# CS610 - Programming for Performance
# 2020-2021-I: Assignment 4

Name: Sarthak Singhal Roll No.: 170635

Earlier there was some issue with tbb on running problem1 on devcloud but I figured out the solution after doing evaluations on cse machines. So, I did evaluations for problem1 on csews10.cse.iitk.ac.in and for the rest of the problems I did evaluations on the compute node of devcloud(i.e. connecting to login node by ssh and then running *qsub -I* to connect to the compute node).

## System I Specifications(csews10)

- Model name: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
- L1d cache: 32KB
- L2 cache: 256KB
- L3 cache: 12288KB
- Sockets: 1
- Cores per socket: 6
- Threads per core: 2
- $\therefore$ Logical cores: 12
- gcc version: 7.5.0

## System II Specifications(compute node of devcloud)

- Model name: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
- L1d cache: 32KB
- L2 cache: 1024KB
- L3 cache: 19712KB
- Sockets: 2
- Cores per socket: 6
- Threads per core: 2
- $\therefore$ Logical cores: 24
- gcc version: 7.4.0

## 1. Problem 1

**Compilation and execution instructions:**

- g++ -std=c++11 -fopenmp fibonacci.cpp -o fibonacci -ltbb
- ./fibonacci ⟨num_threads⟩ ⟨threshold⟩

**Explanation and results:**
This problem was evaluated on csews10 and the num_threads was kept to 12 and some experiments were done for the threshold(to be explained later). In the OpenMP version1, I created tasks for the 2 recursive calls to parallelize the execution. But as the number of tasks increased exponentially there were a large number of tasks leading to degrade in performance due to overhead of scheduling. On average the serial version took 70.2 seconds and the OpenMP version1 took 17105.8 seconds. So, I tried to optimize the version1 by invoking the serial code if $n$ was less than equal to a threshold value as making the code parallel for small values of $n$ leads to much scheduling overhead although the task can be completed serially in much lesser time. So the version 2 is same as version 1 with difference that the serial code is invoked if $n$ is less than equal to a threshold.
I also implemented the parallel versions using intel tbb with both blocking and continuation passing style. To reduce the number of tasks created, these 2 also invoke the serial version if $n$ is less than equal to a threshold. The OpenMP version1, version2, tbb blocking style and tbb continuation passing style are implemented in the functions *omp_fib_v1*, *omp_fib_v2*, *tbb_fib_blocking* and *tbb_fib_cps* respectively.
I tried executing the code with different threshold values like 10,15,20,25,30. The execution time(in sec) for these experiments:

| **Threshold** | **omp v2** | **tbb blocking** | **tbb cps** |
|---|---|---|---|
| **10** | 22.2 | 21.8 | 17.4 |
| **15** | 28.5 | 12.8 | 11.9 |
| **20** | 22.1 | 12.0 | 11.4 |
| **25** | 19.3 | 11.8 | 11.3 |
| **30** | 19.5 | 11.4 | 11.3 |

On average increasing threshold increases the performance as the scheduling overhead is reduced but on increasing it too much will lead to very less parallelism which will also degrade the performance. So, we'll take threshold = 30. So, to get similar results you should run the executable as: *./fibonacci 12 30*. OpenMP version2 has a speedup of 70.2/19.5 = 3.6. Both versions of tbb perform better than OpenMP version2 because of better work stealing and optimal scheduling done in tbb. The blocking style gives a speedup of 70.2/11.4 = 6.15 and the continuation passing style gives a speedup of 70.2/11.3 = 6.21. The slight better performance of cps as compared to blocking style is due to the fact that the parent task now has not to wait till the children tasks complete.

## 2. Problem 2

**Compilation and execution instructions:**

- g++ -std=c++11 -fopenmp quicksort.cpp -o quicksort
- ./quicksort ⟨num_threads⟩ ⟨threshold⟩

**Explanation and results:**
This problem was evaluated on devcloud and the num_threads was kept to 24 and different values of N were tried(for further discussion take $N = 2^{21}$) and some experiments were done for the threshold(to be explained later). In the parallel implementation using OpenMP, I created tasks for the 2 recursive calls of quicksort to parallelize the execution. I also tried to optimize it by invoking the serial code if the length of array under consideration at a particular recursive call was less than equal to a threshold value as making the code parallel for small values of length can lead to much scheduling overhead although the task can be completed serially in much lesser time. The serial version on average took 10.4 seconds for $N = 2^{21}$. The OpenMP version is implemented in the function *par_quicksort.*
I tried executing the code with different threshold values like 1000,2000,3000,4000,5000. The execution time(in sec) for these experiments:

| **Threshold** | **par_quicksort** |
|---|---|
| **1000** | 1.85 |
| **2000** | 1.01 |
| **3000** | 1.01 |
| **4000** | 0.99 |
| **5000** | 1.00 |

On average increasing threshold increases the performance as the scheduling overhead is reduced but on increasing it too much will lead to very less parallelism which will also degrade the performance. We'll take threshold = 4000. So, to get similar results you should run the executable as: *./quicksort 24 4000.* The OpenMP version has the speedup of 10.4/0.99 = 10.5. I also tried parallelizing the partition function by parallelizing the for loop in it. The parallel partition is implemented in the function *par_partition.* So, to use the parallel partition function in parallel quicksort you have to just the change the *partition* to *par_partition* inside the parallel quicksort. But it degrades the performance which maybe due to more scheduling overhead or it is maybe because I have used an extra array and an extra for loop in the parallel version of partition so that I can parallelize the loop, but including them is maybe decreasing the performance. Although, in the parallel version of quicksort I have not used the parallel partition by default and the above table is for the parallel quicksort with serial partition.

## 3. Problem 3

**Compilation and execution instructions:**

- g++ -std=c++11 -fopenmp pi.cpp -o pi -ltbb
- ./pi ⟨num_threads⟩

**Explanation and results:**

This problem was evaluated on devcloud and the num_threads was kept to 24. The serial version on average took 34.34 seconds. In the OpenMP version I used reduction on the sum variable and there is no scope of false sharing in this version. The OpenMP version took 1.51 seconds on average. For the Intel TBB version I used parallel reduce and it took 1.49 seconds on average. So, the speedups for OpenMP and Intel TBB are 34.34/1.51 = 22.74 and 34.34/1.49 = 23.04 respectively.

The OpenMP version and the tbb version are implemented in the functions *omp_pi* and *tbb_pi* respectively.

To get the similar results you should run the executable as: *./pi 24*. This can be observed that both the speedups are around 24 and hence all the logical cores(of devcloud) are exploited to a great extent.

## 4. Problem 4

**Compilation and execution instructions:**

- g++ -std=c++11 find-max.cpp -o find-max -ltbb
- ./find-max

**Explanation and results:**
This problem was evaluated on devcloud and different values of N were tried(for further discussion take $N = 2^{26}$). The serial version on average took 0.15 seconds. For the Intel TBB version I used parallel reduce and it took 0.035 seconds on average. So, the speedup for Intel TBB is 0.15/0.035 = 4.28.
The tbb version is implemented in the function *tbb_find_max*.