# Assignment 1 ECSE 420

Elvric Trombert
260673394

October 10, 2018

1. These tests where run using MatrixMultiplication.java file in the ca.mcgill.ecse420.a1 package which was given to us at the start of the assignment. The number of threads and matrix size was hard coded each time in the class private variables the code was then compile and run each time to generate the data bellow.
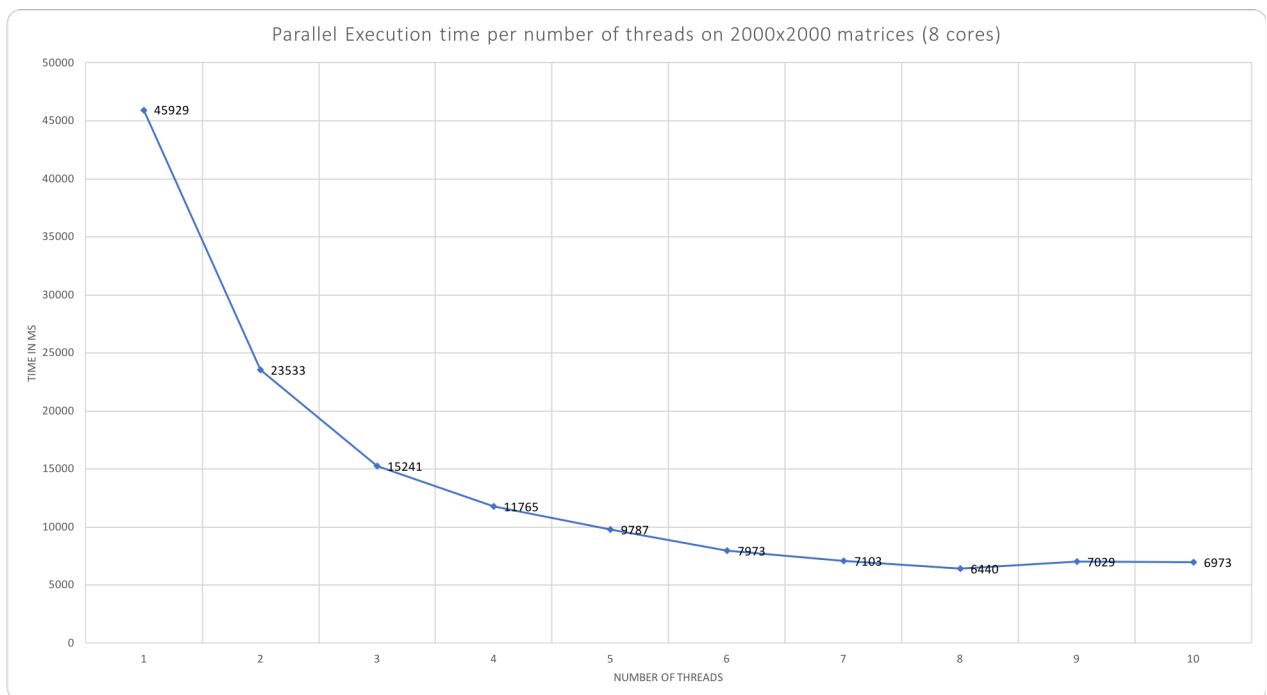
This graph follows an exponential decrease in resolution time as we increase the number of threads. It is important to note that we run this test on a 8 corr CPU machine. The reason for that exponential decrease is that at first the amount of work done by a single thread x becomes x/2 then x/3 and so on. The size of the region that each thread needs to calculate gets reduced by a smaller and smaller ratio every time we add a new thread. This is why the decrease in running time is exponential as the size calculated by each thread decreases by a smaller amount each time.

Since we are running on an 8 core machine it is normal to see a slight increase it running time when the number of threads exceeds the number of core as this adds overhead transitions cost between threads. In our case this was observed for 9 threads however for 10 threads the execution time decreased compare to the one with 9 threads. We think that this may be due to the fact that the execution region of each thread became so small that most thread managed to finish before the hardware algorithm asked for an other thread to get CPU runtime which reduced overhead swtich cost as we do not need to load the interrupted thread back on the CPU again after the swap. In this case we decided
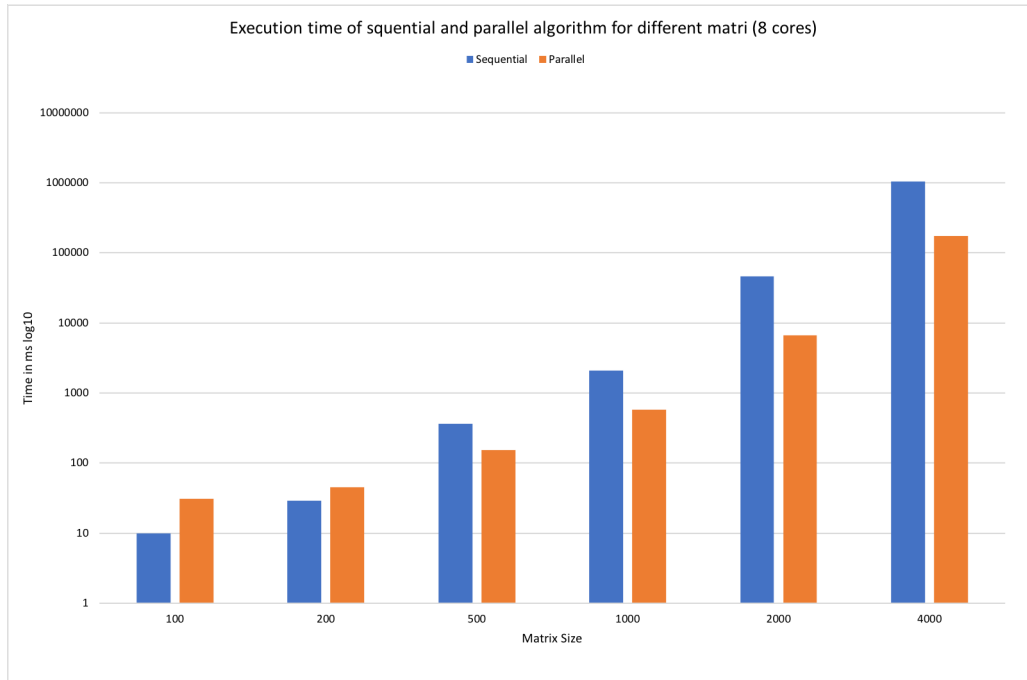
to graph the results on a log scale as the execution time difference between matrix size 100 and 4000 was too large to be visualized on a normal scale. Here the execution time followed a exponential increase for both parallel and sequential. In this case we can see that the execution time of sequential matrix multiplication performed better than the parallel for size 100 and 200 (note that we tried to use 1 to 8 threads for parallel and all times where worse than sequential for all combination at these matrix sizes). We think this is due to the time required to instantiate the executor and the parallelMultiplication class instantiation for parallel that adds greater overheads than just directly starting the sequential execution. However the costs of these overheads has less impact as soon as the matrix size goes over 200 here the parallel execution despite the overheads performs better.

