Operating System -2 Programming Assignment 2

Name : Ahmik Virani
Roll Number : ES22BTECH11001

Low Level Design :

From the main function, I am creating the thread onto the runner function in a for loop as shown below

```
    if(N > K)
    {
        for(int i = 0 ; i <  K ; i++)
        {

            threadNumber[i] = i;
            pthread_create(&tid[i], &attr, runner, &threadNumber[i]);
        }

        /* Wait for threads to finish */
        for(int i = 0 ; i < K ; i++)
        {
            pthread_join(tid[i], NULL);
        }

    }

    else
    {

        for(int i = 0 ; i <  N ; i++)
        {

            threadNumber[i] = i;
            pthread_create(&tid[i], &attr, runner, &threadNumber[i]);

        }

        /* Wait for threads to finish */
        for(int i = 0 ; i < N ; i++)
        {
            pthread_join(tid[i], NULL);
        }

    }
```

The if-else statement is only for efficiency reason. Say if N<K, then there is no need to create the extra threads.

Below are the runner functions for mixed and chunk algorithms :

1. *Mixed technique for computing the matrix using thread affinity*

```
void *runner(void *param)
{
    long start = getTime();

    int k = (*(int *)param);

//    int b = K/C;
    int b = 0;

    if(k < BT && b != 0)
    {
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);

        int s;

        CPU_SET(k/b , &cpuset);

        s = pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);

        if(s != 0)
        {
            perror("pthread_getaffinity_np");
        }
    }

    for(int i = k; i < N; i += K)
    {
        multiply(i);
    }

    long end = getTime();

    if(k < BT && b != 0)
        timeBounded[k] = end - start;
    else
        timeUnbounded[k - BT] = end - start;

    pthread_exit(0);
```

The input parameter is the thread number.

I am using two values of b, either b = K/C as per graphs 2 and 4, or I am using b = 0, as for graphs 1 and 3.

Now, if the input parameter, i.e. the thread number, if it falls in the first BT threads, i.e. it must be bounded, then I manually set the thread to the particular CPU. I set it to CPU k/b because, from k = 0 to b – 1, all threads are bound to CPU 0. Since b and k are both off data type int, k/b is also int, so these values from k = 0 to b – 1 for k/b will be 0, similarly the next b threads will pass to CPU 1 and so on.

Next, I will call the multiply function to compute the values of the corresponding rows as per the mixed algorithm definition.

2. *Chunks technique for computing the matrix using thread affinity*

Low Level Design

```
void *runner(void *param)
{
    long start = getTime();

    int k = (*(int *)param);

//    int b = K/C;
    int b = 0;


    if(k < BT && b != 0)
    {
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);

        int s;


        CPU_SET(k/b , &cpuset);

        s = pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);

        if(s != 0)
        {
            perror("pthread_getaffinity_np");
        }
    }

    for(int i = k; i < N; i += K)
    {
        multiply(i);
    }
    long end = getTime();

    if(k < BT && b != 0)
        timeBounded[k] = end - start;
    else
        timeUnbounded[k - BT] = end - start;

    pthread_exit(0);
}
```

The input parameter is the thread number.

I am using two values of b, either b = K/C as per graphs 2 and 4, or I am using b = 0, as for graphs 1 and 3, as mentioned above

Next, I will call the multiply function to compute the values of the corresponding rows as per the chunk algorithm definition.

