***All source code must be compiled with JRE1.8***

***Question 1***

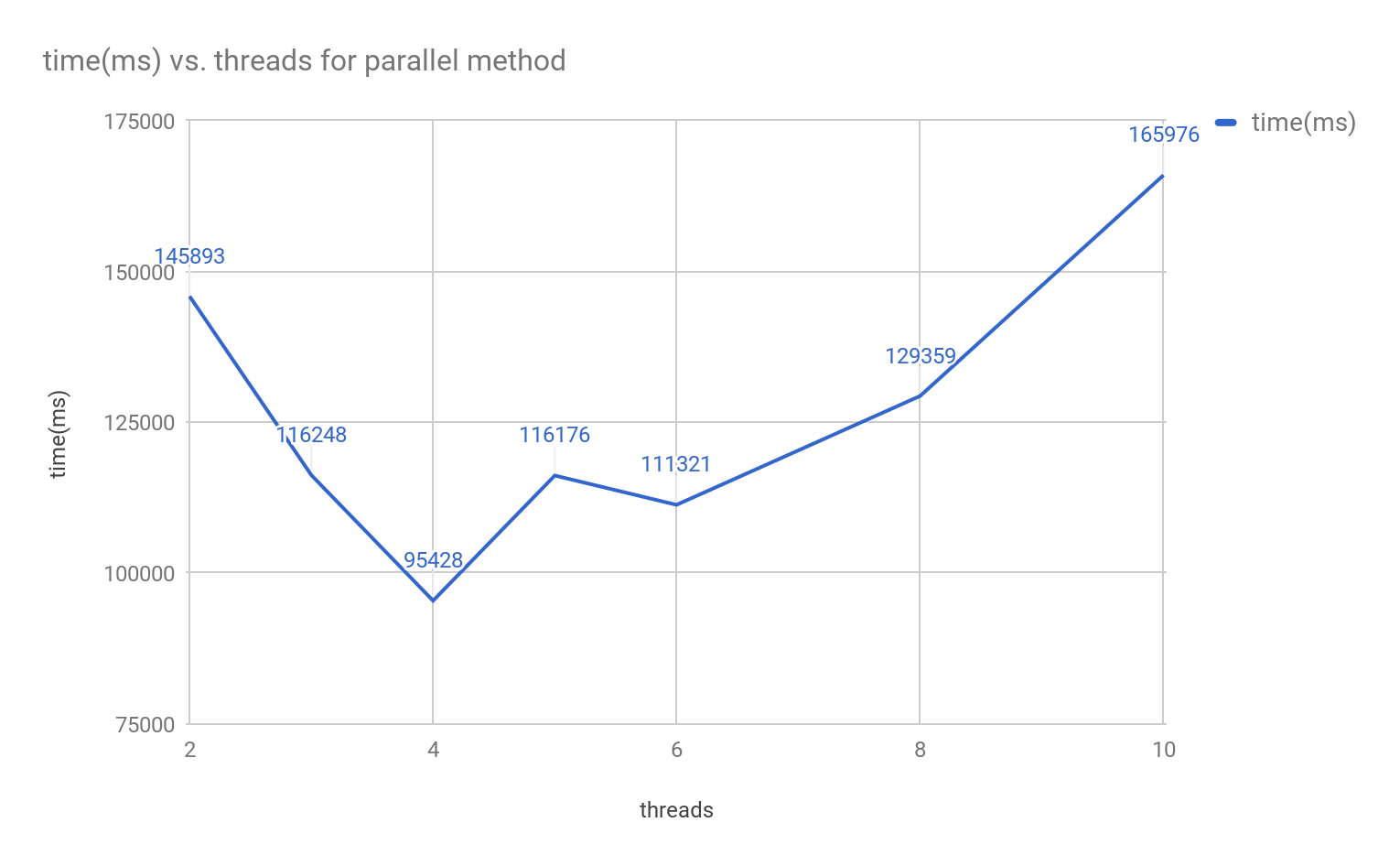
Matrix.java is the source code for this question.

