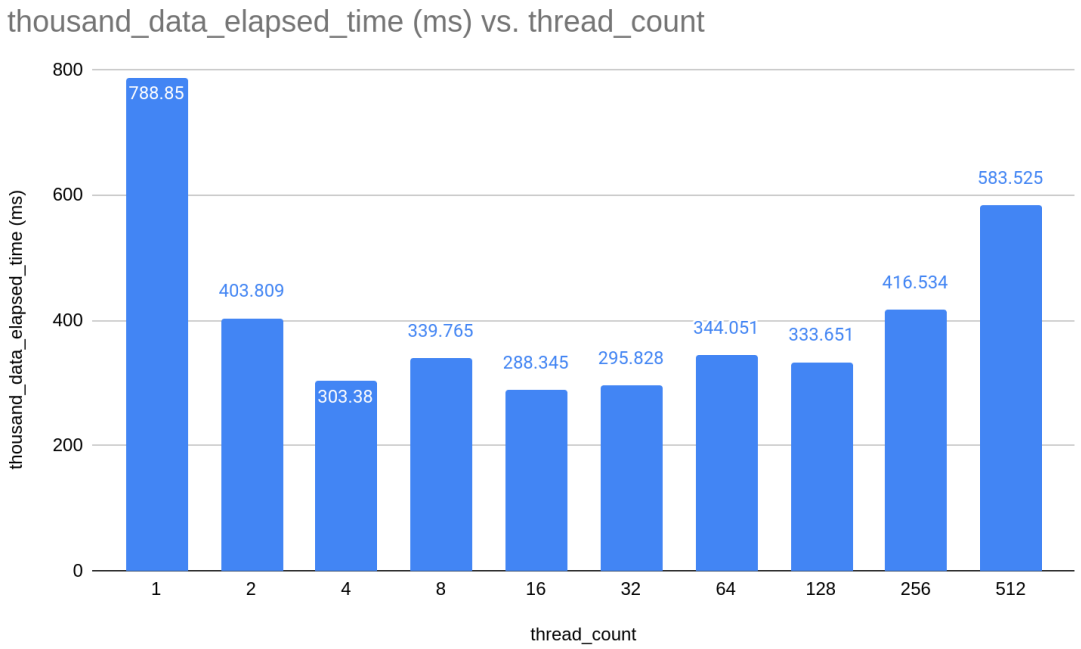
CS 646 - Principles of Operating Systems
Homework 2
Abdur Rouf

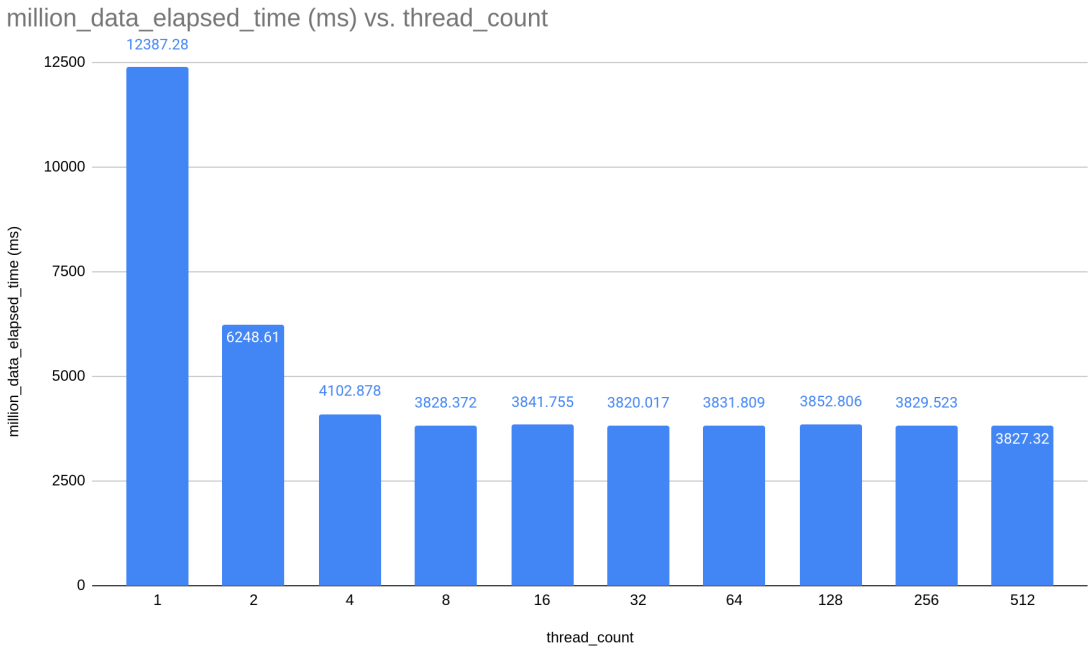
Experimentation: I've run my own two dataset. One has 1000 integers containing a file named "oneThousand.txt" and another one has 1000000 integers containing a file named "oneMillion.txt". Here, I considered these counts given below:
1, 2, 4, 8, 16, 32, 64, 128, 256, 512.
Another thing that should be noted is, I've used my own function to give extra load to the cpu. The function definition is given here,

```
Void extra_load() {
      Long long total_sum = 0;
      for(long long i = 0; i < 10000LL; i++) {
            Total_sum += i;
      }
}
```

