**Assignment1**
**Student Name:** Yi-Ting, Hsieh(a1691807)

## Objective:

1. Designing a program which can multiply two very large matrices A and B using threads, with equal numbers of rows and columns, which is given as an input parameter for the program. The values for matrices A and B will be generated randomly. The program will output the sum of all the elements on the diagonal of matrix C, where C = A*B.

2. Analysing the performance of the program on two different machine: one on my own device, and also on the high-performance cluster Phoenix.

## Design:

Let's split out the program to three parts:

First, creating a *public class called Test* where our main function will be placed. Our main function reads first argument and *sets up the matrix size n before it calls the random function to randomly generate two n*n matrices as A and B*. Then, it sets up the value about how many demanded threads that user want to use; however, for our program, it is recommed that the demanded number should be a factor of n, we will talk about why is that later.

Second, we have to calculate our matrix C using desired number of threads, where C=A*B. We could easily achieve this by using the property of inheritance. Storing some static values to our public class Test first, which is our A and B matrix, and creating another class called Multiply which is inheirted from Test class, so that it has access to A, B and C. Since we have already assigned desired number of threads to do matrices multiplication, each thread is suppose to handle at least one row of our C, which means if C is n*n matrix, then *each thread has to calculate at least n elements in the same row to finish it's job*. So how I achieve and make sure that other threads will not calculate the same place twice. The idea is simple, by storing a private value called *SelectBlock* shared in all the threads. So once a thread is called, it will know which row it is assigned for by checking the value of SelectBlock before it increases the SelectBlock by one for next candidate.

However, we do not want to see our main thread finish it's job and exit before other threads. So I *use join() to make sure it(main thread) will wait until all the threads finish theirs' jobs*. This is not an efficient method, but it is easy to design and implement. Even though it may lead to delaying the computation time, as long as our n is large enough, the influence would be less noticable. There is another thing should be awared, that is we only use int type to store our A, B and C matrices. Because n will up to few thousands, the elements in A and B can not be too large. So I set up the *random value for A is from 0 to 10, and B is from 0 to 100*. So the matrice could up to few thousand without overflowing. This could be improved next time by assigning another type to store our integer values, such as BigInteger.

## Analyze:

## Local Machine(Dell E7250):

**Specs:**

i7-5600u dual cores @3.2GHz
(2 physical cores, 4 logical)
16 gb ram ddr-3l 1600mHz
Hard Drive: 512 Gb SSD(mSATA)
Intel HD Graphics 5500
OS: Arch Linux x86_64
Kernel: 4.15.8-1-ARCH

### 1.) Find Knee Value:

The first task is finding our knee value, which is the maximum number of threads beyond which the performance (in terms of runtime) does not improve proportionally or even starts to deterlorate. By utilizing the *time command in Linux terminal*, we could easily get *runtime* of our program. The time command runs the specified program command with the given arguments. When commands finishes, time writes a message to standard error giving timing statistics about this program time. And we only focus on the real time between invocation and termination.

**Command: (time java Test n number_of_threads)**

However, in our program, we initialise random matrices A and B before we call n threads to do matrice multiplication. *Our real time includes the time we initialise the random matrices*, as n goes large, the longer time it will take. More than that, before different threads doing it's multiplication, the program has to call *the system to assign our demanded number of threads, which is also included in the real time*. But we already know the *multiplication is a heavy task* for computer (from ADSA course, especially the Karatsuba Multiplication, we know when the program is doing multiplication, it will heavily use registers to do the calculation). *As the user requests large numbers of multiplication or calling program to do multiplication several time, it would take much longer time to do the task compared to setting up environment values and doing system calls.* So we can assume that as n is large enough, the required time of initialising random matrice process and assigning threads would be negligible.