1.1 public static double[][] sequentialMultiplyMatrix(double[][] a, double[][] b) is the method for sequential computation of matrix multiplication.

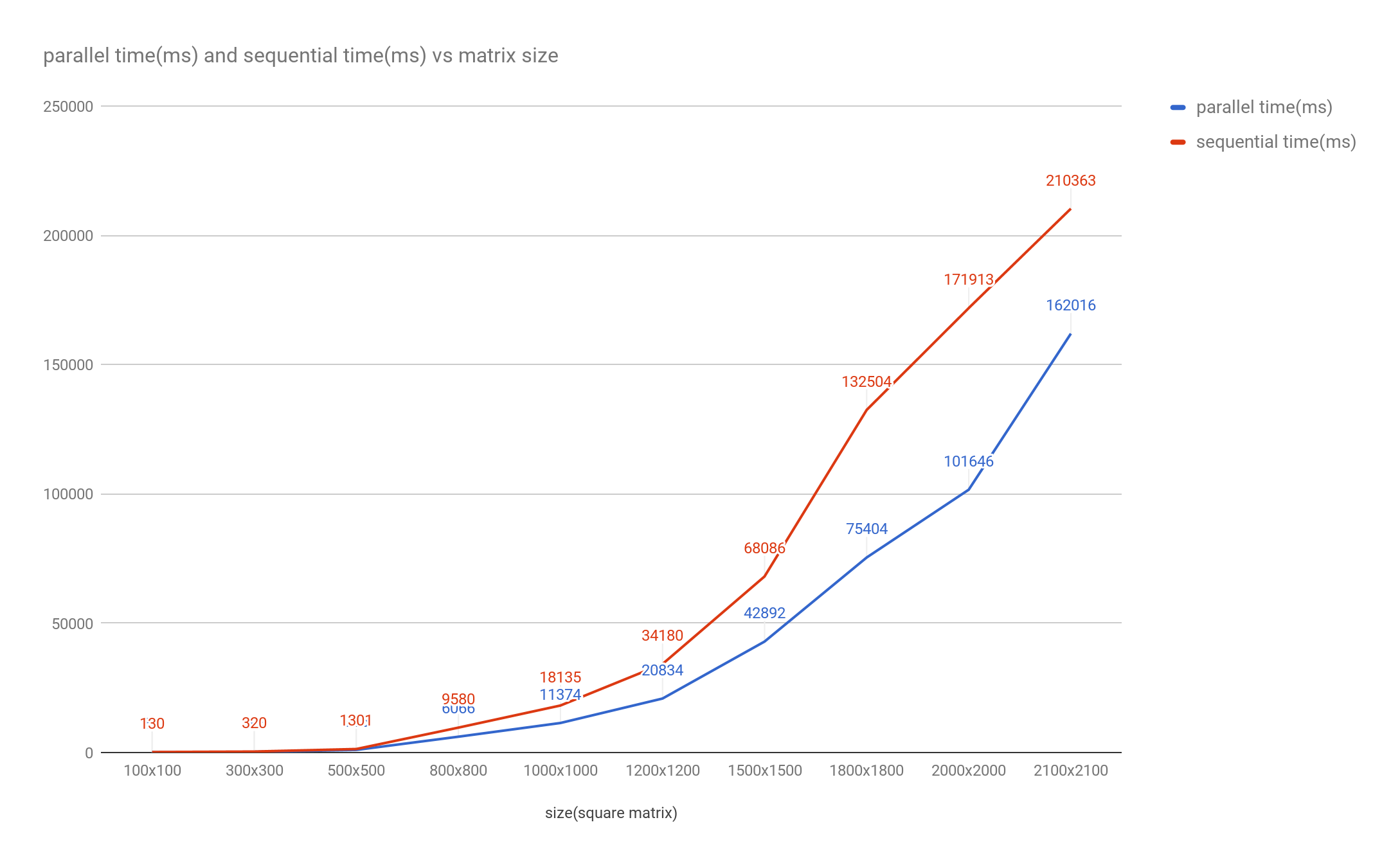
1.2 public static double[][] parallelMultiplyMatrix(double[][] a, double[][] b) is the method for parallel computation of matrix multiplication.

