
```

Parallel Run using 7 threads:

The following image should be under each other but they are too big.

```
               Matrix1
 3.450000, -5.120000, 2.056000, -19.100000
 1.230000, 0.900000, 6.200000, 10.000000
 9.570000, 9.010000, -2.056000, 2.390000
               Matrix2
 5.430000, -12.500000, 65.020000, -1.100000, 6.200000
 3.210000, -9.020000, 4.560000, -11.220000, 4.010000
 5.820000, 1.080000, -2.056000, -23.450000, -13.720000
 83.470000, 56.100000, 4.283200, 42.200000, -2.340000

There are 7 Threads Allocated!

Processing in parallel.......
Thread: 1 Number of Jobs = 2
Thread: 2 Number of Jobs = 2
Thread: 3 Number of Jobs = 2
Thread: 4 Number of Jobs = 3
Thread: 5 Number of Jobs = 2
Thread: 6 Number of Jobs = 2
Thread: 7 Number of Jobs = 2

I am thread 1 Processing result array[0][0]
I am thread 5 Processing result array[1][4]
 Thread 5 Processing ... Matrix1[1][0] * matrix2[0][4] = 7.626000
 Thread 5 Processing ... Matrix1[1][1] * matrix2[1][4] = 3.609000
 Thread 5 Processing ... Matrix1[1][2] * matrix2[2][4] = -85.064000
 Thread 5 Processing ... Matrix1[1][3] * matrix2[3][4] = -23.400000
I am thread 4 Processing result array[1][1]
I am thread 3 Processing result array[0][4]
 Thread 3 Processing ... Matrix1[0][0] * matrix2[0][4] = 21.390000
 Thread 3 Processing ... Matrix1[0][1] * matrix2[1][4] = -20.531200
I am thread 2 Processing result array[0][2]
 Thread 3 Processing ... Matrix1[0][2] * matrix2[2][4] = -28.208320
 Thread 4 Processing ... Matrix1[1][0] * matrix2[0][1] = -15.375000
 Thread 4 Processing ... Matrix1[1][1] * matrix2[1][1] = -8.118000
I am thread 5 Processing result array[2][0]
 Thread 5 Processing ... Matrix1[2][0] * matrix2[0][0] = 51.965100
I am thread 7 Processing result array[2][3]
 Thread 7 Processing ... Matrix1[2][0] * matrix2[0][3] = -10.527000
 Thread 1 Processing ... Matrix1[0][0] * matrix2[0][0] = 18.733500
 Thread 1 Processing ... Matrix1[0][1] * matrix2[1][0] = -16.435200
I am thread 6 Processing result array[2][1]
 Thread 6 Processing ... Matrix1[2][0] * matrix2[0][1] = -119.625000
 Thread 6 Processing ... Matrix1[2][1] * matrix2[1][1] = -81.270200
 Thread 1 Processing ... Matrix1[0][2] * matrix2[2][0] = 11.965920
 Thread 7 Processing ... Matrix1[2][1] * matrix2[1][3] = -101.092200
 Thread 7 Processing ... Matrix1[2][2] * matrix2[2][3] = 48.213200
 Thread 7 Processing ... Matrix1[2][3] * matrix2[3][3] = 100.858000
 Thread 5 Processing ... Matrix1[2][1] * matrix2[1][0] = 28.922100
 Thread 5 Processing ... Matrix1[2][2] * matrix2[2][0] = -11.965920
 Thread 5 Processing ... Matrix1[2][3] * matrix2[3][0] = 199.493300
 Thread 4 Processing ... Matrix1[1][2] * matrix2[2][1] = 6.696000
 Thread 4 Processing ... Matrix1[1][3] * matrix2[3][1] = 561.000000
I am thread 4 Processing result array[1][2]
 Thread 4 Processing ... Matrix1[1][0] * matrix2[0][2] = 79.974600
 Thread 4 Processing ... Matrix1[1][1] * matrix2[1][2] = 4.104000
 Thread 3 Processing ... Matrix1[0][3] * matrix2[3][4] = 44.694000
 Thread 2 Processing ... Matrix1[0][0] * matrix2[0][2] = 224.319000
I am thread 3 Processing result array[1][0]
 Thread 4 Processing ... Matrix1[1][2] * matrix2[2][2] = -12.747200
 Thread 4 Processing ... Matrix1[1][3] * matrix2[3][2] = 42.832000
I am thread 4 Processing result array[1][3]
 Thread 4 Processing ... Matrix1[1][0] * matrix2[0][3] = -1.353000
                Thread# 5 is done!!!