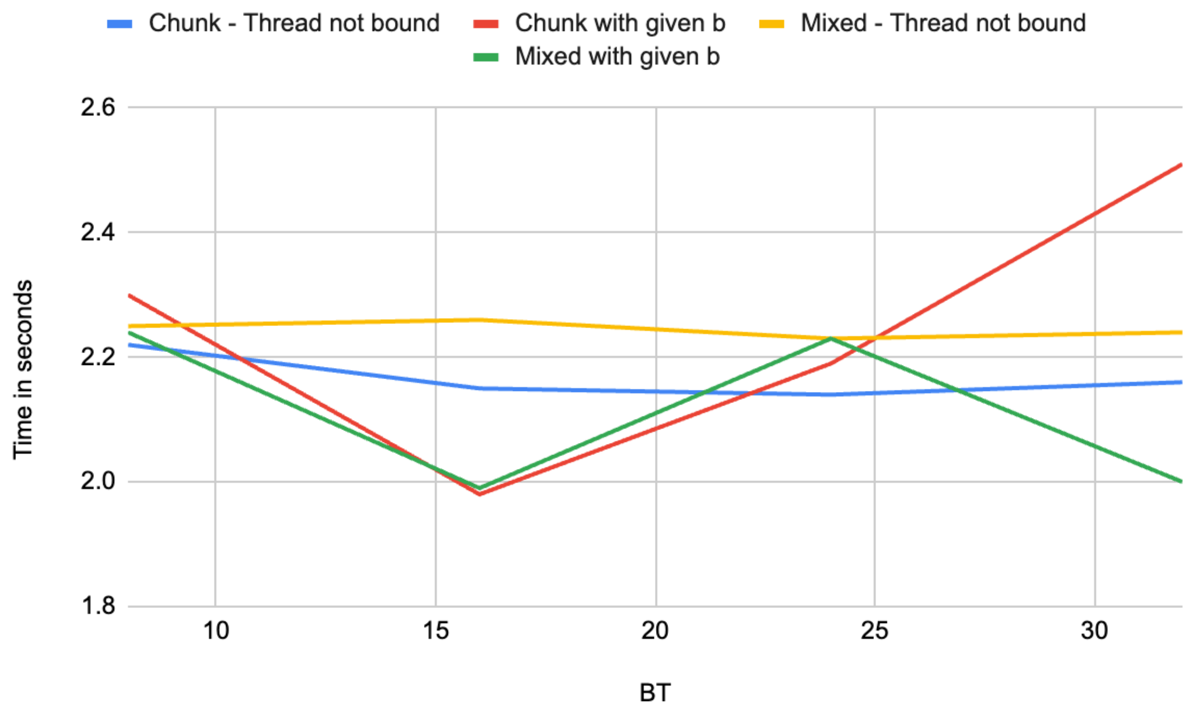
Why these Runner Functions work for experiment 1?

As mentioned, the number of threads per CPU is b, as explained above. The first b threads will be assigned to CPU 0, next b to CPU 1 and so on.

Performance of Algorithm for experiment 1 : Total Time vs Number of Bounded Threads, BT:

BT ->	**8**	**16**	**24**	**32**
Chunk - Thread not bound	2.22	2.15	2.14	2.16
Chunk with given b	2.3	1.98	2.19	2.51
Mixed - Thread not bound	2.25	2.26	2.23	2.24
Mixed with given b	2.24	1.99	2.23	2

All the values in the above table are in seconds



The general trend is that, when threads are not bound, the curves are almost constant, this is trivial because K (the total number of threads) are kept constant, so, with no threads being bound, the overall time must remain constant.

There is no clear pattern as of such that is noticeable for the case when some of the treads are bound and other not bound.

The unpredictability may be due to

> **Resource Allocation and Contention**:
> When we use an algorithm where there is both bounded and unbounded threads, the resource allocation becomes more complex, that is because the bounded threads are permanently associated with specific resources wihle unbounded ones are dynamically allocated their resources. Contentions for shared resoucres such as CPU cores or memory or CPU time can lead to unpredictable execution times

Why these Runner Functions work for experiment 2?

Here we have to assign K/2 threads to C/2 cores equally. Basically if we use the same variable b = (K/2) / (C/2) = K/C, then we are assigning K/2 threads to C/2 cores. For simplicity, take all parameters K, C, BT to be a power of 2 as mentioned in the question, so no discrepancy occurs.

Please note that, for the codes to be uniform in bith experiment 1 and 2, I have not included the % (C/2), in front of k/b.

