CS610 : Assignment 4

Parth Sharma
20111039
prthshrma20@iitk.ac.in

Configuration

Personal System

- Processor : Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz
- RAM : 8GB
- Number of cores : 2
- Thread(s) per core : 2
- Logical Processors : 4
- 64-bit Operating System
- L1d cache : 64 KB
- L1i cache : 64 KB
- L2 cache : 512KB
- L3 Cache : 3MB

Intel Devcloud

- Processor : Intel(R) Xeon(R) Gold 6128 CPU@ 3.40GHz
- CPU(s) : 24
- Threads per Core: 2
- Cores Per Socket: 6
- Socket : 2
- L1d cache=32KB
- L1i cache=32KB
- L2 cache=1024KB
- L3 Cache=19712KB

NOTE: Intel TBB on Intel Devcloud is compiled after creating script file as it was mentioned on Piazza and hence, the provided command to compile on piazza is used to compile the code on Intel Devcloud.

Problem 1

Solution

All experiments are performed with N=50.

Observation

On Personal System,

Command to Compile : g++ -std=c++11 20111039-p1 -fopenmp -ltbb -o 20111039-p1.cpp
Command to Run : ./20111039-p1

Implementation	Time in micro seconds	Speedup
Serial version	162108934	-
Explicit OMP tasks	1253365474	0.12
Optimized OMP tasks	78206316	2.07
Blocking style parallelism in Intel TBB	83809970	1.92
Continuation passing style parallelism in Intel TBB	83498006	1.93

Table 1: Performance of fibonacci calculation on Personal System

On Intel Devcloud,

Command to Compile : g++ -o 20111039-p1.out 20111039-p1.cpp -fopenmp -pthread -ltbb -I ./tbb/include -L ./tbb/lib
Command to Run : ./20111039-p1.out

Implementation	Time in micro seconds	Speedup
Serial version	74771943	-
Explicit OMP tasks	702965365	0.106
Optimized OMP tasks	8035238	9.3
Blocking style parallelism in Intel TBB	6745358	11.08
Continuation passing style parallelism in Intel TBB	7275606	10.27

Table 2: Performance of fibonacci calculation on Intel Devcloud

After observing data, we can conclude that,

- While using explicit OpenMP tasks, we get less speedup i.e. more execution time than the usual serial code. This happens because the code is un-optimized as it creates tasks upto n=1 which results in more tasks creation. Handling large amount of tasks compared to threads, their context switching becomes overhead and hence increases the execution time.
- Optimizing OpenMP code by calling serial version for $n <= 20$ consequently increases the speedup by reducing the more task creation overhead.
- Optimized version of OpenMP creates parallel region using **#pragma omp parallel** for the tasks, which makes tasks to perform in parallel, consequently increasing the speedup.
- Blocking Style and Continuation passing Style gives similar though fair speedup. Intel TBB task scheduler implicitly manages tasks which results in less overhead and hence provide good performance.

Problem 2

Solution

Experiments are performed on three different values of N.
Command to Compile: g++ -std=c++11 -fopenmp 20111039-p2.cpp -o 20111039-p2
Command to Run: 20111039-p2

Observation

On Personal System,

Different value of N	Serial version	Parallel version with OMP	Speedup
N (1 << 16)	32818	20337	1.61
N (1 << 18)	347506	170240	2.04
N (1 << 20)	4949163	2159834	2.29

Table 3: Time taken for serial and parallel version in microseconds on Personal System

On Intel Devcloud,

Different value of N	Serial version	Parallel version with OMP	Speedup
N (1 << 16)	28605	26719	1.07
N (1 << 18)	191203	44328	4.31
N (1 << 20)	2660940	330446	8.05

Table 4: Time taken for serial and parallel version in microseconds on Intel Devcloud

After observing data, we get to know that,

- Creating tasks in OpenMP as Quicksort is using recursive way to sort the array, helps in getting fair speedup.
- Un-optimized version of Quicksort for array size less than 1024 was working better than parallel, hence used to optimized the parallel OpenMP code.

Problem 3

Solution

Observation

On Personal System,

Command to Compile : g++ -std=c++11 20111039-p3 -fopenmp -ltbb -o 20111039-p3.cpp
Command to Run : ./20111039-p3

Implementation	Time in micro seconds	Speedup
Serial version	34565275	-
OpenMP parallel algorithm	9845289	3.51
Intel TBB parallel algorithm	9963926	3.46

Table 5: Performance of pi computation on Personal System

On Intel Devcloud,

Command to Compile : g++ -o 20111039-p3.out 20111039-p3.cpp -fopenmp -pthread -ltbb -I ./tbb/include -L ./tbb/lib
Command to Run : ./20111039-p3.out

Implementation	Time in micro seconds	Speedup
Serial version	34350564	-
OpenMP parallel algorithm	1509453	22.7
Intel TBB parallel algorithm	1487527	23.1

Table 6: Performance of pi computation on Intel Devcloud

After observing data, we pointed out that,

- OpenMP and Intel TBB both are having similar speedup.
- Both parallel algorithms used Parallel Reduction concept, where **#pragma omp parallel reduction** is used in OpenMP and parallel_ reduce() and blocked_ range() in Intel TBB without any explicit partitioner so that it partitions the intervals according to the threads.

Problem 4

Solution

Observation

On Personal System,

Command to Compile : g++ -std=c++11 20111039-p4 -fopenmp -ltbb -o 20111039-p4.cpp
Command to Run : ./20111039-p4

Implementation	Time in micro seconds	Speedup
Serial version	279614	-
Intel TBB parallel algorithm	205164	1.36

Table 7: Performance of find-max on Personal System

On Intel Devcloud,

Command to Compile : g++ -o 20111039-p4.out 20111039-p4.cpp -fopenmp -pthread -ltbb -I ./tbb/include -L ./tbb/lib
Command to Run : ./20111039-p4.out

Implementation	Time in micro seconds	Speedup
Serial version	150007	-
Intel TBB parallel algorithm	35960	4.17

Table 8: Performance of find-max on Intel Devcloud

After observing data, we can conclude that,

- Parallel reduction concept with dividing the range of array and assignment of subranges to different threads considerably reduce the execution time and increase the speedup.
- Intel TBB parallel algorithm on Intel Devcloud is providing better speedup than on Personal System as there are more logical processors available on Intel Devcloud.
- Join function also contain the logic of finding lower index in case of duplicate maximum numbers.