Assignment 4

Ayush Kumar (170195)

9 November 2020

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

Solutions

Please note that all benchmarking for this assignment (except Problem 1) was performed on the Intel DevCloud using Intel's OneAPI. The specifications of the Linux system of the compute node on which the calculations were performed are listed below. Also, the version of the GCC compiler used is **gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1 18.04.1)**. For all evaluations that we performed in this assignment, we took the median runtime of 5 consecutive runs of the program.

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:	1200.480
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

Table 1: System Specifications of Intel Devcloud

1. For this question, we used a different machine for our testing as the Intel Devcloud had issues with the Intel TBB library. The specifications of the local machine (Macbook Pro Late 2013 model) are provided below.

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	2
Core(s) per socket:	2
Socket(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	69
Model name:	Intel(R) Core(TM) i5-4258U CPU @ 2.40GHz
Stepping:	1
CPU MHz:	2400.0000
CPU max MHz:	3000.0000
CPU min MHz:	2000.0000
L1d cache:	32K
L1i cache:	32K
L2 cache (per core):	256 KB
L3 cache:	3 MB

Table 2: System Specifications of Local Machine

To run the code, use the following command:

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

The runtime for $N = 50$ for the **serial implementation** of Fibonacci comes out to be **123.168672 seconds**.

(a) For parallelizing the serial Fibonacci code, we use tasks in OpenMP, where each node in the recursion tree is a task that can be executed by any available thread. For any given node **Fib(n)** in the tree, the thread executing the task creates two child tasks **Fib(n-1)** and **Fib(n-2)** and then waits for their completion before returning their sum. The final runtimes obtained for different number of threads for $N = 50$ is shown below.

Number of Threads	Runtime (in seconds)	Speedup
2	9077.933953	0.013568
3	7522.167024	0.016374
4	**6965.777790**	**0.017681**

Table 3: Runtimes for fibonacci, parallelized using OpenMP tasks

We notice that for each case, the speedup is actually much less than 1 and the runtimes increase by a huge factor. The reason behind this is quite evident. Since each task is quite small and the number of tasks are quite large (due to the recursive approach), thread scheduling and task creation are quite frequent and their costs largely outweigh any gains obtained by parallelization.

(b) The initial approach is now **optimized by increasing the cutoff for serial computation**. Earlier all tasks were run in parallel. Now, we enforce that for $n \leq cutoff$, the computation will be performed serially. Clearly, this will make the terminal tasks larger and also reduce the total number of tasks hence reducing a lot of the scheduling/task creation overheads. We provide the runtimes obtained for $N = 50$, $num\_threads = 4$ for different cutoffs.

Cutoff	Runtime (in seconds)	Speedup
10	135.556208	0.908617
20	51.365900	2.397868
25	51.165570	2.407257
30	**50.484653**	**2.439725**
35	51.051066	2.412656
40	53.615885	2.297242

Table 4: Runtimes for fibonacci, parallelized/optimized using OpenMP tasks

We now obtain a significant gain with a speedup of over 2 compared to the serial implementation. This indicates that out presumption was correct and **increasing the size of terminal tasks reduces the number of tasks and hence the overall overhead**. This is basically **equivalent to controlling the grain size** in Intel TBB. We already know that there exists a fine balance between grain size and the number of tasks that can be run in parallel, here we improve that balance to improve runtime.

In fact recursion is not at all an efficient approach for calculating fibonacci numbers. There is a much more efficient way of doing this using Dynamic Programming.

(c) The runtime obtained for $N = 50$ with Intel TBB using the blocking style and auto-partitioner comes out to be **2118.788786 seconds**. This is much less than the one obtained with 4 threads using OpenMP version 1. The reason is that the auto-partitioner automatically divides the workload appropriately. Also, since **Intel TBB binds a task and its child tasks to a thread** (only allowing work stealing from the back of the queue), **scheduling overheads decreases** causing lesser runtimes.