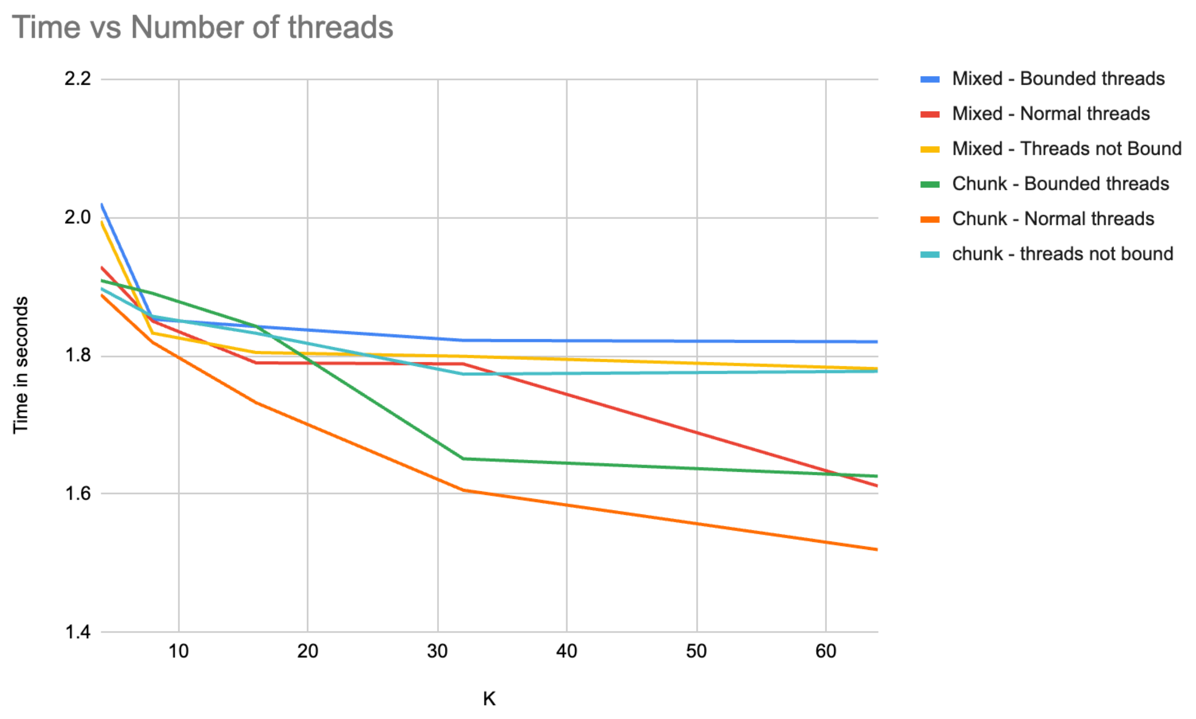
```
	CPU_SET(k/b % (C/2) , &cpuset);
```

Would be a better improvisation, however may not comply with experiment 1.

Performance of Algorithm for experiment 2 :

