Assignment 4

Aniket Sanghi (170110)

13th November 2020

I pledge on my honor that I have not given or received any unauthorized assistance.

System and Architecture Information (Used CSE server)

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list: 0-11
Thread(s) per core:  2
Core(s) per socket:  6
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
Stepping:              10
CPU MHz:               900.308
CPU max MHz:           4600.0000
CPU min MHz:           800.0000
BogoMIPS:              6384.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0-11
```

Compilers Information

```
g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
```

System SSH

```
ssh <username>@csews16.cse.iitk.ac.in
```

Problem 1 has been executed on this server

1 Fibonacci

1.1 Compilation and execution instruction

For GCC

```
g++ -std=c++11 -fopenmp fibonacci.cpp -o fibonacci -ltbb
./fibonacci
```

1.2 (i) OpenMP explicit tasks

The code ran approximately 200 times slower than the serial version on the system above.

The high slowdown can be attributed to the scheduling overhead of the task schedular, due to the very large number of small tasks created (particularly near the base case of recursion).

1.3 (ii) OpenMP explicit tasks Optimised

The final optimised code gave a speed up of around **3.5X**.

The primary slowdown in the above case was due to the very huge number of explicit tasks, so to reduce that, we added a cut-off. This will execute serialised version of fibonacci for all calls to value less than cut-off. By this, we will have few explicit tasks having large work rather than large number of tasks with small work.

Performance observed: **1.8X, 2.6X and 3.5X speedup for 4, 8 and 12 threads respectively**
Performance observed: **3.1X, 3.5X and 3.4X speedup for N = 20, 25, 30 cut-off respectively [12 threads]**

1.4 (iii) Intel TBB (Blocking Style)

The code gave a speed-up of **6.2X** as compared to serialised code.

The better speed-up as compared to OpenMP can be attributed to the optimised task scheduling and handling done by TBB.

1.5 (iv) Intel TBB (Continuation Passing Style)

The code gave a speed-up of **6.5X** as compared to serialised code.

The better speed-up as compared to above code can be attributed to the continuation passing style. Where optimised and better work-stealing can be used to execute the continuation code by any other/same task.

System and Architecture Information (Used DevCloud)

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz
Stepping:              4
CPU MHz:               1200.183
CPU max MHz:           3700.0000
CPU min MHz:           1200.0000
BogoMIPS:              6800.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              19712K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
```

Compilers Information

```
g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
```

All the following problems have been tested in this system

2 QuickSort

2.1 Compilation and execution instruction

For GCC

```
g++ -std=c++11 -fopenmp quicksort.cpp -o quicksort
./quicksort
```

2.2 Performance Report

The final code gave a speed-up of **3.8X** as compared to serialised version.

The code gave a poor performance with just making quick-sort recursive calls done by explicit tasks because of the huge number of tasks created. So, to resolve this, below a cut-off, called quick sort like serial version.

Performance: Code gave 2.2-2.5X speed-up on values around 100 and 10000, best speedup was observed on cut-off value around 1000 on various values of N ($2^{16}, 2^{21}, 2^{22}$). On 2^{20}, it gave more than **10X** speedup.

Tried to parallelise the **partition** code also but it slowed down the code further, primary reason could be because of partitioning small job and also synchronisation between threads.

3 Pi

3.1 Compilation and execution instruction

For GCC

```
g++ -std=c++11 -fopenmp pi.cpp -o pi -ltbb
./pi
```

3.2 Performance Report

Both codes (OpenMP and TBB) gave a speed-up of **23X** as compared to serialised version.

Used reduction in OpenMP and dynamic scheduling with various chunk sizes, the chunk size of 10^6 gave the best speedup. Best performance was observed in default and 24 threads. So, used default in code. Similarly in TBB reduction was used to parallelise the code. No issues of false sharing observed as no arrays been used.

4 Find Max

4.1 Compilation and execution instruction

For GCC

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

4.2 Performance Report

The code gave a speed-up of **4X** as compared to serialised version.

Used TBB parallel-reduce to parallelise the code.