**Method:**
*Keeping n constant(but a large value), and running the program using incresingly large numbers our thread.*

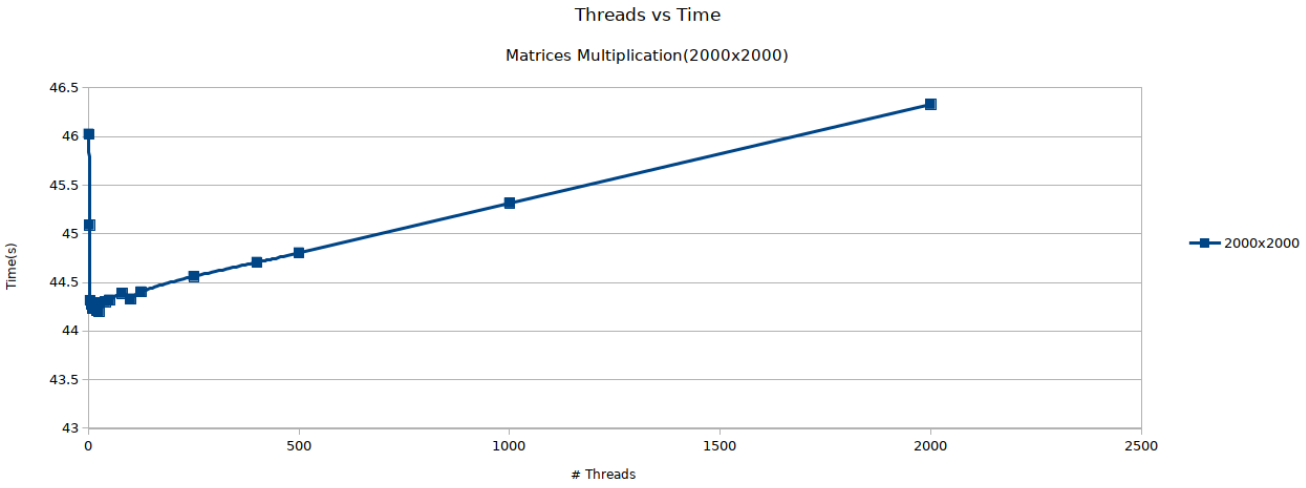
As far as my local machine's spec is concerned, we can *expect that the knee value should be 4*. As mentioned before, this Dell laptop has i7-5600 CPU, which has *2 cores and 4 threads*(there's another interesting fact that if it is i5-5400u, it  have 2 cores with 2 threads. According to my friend, who works as IC RD for a famous company in Taiwan, saying the architecture and the way how they makign i5 and i7 are completely the same. He pointed out that they were all manufacturing in the same manufacturers, the only difference was the production rate. If it's above value, like 70%, then it would be sold as i7 processor. If below that value, it would be limited the number of threads and sold as i5). We know there is a primary constraint for thread, that is it requries hardware support, which means it requires a processor with hardware threads. So far, one processor supports up to two threads executing concurrently. Since the Dell laptop has 2 cores, so it can support up to 4 threads executing at the same time. Above that numbers, it will not accelerate the process. The reason is that there is no spare hardware threads for the machine to execute it. So the kernel will keep looking for free threads, and once another thread finish it's job and restore the resource back to kernel, it will be assigned the task again in our case until we get our C matrix. Because it can not run all the threads simultaneously, some of the

threads have to wait. That is why it will not improve the performance. However, different operating systems and other hardwares, such as memory and hard drive may also improve the exepectd result. The best value might be *number_of_cores * 2 + x*(value that depends on hardware and software) on Linux. But it beyond my knowledge, I am not going to explore the detail here.

The data below is the required time for 2000*2000 matrices multiplication on different number of threads:

| #Threads | 1 | 2 | 4 | 8 | 10 | 16 | 20 | 25 | 40 |
|---|---|---|---|---|---|---|---|---|---|
| Time(s) | 46.0258 | 45.0897 | 44.3155 | 44.2738 | 44.2333 | 44.2934 | 44.2101 | 44.2017 | 44.3 |

| #Threads | 50 | 80 | 100 | 125 | 250 | 400 | 500 | 1000 | 2000 |
|---|---|---|---|---|---|---|---|---|---|
| Time(s) | 44.3195 | 44.3866 | 44.33025 | 44.4036 | 44.5621 | 44.7073 | 44.8047 | 45.3147 | 46.3304 |