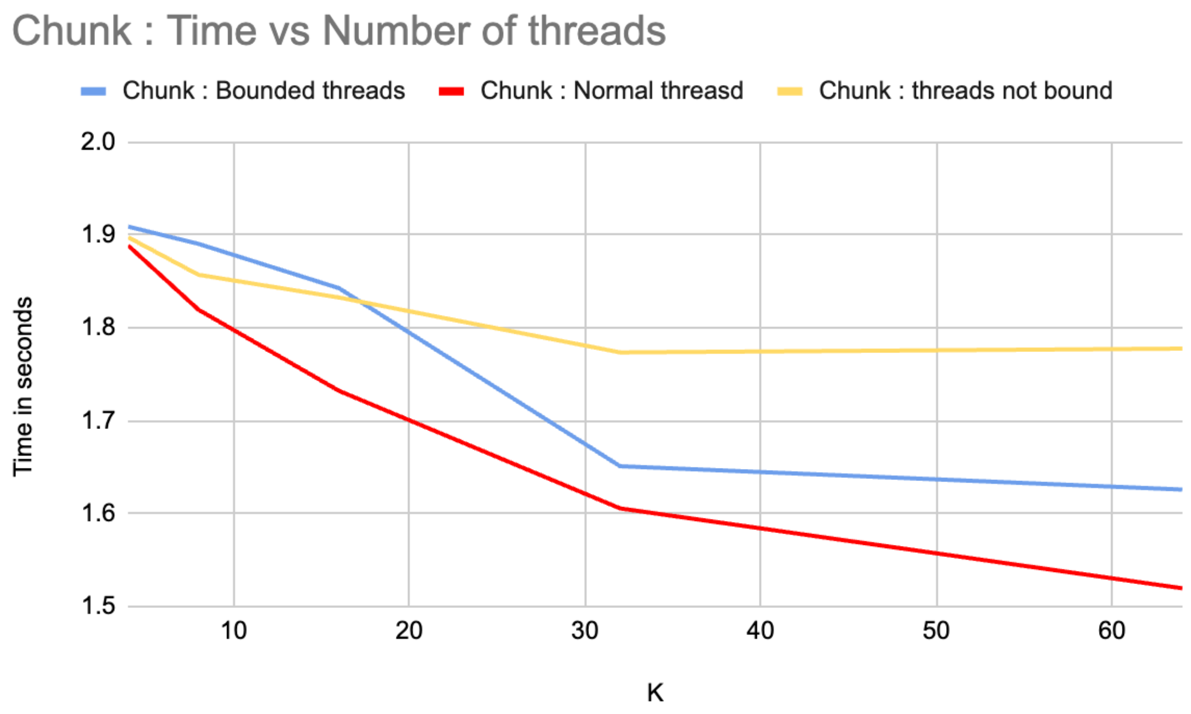
K	Mixed - Bounded Threads	Mixed - Normal Threads	Mixed - Threads not bounded	Chunk - Bounded threads	Chunk - Normal Threads	Chunk - Threads not bounded
4	2.020559	1.928966	1.995037	1.909061	1.888745	1.897621
8	1.853049	1.850343	1.832974	1.890494	1.81963	1.857182
16	1.84242	1.789882	1.804811	1.842759	1.732246	1.8327
32	1.822395	1.788331	1.799388	1.651076	1.605643	1.773585
64	1.820432	1.611749	1.781448	1.625903	1.519622	1.777655