Using a **cutoff of 30**, we get a runtime of **52.667360 seconds (speedup 2.338615)** which appears to be similar to the one obtained with OpenMP version 2.

(d) Using Intel TBB with continuation passing style and $N = 50$ (again using auto-partitioner), we obtain a runtime of **2268.082700 seconds**. This is slightly greater than the blocking style perhaps due to the **extra cost of creating a new continuation task** and destroying the parent task after spawning the child tasks.

With a **cutoff of 30**, we get a runtime of **55.520535 seconds (speedup 2.218435)** which is again slightly greater than the blocking style implementation.

2. To run the code, use the following command:

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

We again parallelize quicksort using OpenMP tasks. More specifically, after partitioning a subarray based on a given pivot, sorting the left and right parts of the partition become two independent child tasks. Note that here, the parent task does not need to wait for the child tasks to complete before returning. Varying the number of threads used, we get the following runtimes for different sizes of the array. Best performance was obtained with different number of threads on different sizes of the array. We test and compare the performances for $N = 2^{16}, 2^{18}, 2^{20}, 2^{22}$.

(a) $N = 2^{16}$
Runtime of Serial implementation: **0.022997 seconds**
Best runtime of Parallel implementation: **0.023192 seconds (using 2 threads)**

Number of Threads	Runtime (in seconds)	Speedup
2	**0.023192**	**0.991592**
4	0.036126	0.636577
6	0.050865	0.452118
12	0.096065	0.239390
24	0.209687	0.109673

Table 5: Runtimes for quicksort, parallelized using OpenMP tasks, $N = 2^{16}$

(b) $N = 2^{18}$
Runtime of Serial implementation: **0.190848 seconds**
Best runtime of Parallel implementation: **0.131612 seconds (using 4 threads)**

Number of Threads	Runtime (in seconds)	Speedup
2	0.132808	1.437022
4	**0.131612**	**1.450081**
6	0.162800	1.172285
8	0.196648	0.970506
12	0.278472	0.685340
24	0.554645	0.344090

Table 6: Runtimes for quicksort, parallelized using OpenMP tasks, $N = 2^{18}$

(c) $N = 2^{20}$
Runtime of Serial implementation: **2.654962 seconds**
Best runtime of Parallel implementation: **0.868560 seconds (using 6 threads)**

Number of Threads	Runtime (in seconds)	Speedup
2	1.419758	1.870010
4	0.901909	2.943713
6	**0.868560**	**3.056740**
8	0.971955	2.731569
12	1.356494	1.957223
24	1.947028	1.363597

Table 7: Runtimes for quicksort, parallelized using OpenMP tasks, $N = 2^{20}$

(d) $N = 2^{22}$
Runtime of Serial implementation: **41.251420 seconds**
Best runtime of Parallel implementation: **6.834705 seconds (using 10 threads)**

Number of Threads	Runtime (in seconds)	Speedup
4	10.758996	3.834132
6	7.617194	5.415566
8	6.845063	6.026449
10	**6.834705**	**6.035581**
12	7.071603	5.833390
16	8.295298	4.972868
24	13.950856	2.956910

Table 8: Runtimes for quicksort, parallelized using OpenMP tasks, $N = 2^{22}$

We observe that **as N increases, the amount of parallelization and hence the speedup achieved also increases**. The reason is that for small N, the gains due to parallelization are small compared to the thread creation and scheduling overheads, hence for $N = 2^{16}$, the performance slightly degrades even with 2

threads. In contrast, for $N = 2^{22}$, performance improves by almost 6 times. Also, note that **the number of threads required for best performance also increases with** N. This can again be attributed to the fact that dividing a large workload among a large number of threads is more efficient than dividing a smaller workload among the same number of threads.

The **speedups can be improved by setting a higher cutoff for serial computation** as was done in problem 1 part (b) to optimize the version 1 of OpenMP parallel code. For example, by enforcing that arrays of size less than 1000 get sorted using *serial_quicksort*, we can gain significant speedups even for $N = 2^{16}$. We show the speedups obtained using cutoffs below.