So, I run the loop 500000 times for oneThousand dataset and 10000 times for oneMillion dataset.

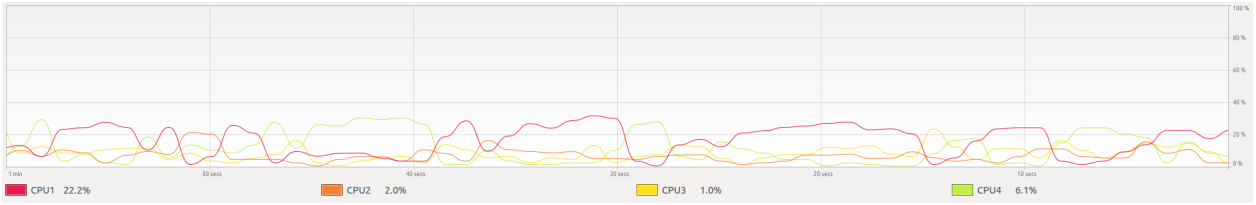
Results: Here, are the graph of time vs thread_count for oneThousand and oneMillion respectively:



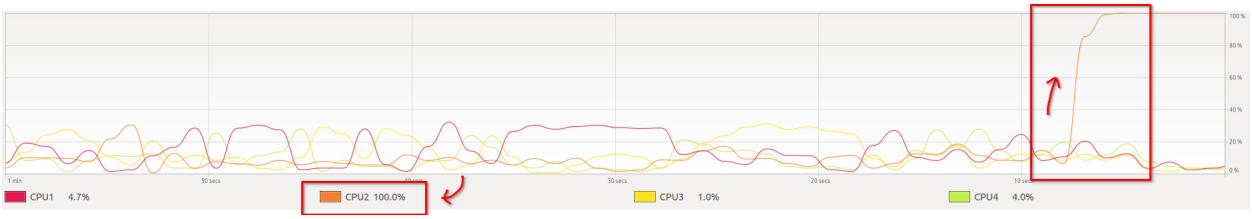


Observations:

a) Initially, I have to mention that my machine processor has 4 physical cores. My observation is the elapsed time is approximately inversely proportional with the number of threads and both graphs are saying exactly the same thing. After that, the elapsed time fluctuates irregularly within a short range. So, based on the cpu cores, it's obvious that we can actually achieve certain improvement by incorporating threads. Here are some snaps from system monitors when I ran the code.

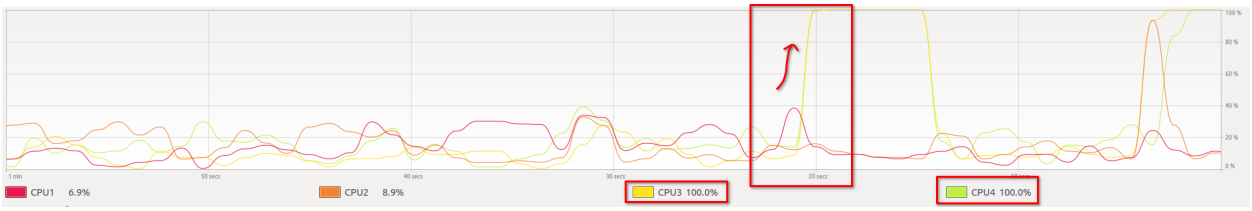


This is the situation before running my threaded_sum code.

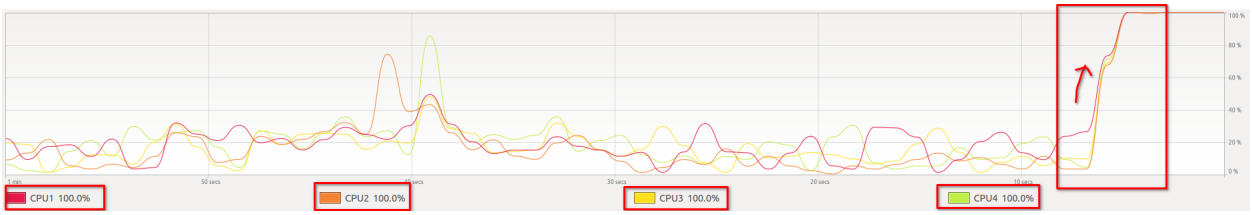


This snap illustrates the situation when I ran the code using 1 thread. The figure clearly demonstrates the utilization of a core which is 100%.