All the values in the above table is in seconds

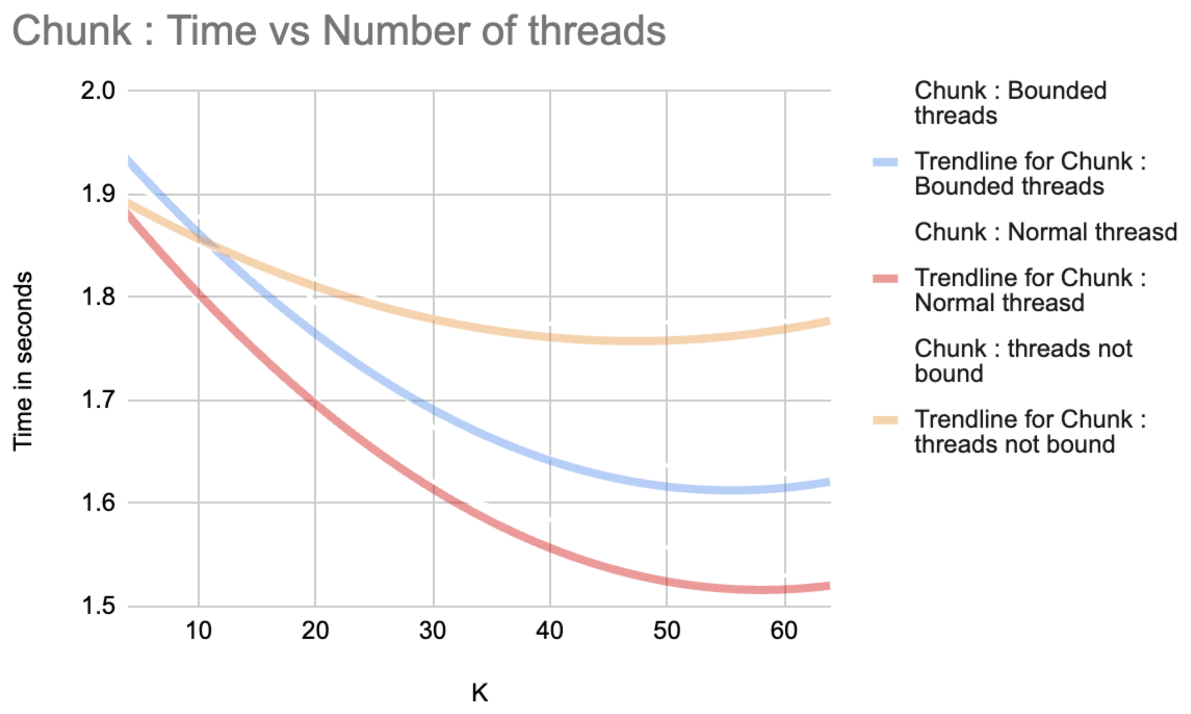


As this graph is very messy, let us analyze the mixed and chunk graphs separately and then come back to this.

First Chunk:



The trendline is shown below



We can see that the graphs are exponentially decreasing, then attain a somewhat constant level.

The reason that the time per thread on average is decreasing as number of threads increases is because

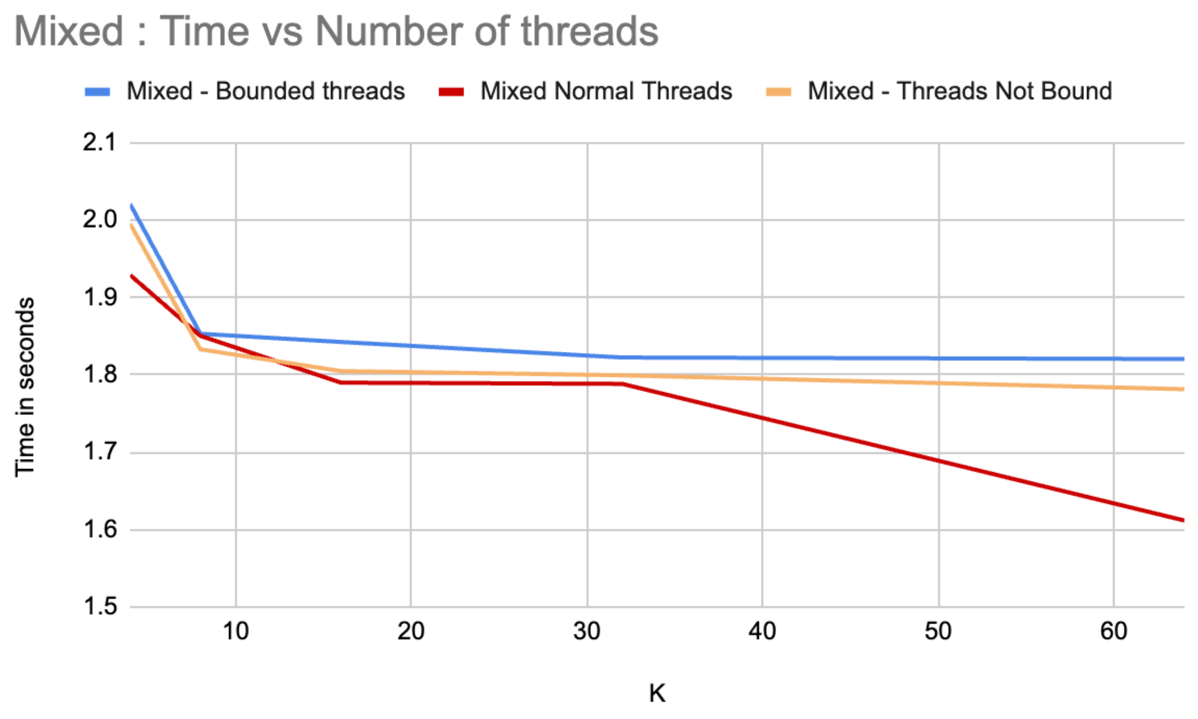
<u>**Cache Effects**</u>: Increasing the number of threads can lead to better cache utilization, because the threads are processing data independently. This improved cache utilization can result in fewer cache misses and better overall performance.