I am thread 7 Processing result array[2][4]
 Thread 7 Processing ... Matrix1[2][0] * matrix2[0][4] = 59.334000
 Thread 7 Processing ... Matrix1[2][1] * matrix2[1][4] = 36.130100
 Thread 7 Processing ... Matrix1[2][2] * matrix2[2][4] = 28.208320
 Thread 1 Processing ... Matrix1[0][3] * matrix2[3][0] = -1594.277000
 Thread 6 Processing ... Matrix1[2][2] * matrix2[2][1] = -2.220480
I am thread 1 Processing result array[0][1]
 Thread 1 Processing ... Matrix1[0][0] * matrix2[0][1] = -43.125000
 Thread 7 Processing ... Matrix1[2][3] * matrix2[3][4] = -5.592600
 Thread 4 Processing ... Matrix1[1][1] * matrix2[1][3] = -10.098000
 Thread 4 Processing ... Matrix1[1][2] * matrix2[2][3] = -145.390000
 Thread 4 Processing ... Matrix1[1][3] * matrix2[3][3] = 422.000000

                Thread# 4 is done!!!

 Thread 3 Processing ... Matrix1[1][0] * matrix2[0][0] = 6.678900
 Thread 2 Processing ... Matrix1[0][1] * matrix2[1][2] = -23.347200
 Thread 2 Processing ... Matrix1[0][2] * matrix2[2][2] = -4.227136
 Thread 2 Processing ... Matrix1[0][3] * matrix2[3][2] = -81.809120
 Thread 3 Processing ... Matrix1[1][1] * matrix2[1][0] = 2.889000
 Thread 3 Processing ... Matrix1[1][2] * matrix2[2][0] = 36.084000

                Thread# 7 is done!!!

 Thread 1 Processing ... Matrix1[0][1] * matrix2[1][1] = 46.182400
 Thread 6 Processing ... Matrix1[2][3] * matrix2[3][1] = 134.079000
 Thread 1 Processing ... Matrix1[0][2] * matrix2[2][1] = 2.220480
 Thread 3 Processing ... Matrix1[1][3] * matrix2[3][0] = 834.700000
I am thread 2 Processing result array[0][3]

                Thread# 3 is done!!!

 Thread 1 Processing ... Matrix1[0][3] * matrix2[3][1] = -1071.510000
I am thread 6 Processing result array[2][2]

                Thread# 1 is done!!!

 Thread 2 Processing ... Matrix1[0][0] * matrix2[0][3] = -3.795000
 Thread 6 Processing ... Matrix1[2][0] * matrix2[0][2] = 622.241400
 Thread 2 Processing ... Matrix1[0][1] * matrix2[1][3] = 57.446400
 Thread 6 Processing ... Matrix1[2][1] * matrix2[1][2] = 41.085600
 Thread 2 Processing ... Matrix1[0][2] * matrix2[2][3] = -48.213200
 Thread 6 Processing ... Matrix1[2][2] * matrix2[2][2] = 4.227136
 Thread 2 Processing ... Matrix1[0][3] * matrix2[3][3] = -806.020000
 Thread 6 Processing ... Matrix1[2][3] * matrix2[3][2] = 10.236848

                Thread# 2 is done!!!


                Thread# 6 is done!!!

Everyone meets here after being done!!!