N, Cutoff	Number of Threads	Runtime (in seconds)	Speedup
	8	0.008586	2.678430
$2^{16}, 1000$	**10**	**0.007275**	**3.161100**
	12	0.007305	3.148118
	12	0.035260	5.412592
$2^{18}, 2000$	**16**	**0.034376**	**5.551780**
	20	0.035401	5.391034
	16	0.245832	10.799904
$2^{20}, 4000$	**20**	**0.244210**	**10.871635**
	24	0.235908	11.254226
	16	3.415934	12.076176
$2^{22}, 8000$	**20**	**3.320931**	**12.421643**
	24	3.212950	12.839110

Table 9: Runtimes for quicksort using varying cutoffs, parallelized using OpenMP tasks

Clearly, the **speedups increase by a large factor**. This is because we increased the grain size, thereby reducing the number of tasks and hence the scheduling and task creation overheads. This enables us to extract more parallelism out of programs. We can see this for $N = 2^{16}, 2^{18}, 2^{20}, 2^{22}$ where now parallelizing with more number of threads (10, 16, 24, 24 respectively compared to the previously used 2, 4, 6, 10 threads) gives much better runtimes.

3. To run the code, use the following command:

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

The runtime for the **serial implementation** of π computation comes out to be **34.451188 seconds**. We first parallelize it using OpenMP tasks. Varying the number of threads used, we get the following runtimes. Best performance of **1.545726 seconds** was **obtained with 24 threads**.

Number of Threads	Runtime (in seconds)	Speedup
4	8.775655	3.925768
8	4.388131	7.850994
12	2.943511	11.704114
16	2.263376	15.221151
20	1.828592	18.840281
24	**1.545726**	**22.288030**

Table 10: Runtimes for π computation, parallelized using OpenMP tasks

We now parallelize this using Intel TBB. We used the *parallel_reduce*() method to divide the iteration range into subranges and perform a sum reduction to get the final result which would be multiplied by 4 to get the value of π. We used the **default auto-partitioner** and the **default grain size of 1** for dividing the ranges. A runtime of **1.507988 seconds (speedup 22.845797)** was observed. This is a slight improvement

from the OpenMP implementation which indicates that Intel TBB's auto-partitioner divides the work more appropriately among the threads.

Also note that we do not encounter any issue of false sharing because no two data elements accessed by different threads are close enough to be located in the same cache line. Each task body (instance of $TbbPiTask$) has its own private copy of the variable $partial\_sum$ in which the result of each subrange is stored and is finally reduced according to the $join()$ method. Even for the OpenMP version, each thread creates its own copy of the variable sum in which the partial sum of the subrange is stored. This is reduced in the end to the sum variable of the outer lexical scope.

4. To run the code, use the following command:

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

The **runtime for serial version of find-max** obtained on an array size of $N = 2^{26}$ comes out to be **0.150125 seconds**. Similar to the previous problem, we parallelize this using Intel TBB's $parallel\_reduce()$ method. Again we tested the runtimes for $N = 2^{26}$ using the **default auto-partitioner** and a **default grain size of 1**. A runtime of **0.035058 seconds (speedup 4.282190)** was obtained. We compare the results on other array sizes.

Size of Array	Serial Runtime (in seconds)	Parallel Runtime (in seconds)	Speedup
2^{20}	0.004339	0.005630	0.770693
2^{22}	0.009446	0.007905	1.194940
2^{24}	0.037572	0.014953	2.512673
2^{26}	**0.150125**	**0.035058**	**4.282190**

Table 11: Runtimes for $find\_max$, parallelized using Intel TBB

Once again, we see that the **speedups become more significant when the total workload becomes larger**. On smaller workloads, the gains due to parallelization are minor and is just enough to offset the overheads caused.