One reason why bounded threads are taking less time than non-bounded threads is :

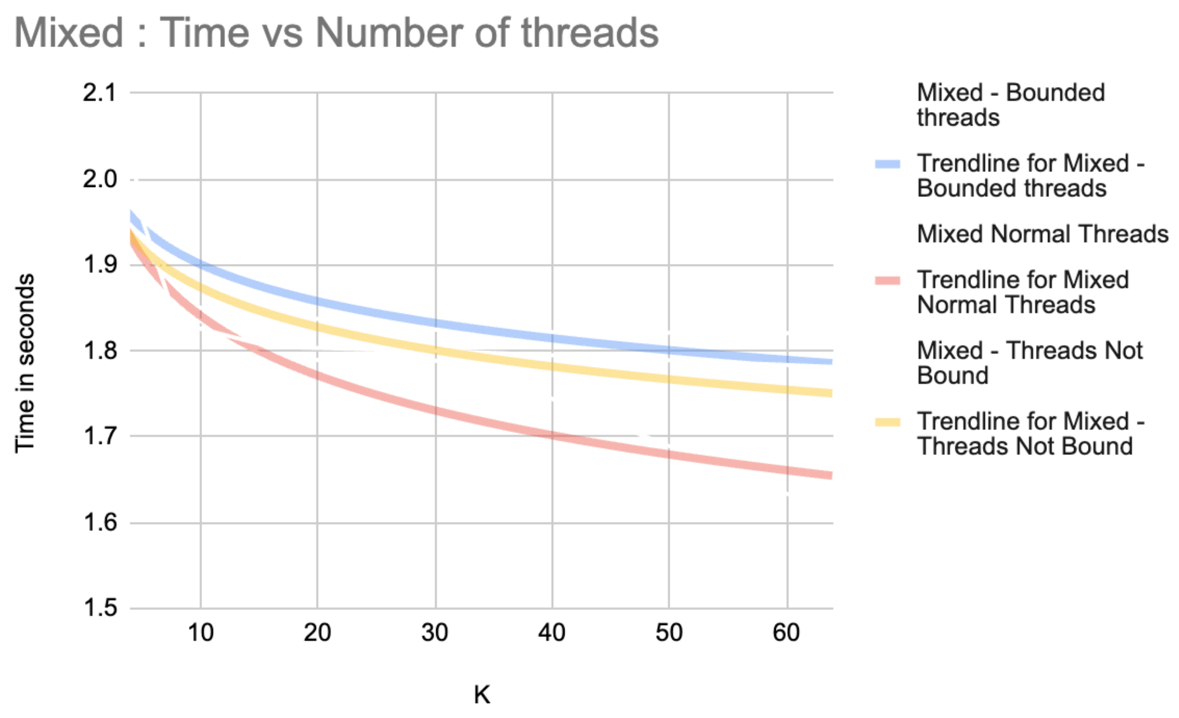
Predictability : Bounded threads are associated with lightweight processes. Since they always run on the same lightweight process, there is no need to frequently create or destroy the lightweight processes. Therefore this ease of predictability will reduce overhead caused to the CPU for instance to predict the time for each thread and make sure that it uniformly distributes thread as per workload.

Next, let us look at mixed algorithm charts

The time in the below graphs are in seconds



Below is the trendline the graph is following



This too is following a similar tread in terms of time vs number of threads… As threads are increasing the average time is decreasing, however here the time required for normal threads are less than bounded threads

Possible reason to this is

1. **General Multithreading Efficiency**:

   For most applications, unbounded threads are suitable. They strike a balance between resource utilization and execution speed.

2. **Flexibility**:

   Unbounded threads are not permanently associated with specific **Lightweight Process (LWP)**. They can run on any available LWP.This flexibility allows efficient caching of LWP and thread resources. LWPs are reused, reducing thread creation and destruction time.