        Result of multiplying Matrix 1 * Matrix 2
 -1580.012780, -1066.232120, 114.935544, -800.581800, 17.344480
 880.351900, 544.203000, 114.163400, 265.159000, -97.229000
 268.414580, -69.036680, 677.790984, 37.452000, 118.079820
```

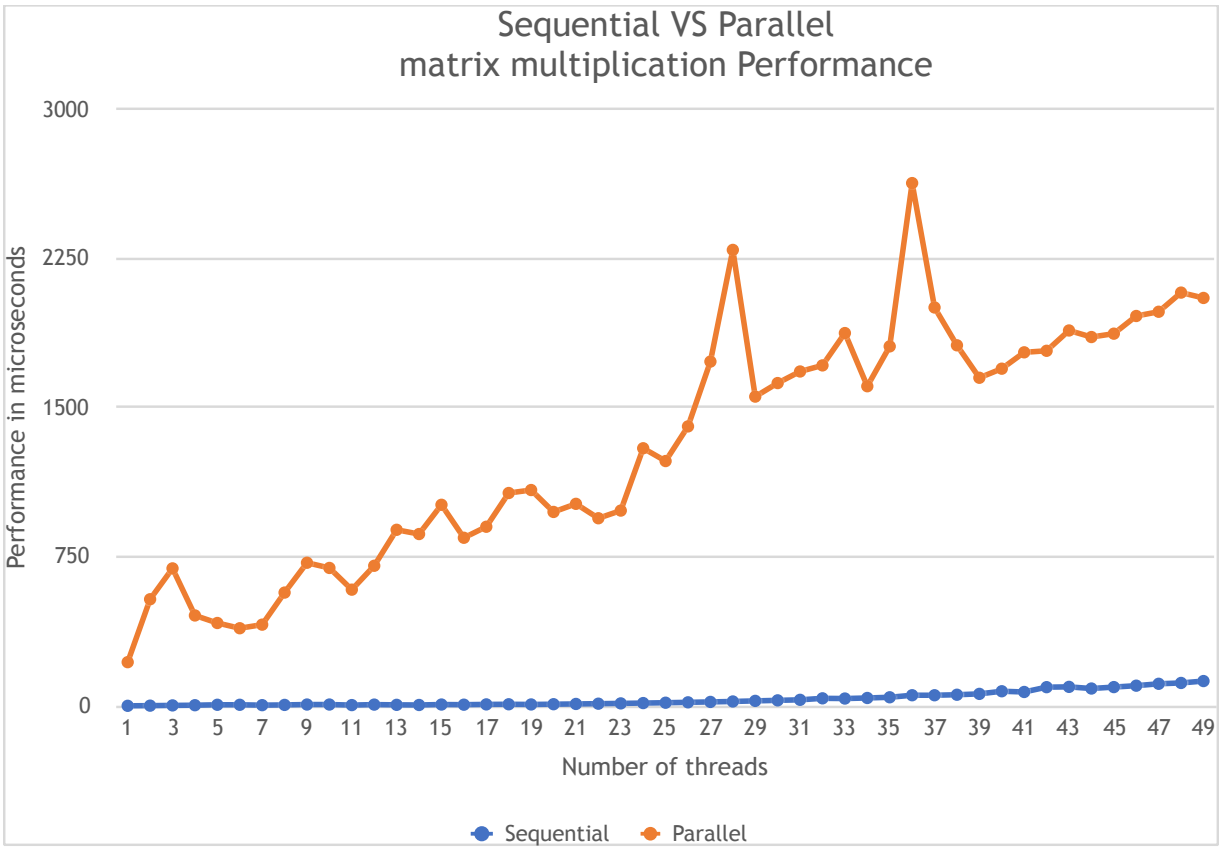
As you can see in the parallel some threads that start first can finish after threads that started later. Each thread has its own world when computing and thats why so the context switching between them that make them look parallel is the reason this happens because they are in user - level threads.

Part(b):

This part will have identical code from the previous part for solving the matrices in sequential and parallel. The main class here will be ran with an argument of 1 or 2 which will represent -> 1 fro sequential test and 2 for the parallel test.
When the user chooses 1 then the method soleSequential is called. This method takes in parameter (int size) which represent the n=2 to n=50 matrices that will be populated.
The main point here for each iteration, each time I create a new matrix of size n*n, I call a separate method to repopulate the matrix with new random data. The method that populates the data chooses random numbers for the integer part and random declaims for the decimals part. I attach them together and make it a double. After the two matrices haven been populated separately with total random numbers, I run the same method in part (a) to solve the matrices sequential. I run the above algorithm 100 times. Each time it is ran a newly random matrices are populated for the same size n*n. I start recording the time in nano seconds after the matrices have been populated and stop it after every iteration and get the difference. The time I get, I add it to a separate variable that later I will use to divide by the number of iterations which will give me the average run for the sequential solution.

To test the parallel run which is threads, I use the same algorithm I used in part A to solve matrices with threads. In here I record the time before each thread calculates its queues on which jobs it is suppose to do. Then stop the time after the barrier was reached because I want to calculate the time for all the threads solving the matrix multiplication. I run this process 100 times as I mentioned previously fro sequential, each 100 iteration a new n*n size matrix is created and also iterated 100 times to calculate the speed of the run.

For sequential and parallel, I copy the performance of each n*n size matrix to an array that will match the n size so later I can print them in a csv file format and graph.



The graph shows that the speed of threads much slower than the sequential run. This was expected because the threads are user-level threads so they are in one process and they are given the same equal share of switching. The amount of how slow the parallel run is crazy but I am sure it is because of my algorithm using queues to know what each thread should do.