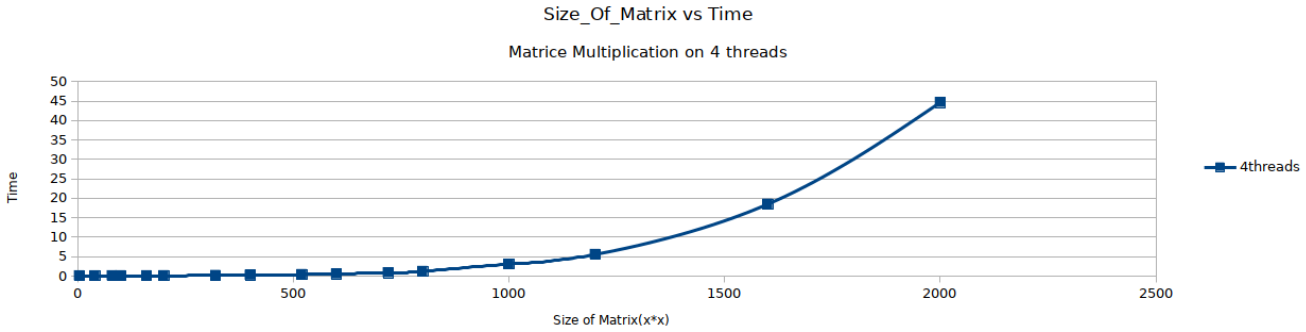

As expected, when there is only one thread, it took around 46 seconds to finish the task. We can see from the above diagram and table that shows the runtime improved significantly when we increase the t to 4.  But it does not have noticeable change since that. More than that, when the demand number of threads is above 500, we can see it slows down the program. *With the deduction above, as well as the data proved that the knee value should be 4.*

**2.) Using the knee value, which is 4, running our program using increasingly large values of n**

Now we know our *knee value is 4*, let us keep the knee value as constant and testing our program with different sizes of our matrices.

| Size of n | 4 | 40 | 80 | 100 | 160 | 200 | 320 | 400 | 520 |
|---|---|---|---|---|---|---|---|---|---|
| Time(s) | 0.0749 | 0.0749 | 0.0.862 | 0.0907 | 0.0901 | 0.0988 | 0.1572 | 0.2238 | 0.3836 |

| Size of n | 600 | 720 | 800 | 1000 | 1200 | 1600 | 2000 |
|---|---|---|---|---|---|---|---|
| Time(s) | 0.5286 | 0.8589 | 1.1996 | 3.0654 | 5.5685 | 18.4732 | 44.6501 |




The diagram and table above shows a resonable fact that under the same condition(same number of threads), when *the size of our matrices become larger, the longer time it will take for our program to finish the matrices multiplication.*

## Phoenix Supercomputer:

**Specs:**

Xeon E5-2698v3 @ 2.3GHz,  32 cores per node
Sockets: 2 per node
Memory: 128 Gb per node

**Environment setting:**

```
#!/bin/bash
#SBATCH -p batch
#SBATCH -N 1
#SBATCH -n 32
#SBATCH --time=01:00:00
#SBATCH --mem=4GB
```

As the information shown above, we request a node with 32 cores and 4 Gb memory space for our program.