1.3 run the code in main to see the 2000x2000 multiplication running time for each method. We use 2 threads for parallel method. Runtime for sequential is 182035ms and for parallel is 142285ms

1.4 we vary the variable ‘public static final int THREADS’, to get runtime for parallel method for 2000x2000 square matrix multiplication under different number of threads. The result is as following:



1.5 we vary the variable int size to get different size square matrix. Also use 4 threads for parallel method. The result is as following:



1.6 we used 4 cores machine with 4GB RAM for running all the experiment. For graph in 1.4, when increasing the threads from 2 to 4, the runtime decreases, which means with more threads running for parallel method, the multithreading is more efficiency. But when the threads are more than 4, parallel method doesn’t have better performance with more threads. CPU resource and RAM resource both affect the performance. When the threads number exceeds the physical CPU cores, more threads cause overhead and fight for CPU resources. For 1.5 graph, we observe when the matrix size is small (less than 500x500), parallel and sequential method don’t have huge performance difference. But with the increasing of matrix size, parallel program has a better performance, and this advantage increases. This is because with multiple core machine, multi-threading allows for each multiple intensive thread to run simultaneously, which makes full use of CPU resources.

***Question 2***

Deadlock.java demonstrates the deadlock problem. Run the code in main see the result from console window.

2.1 Consequences of deadlock

Deadlock will stop the program. When a deadlock occurs, processes in deadlock cannot get the new resources they need, so the program cannot run following code anymore. Also, since the processes in deadlock will not release the resources they already have, it will decrease the efficiency of using resources. If the deadlock occurs in the kernel program, it will crash the operating system.

2.2 Possible design solutions to avoid deadlock

· To prevent hold-and-wait condition, requiring that a process requests all its required resources at one time, and blocking the process until all requests can be granted simultaneously.

· To prevent no-preemption condition, if a process that holding certain resources is denied a further request, that process must release its unused resources and request them again.

· Using safe and unsafe states, let system evaluate current request and deny it if granting it will lead to potential deadlock.

**Question 3**

3.1DineQ1.java is for this question. Run the program from main. For each philosopher, we lock their left chopstick first and then right chopstick. Deadlock occurs when all philosophers pick up their left chopstick but can’t acquire his right chopstick, because his neighbor has already acquired it.

3.2 DineQ2.java is for this question. Run the program from main. There are four conditions cause deadlock: mutual exclusion, hold and wait, no preemption, and resource waiting. Breaking one of the conditions will solve the deadlock. We find some ways to avoid deadlock:

* Give alternate order of chopstick.
* Don't allow all philosophers to sit and eat/think at once.
* Pick up both chopsticks in a critical section

In our solution, we reverse the left and right chopstick of fifth philosopher, therefore not everyone picks up left chopstick first.

3.3 The code is same as DineQ2.java. In each chopstick object, we use Java Semaphore to lock the object and it has a fairness variable, which will guarantee first-in first-out granting of permits under contention.

3.4 DineQ4.java is for n philosophers which is based on DineQ2.java. Variable ‘final static int NUMBER’ represents n philosophers. Run program from the main.

***Question 4***

4.1

From Amdahl’s law

the limit of the overall speedup is when n -> infinity, p/n -> 0

4.2

parallel part p = 0.3

sequential part (1 - p) = 0.7

Sn represents for speedup on n processes:

Given requirement: S'n> 2Sn

2\*0.3n+2\*0.7K<0.7Kn+0.7K

0.6n+0.7K<0.7Kn

K>0.6n/(0.3n-0.7)

4.3

Torginal=Tseq+Tpara

Tnew= Torginal /2=Tseq/3+Tpara

Tseq=1-p

Tpara=p/n

2Tnew= Torginal

2\*(Tseq/3+Tpara)= Tseq+Tpara

Tseq/3 = Tpara

(1-p)/3=p/n

p=n/(3+n)

sequential part 1-p=3/(n+3)