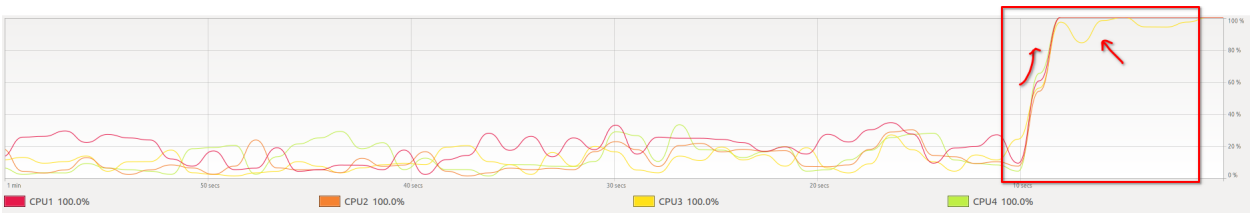
My other observation from single threaded run is, it doesn't occupy only one core all time. Sometimes, the job can be scheduled in a separate cpu but not concurrently. OS scheduled different cpu or core one after another.



This snap shows the situation for running code on two threads. The system monitor snap is also self explanatory.



Here, I ran the code on 4 threads. This snap also says the same thing as above.



Here, the code was run on 8 threads. But, we can see here is a drop of utilization line for the yellow core for a certain period of time. This thing happened for scheduling more threads on top of 4 cores.

b) Yes, I find the observed behavior reasonable. Well, the CPU has 4 cores and that's the reason for being consistent till the thread count is 4. After that, the elapsed time reduction is not consistent because the OS has to schedule more jobs on a constant number of physical cores. That is why, for higher count of threads, the OS has to work more for scheduling extra threads which gives inconsistent elapsed time.

c) I put the lock right before the calculation of total_sum for two reasons.
   1. This is the only memory location which is written from different threads.
   2. Each thread summing loop actually accesses a separate chunk of memory that's why we only need the lock before total_sum calculation.

   Let's discuss in depth:
   Here is the critical section code in c:

```
*(thread_data->totalSum) += threadSum; //critical section
```

Here is the critical section code in assembly:

Line 1: LOAD register1, threadSum
Line 2: LOAD register2, [thread_data->totalSum]
Line 3: ADD register2, register2, register1
Line 4STORE [thread_data->totalSum], register2

From the assembly code, we can see that updating the total sum requires two **read**, one **add** and one **write** operation required. If we don't lock this section, there is a chance that another thread might change the totalSum which affects the calculation and will give wrong results.

P.T.O

EXTRA

Before going to the discussion, we need to see the current implementation and what will be the updated implementation.

Current implementation:

```
void * arraySum(void * arg) {
      thread_data_t *thread_data = (thread_data_t *)arg;
      long long int threadSum = 0;
      for(int i = thread_data->startInd; i < thread_data->endInd; i++) {
          threadSum += thread_data->data[i];
      }
      pthread_mutex_lock(thread_data->lock);
      *(thread_data->totalSum) += threadSum; //critical section
      pthread_mutex_unlock(thread_data->lock);
      pthread_exit(NULL);
}
```

Updated Implementation:

```
void * arraySumEXTRA(void * arg) {
      thread_data_t *thread_data = (thread_data_t *)arg;
      long long int threadSum = 0;
      for(int i = thread_data->startInd; i < thread_data->endInd; i++) {
          pthread_mutex_lock(thread_data->lock);
          //critical section
          *(thread_data->totalSum) += thread_data->data[i];
          pthread_mutex_unlock(thread_data->lock);
      }
      pthread_exit(NULL);
}
```

In the updated implementation, we need to apply mutex lock to this section so that other threads can't update the totalSum at the same time and reason explained in (**c)** . And this implementation actually blocks other threads for each data of its own thread which actually makes the implementation more like single threaded. I ran the code for implementation and the first one performs better than the second one.

For the first implementation, I used 4 threads and the elapsed time was **1.14 ms** *(I removed the*

extra_load() for this experiment).
And for the second implementation, I used 4 threads, the elapsed time was **53 ms**.
I also ran the first implementation for a single thread and the runtime was **2.4ms.**

So, the second implementation performs significantly worse because of two reasons, the mutex lock converted the implementation like single threaded and the usage of multi-thread added more overhead for lock and unlocking the critical section as well as context switching.