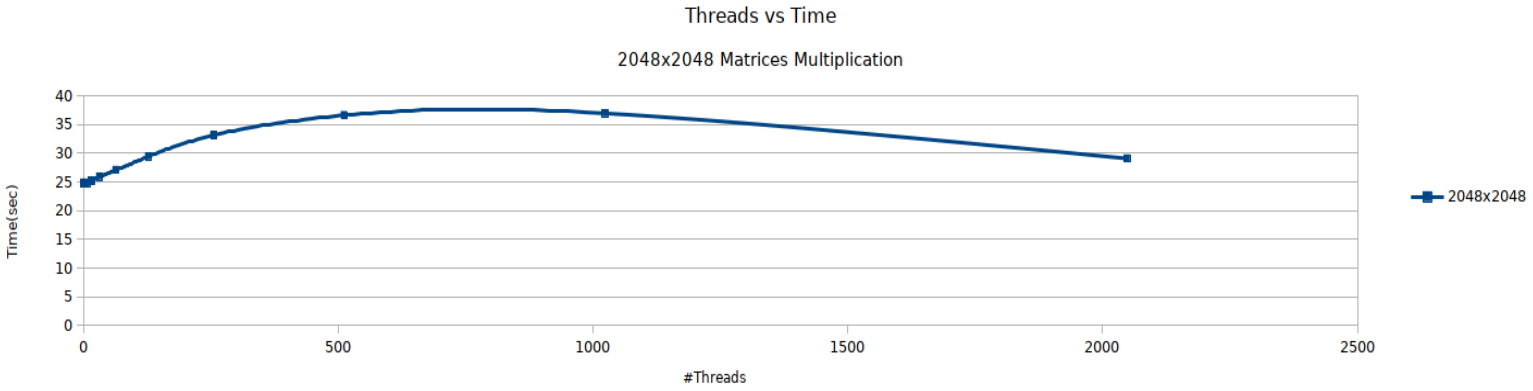
**1.) Find Knee Value:**

Earlier, we just discussed how we used time command to get the program runtime. By the same reason, we could assume that as n is large enough, the required time of initialising random matrice process and assigning threads would be negligible.

The data below is the required runtime time for 2048*2048 matrices multiplication averaged from *30 cases* on different number of threads:

| #Threads | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|---|---|
| Time(s) | 24.843 | 24.878 | 24.791 | 24.816 | 25.189 | 25.928 | 27.076 | 29.456 | 33.162 |

| #Threads | 512 | 1024 | 2048 |
|---|---|---|---|
| Time(s) | 36.635 | 36/948 | 29.086 |




From above diagram and table, we could find out that when t is between 1 and 8, the runtimes do not have much difference from one another. But when t is demanded more than 16, the runtime has significant increased to 25 senconds. So we could deduce that t*he knee value is 8* because it is *the greatest number of threads before it starts to deteriorate the performance.*

However, there was an issue when I was gathering data from Phoenix supercomputer. At first, I wrote two separate job scripts to Phoenix, and they're all requesting 1 node with 32 cores and 4 Gb memory (the partition is batch). The first one was for the knee value which seemed to be 64, and the average runtime for a 2048x2048 matrices multiplication took around 12.20 seconds. Another script was setting the t value to 64 and increased the n. But I noticed that the average runtime took 26.7 seconds when n was equal to 2048. Because the performances were not the same, I submitted more scripts to gather more data for analyzing. And it seemed it has 70-30 chance that most of cases would get the average around 26 seconds, while the rest only took 12 seconds. My explanation to this would be:

(1) The server was doing other heavy tasks at the same time and somehow it got this two different results.

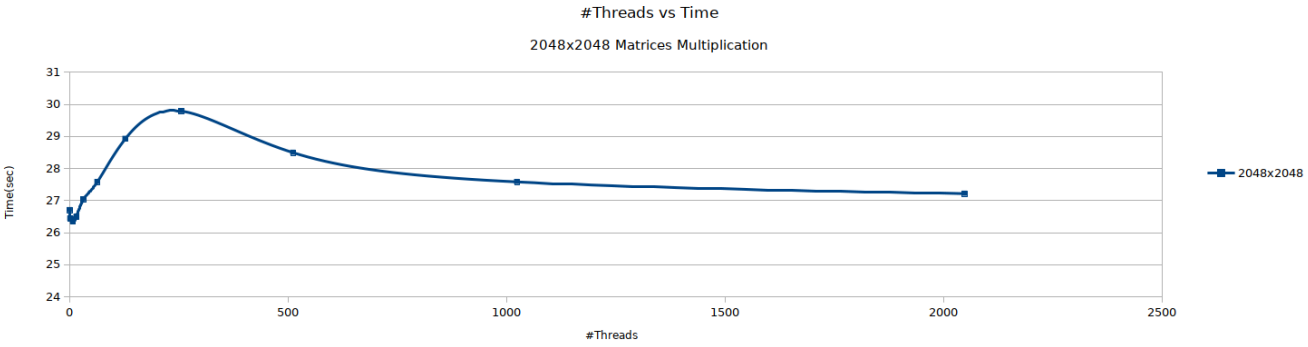
(2) The server distributed my jobs to different hardwares with differentn specs, and that was why it has two distinct results.