2. The deadlock java file is in the package ca.mcgill.ecse420.a1 named: DeadLocker.java. Here is a sample output of the program:
New attempt...
Thread 1 started
Thread 1 waits to acquire lock 0
Thread 1 acquired lock 0

Thread 2 started
Thread 1 sleeps for 694 milliseconds
Thread 2 sleeps for 986 milliseconds
Thread 1 waits to acquire lock 1
Thread 1 acquired lock 1
Thread 1 unlocked both locks
Thread 1 terminated
Thread 2 waits to acquire lock 1
Thread 2 acquired lock 1
Thread 2 waits to acquire lock 0
Thread 2 acquired lock 0
Thread 2 unlocked both locks
Thread 2 terminated
New attempt...
Thread 1 started
Thread 1 waits to acquire lock 0
Thread 2 started
Thread 1 acquired lock 0
Thread 2 sleeps for 42 milliseconds
Thread 1 sleeps for 919 milliseconds
Thread 2 waits to acquire lock 1
Thread 2 acquired lock 1
Thread 2 waits to acquire lock 0
Thread 1 waits to acquire lock 1

(a) In our case deadlock can occur when the sleep time of Thread 1 is longer than than the sleep time of Thread 2. Since Thread 1 would have acquired lock 0 but will not be able to acquire lock 1. As it would have been acquired by Thread 2 who will be waiting for lock 0 already held by thread 1. Hence we are in a deadlock as neither Thread is willing to give up their resources but cannot finish their execution without the other thread's resources.

(b) Hold-and-wait

Require a process to request all of its required resources at a time making it block otherwise only activating it when all the resources are available. This prohibits resources from being use optimally. Sometimes it is hard to know in advance all the resources that a process will need. The programmer will have to give all the possible resources required by the process note that these are possible not required.

No preemption

If a process holding resources is denied a further request that process must release all its unused resources and request them again.

Circular wait

Can be prevented by defining a linear ordering of resource type. If a process has been allocated resource of type R, then it may subsequently request only those resources of types following R in the ordering. So T1 holds are R1 so it can only request $Ri > 1$, T2 holds R2 so it can only request $Ri > 2$.

4. Amdahl's law $= s = \frac{1}{1-p+\frac{p}{N}}$ Where s is the speed up and N the number of processors/threads.

(a) The sequential part of a program takes 40% on a single processor remains the same on multi processor as it cannot be parallelized. Hence to get the speed up limit we get the following equation

$$s = \frac{1}{0.4 + \frac{0.6}{\infty}}$$
$$s = \frac{1}{0.4}$$
$$= 2.5$$

(b) $a = \frac{1}{k}$

$$2s_n = \frac{2}{0.2 + \frac{0.8}{n}}$$
$$2s_n = \frac{1}{0.2a + \frac{1-0.2a}{n}}$$
$$\therefore \frac{2}{0.2 + \frac{0.8}{n}} = \frac{1}{0.2a + \frac{1-0.2a}{n}}$$
$$0.4a + \frac{2 - 0.4a}{n} = 0.2 + \frac{0.8}{n}$$
$$0.4an + 2 - 0.4a = 0.2n + 0.8$$
$$0.4a(n-1) = 0.2n - 1.2$$
$$\therefore a < \frac{0.2n - 1.2}{0.4(n-1)}$$
$$a < \frac{n-6}{2(n-1)}$$
$$\therefore k > \frac{2(n-1)}{n-6}, n > 6$$

(c) s_p = speed up for $s/3$, s = sequential time

$$s_p = \frac{2}{s + \frac{1-s}{n}}$$

$$s_p = \frac{1}{\frac{s}{3} + \frac{1-\frac{s}{3}}{n}}$$

$$\therefore \frac{2}{s + \frac{1-s}{n}} = \frac{1}{\frac{s}{3} + \frac{1-\frac{s}{3}}{n}}$$

$$\frac{2s}{3} + \frac{2 - \frac{2s}{3}}{n} = s + \frac{1-s}{n}$$

$$\frac{2sn}{3} + 2 - \frac{2s}{3} = sn + 1 - s$$

$$1 = \frac{sn}{3} - \frac{s}{3}$$

$$1 = \frac{s}{3}(n-1)$$

$$s = \frac{3}{n-1}, n > 4$$