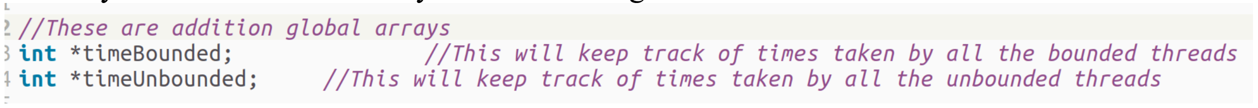
Overall summary of the combined graphs:

- The general trend is that as number of threads increases, the average execution time per thread decreases, with a possible reason that there is better resource utilization and cache efficiency.

- The time for Normal threads in both cases is coming out to be minimum, whereas the time for completely unbounded threads (as in assignment 1), and partially bounded threads using thread affinity is not a clear conclusion.
    - Reasons for both of these possible cases pointed out above
- And, generally the time taken for mixed algorithm is more than chunk when thread affinity is being used. But when completely unbounded, then their times are almost same.

Finally, Points I have kept in mind to ensure efficient and smooth running of code

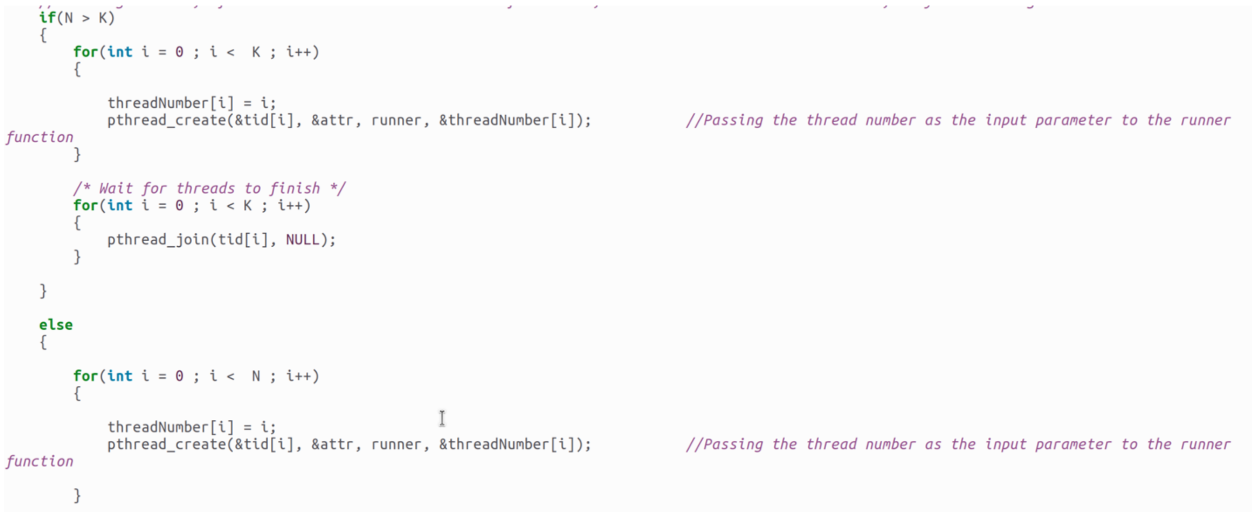
1. Globally I have declared an array and not an integer



```c
//These are addition global arrays
int *timeBounded;               //This will keep track of times taken by all the bounded threads
int *timeUnbounded;     //This will keep track of times taken by all the unbounded threads
```

If instead of global arrays, if I made an int sumBounded and sumUnbounded variables that calculate the sum of time taken by bounded threads and non-bounded threads respectively, then it could've caused race condition and synchronization problem