From above concerned, I decide to choose the majority cases to analyze. After gathering more than 30 cases, I find out that when the demanded number is 4 or 8 would get the best performance, which is the shortest runtime. *Because the runtime for 4 threads and 8 threads are too close to distinguish, and there are some set of cases that shows the program has best performance under 8 threads.* So I decide to *choose the greatest number of threads before it starts to deteriorate the performance remarkably.*

The data below is the average runtime time for 2048*2048 matrices multiplication from *a set of 10 cases* on different number of threads, which could support my deduction above:

| #Threads | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|---|---|
| Time(s) | 26.7064 | 26.441 | 26.4305 | 26.347 | 26.493 | 27.0415 | 27.5844 | 28.9282 | 29.7935 |

| #Threads | 512 | 1024 | 2048 |
|---|---|---|---|
| Time(s) | 28.4961 | 27.5837 | 27.2172 |

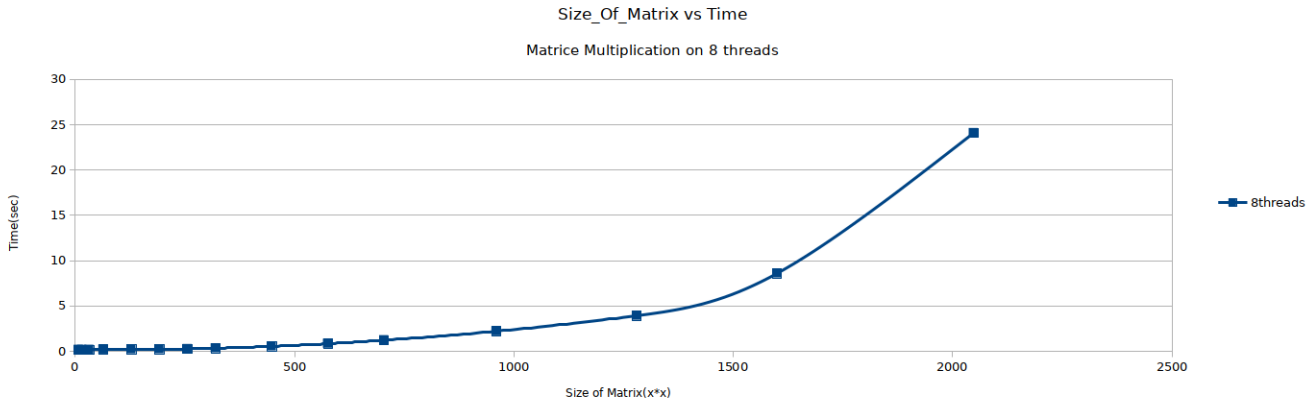


**2.) Using the knee value, which is 8, running our program using increasingly large values of n**

Now we know our *knee value is 8*, let us keep the knee value as constant and testing our program with different sizes of our matrices.

| Size of n | 8 | 16 | 32 | 64 | 128 | 192 | 256 | 320 | 448 |
|---|---|---|---|---|---|---|---|---|---|
| Time(s) | 0.1776 | 0.1704 | 0.1773 | 0.1878 | 0.2173 | 0.2294 | 0.257 | 0.349 | 0.544 |

| Size of n | 576 | 704 | 960 | 1280 | 1600 | 2048 |
|---|---|---|---|---|---|---|
| Time(s) | 0.8375 | 1.2323 | 2.2118 | 3.927 | 8.596 | 24.094 |




The diagram and table above shows a resonable fact that under the same condition(same number of threads), when *the size of our matrices become larger, the longer time it will take for our program to finish the matrices multiplication.*

## Result:

Now, as the diagrams and tables shown above, we could easily find out the fact that *Phoenix Supercomputer has much better performance and faster computational ability compared to normal computer.* For the Dell laptop(e7250), which would take 45 seconds to compute a 2000*2000 matrices multiplication. And the knee value is as expected from the hardware specification. While for the Phoenix Supercomputer, it takes less than 26 seconds to finish a 2048*2048 matrices multiplication task. But as we have already discussed above, there may have two hardwares that performs distinctly different. In either case, the computational ability is still obviously better than the Dell laptop. Another interesting fact is that when we are requesting a 2048*2048 matrice multiplication, if our demanded number of threads are 2048, we could expect that it would take less time to finish the task on Phoenix Supercomputer. Perhaps, we can request more threads to improve the performance and shorten the required time to finish the calculation. But due to our program implementation, which is *each thread has to calculate at least n elements in the same row to finish it's job*, it restricts the potential computational ability for Phoenix Supercomputer to use more threads to handle the task. Therefore, the program design and software implemtnation has significant influence on performance for supercomputer. It shows that it is the software developer or software engineer responsibility to ensure the program efficiency and optimization for parallel programming on supercomputer.