2. The if-else condition to not create extra threads that may potentially get not used and cause wasteful allocation of CPU resources



```c
    if(N > K)
    {
        for(int i = 0 ; i <  K ; i++)
        {

            threadNumber[i] = i;
            pthread_create(&tid[i], &attr, runner, &threadNumber[i]);           //Passing the thread number as the input parameter to the runner
function
        }

        /* Wait for threads to finish */
        for(int i = 0 ; i < K ; i++)
        {
            pthread_join(tid[i], NULL);
        }

    }

    else
    {

        for(int i = 0 ; i <  N ; i++)
        {

            threadNumber[i] = i;
            pthread_create(&tid[i], &attr, runner, &threadNumber[i]);           //Passing the thread number as the input parameter to the runner
function

        }
```

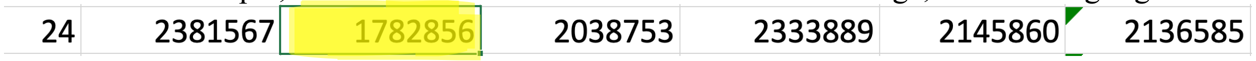
Please note that I have run all my codes on the Virtual Machine of my macbook which has been allocated 4 cores. The results obtained on my TA's device may be different, and this may be due to a different environment (OS), that my TA might be using.

Anomalies in all the observations :

When I ran my code, I calculated the times at night when my laptop was turned on for the entire day, this slowed down the processes very much because the next morning, the same code was running at almost 1/2 to 1/3 of the time.

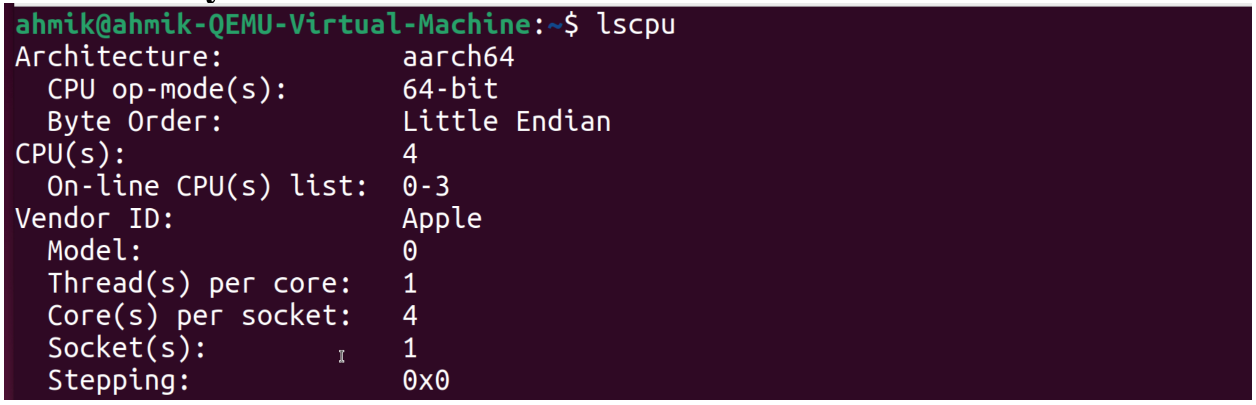
Also, while calculation, I had many outliers, such as listed below, this perhaps affected the average. These outliers may have been caused because the OS did not assign threads efficiently or as I had done in my code, I have first created my thread and then bounded it to a CPU in the runner function. Maybe in those outlier cases, there must have been many context switches, because the OS may have assigned it a different CPU and then I again change it (causing increase in time). Or other case that there is very less context switch where the OS has assigned the threads as I have assigned in the runner function (Optimal case).

In the above example, the first column is BT and last column is average, outlier is highlighted.



24	2381567	1782856	2038753	2333889	2145860	2136585

More on my OS



```
ahmik@ahmik-QEMU-Virtual-Machine:~$ lscpu
Architecture:            aarch64
  CPU op-mode(s):        64-bit
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               Apple
  Model:                 0
  Thread(s) per core:    1
  Core(s) per socket:    4
  Socket(s):             1
  Stepping:              0x0
```

As the pthread_setaffinity_np function was not running on MacOS, I had to do my assignment on the virtual machine which ran Ubuntu Linux on it. I have 4 cores assigned to it.