Class: Final Year (Computer Science and Engineering)
Year: 2023-24
Semester: 1
Course: High Performance Computing Lab

Practical No. 1

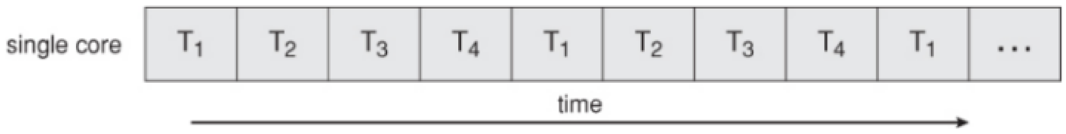
Exam Seat No.: 2020BTECS00029
Name: Akshata Gawande
Title of practical: Study and implementation of basic OpenMP clauses

Q1. Differentiate between Software and Hardware Threads
Solution:
Hardware Thread: A "hardware thread" is a physical CPU or core. So, a 4 core CPU can genuinely support 4 hardware threads at once - the CPU really is doing 4 things at the same time. One hardware thread can run many software threads. In modern operating systems, this is often done by time-slicing - each thread gets a few milliseconds to execute before the OS schedules another thread to run on that CPU. Since the OS switches back and forth between the threads quickly, it appears as if one CPU is doing more than one thing at once, but in reality, a core is still running only one hardware thread, which switches between many software Threads.

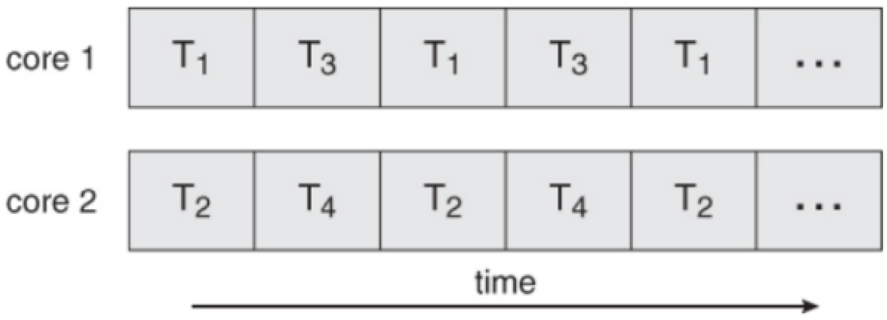
Software Thread: Software threads are threads of execution managed by the operating system. Software threads are abstractions to the hardware to make multi-processing possible. If you have multiple software threads but there are not multiple resources then these software threads are a way to run all tasks in parallel by allocating resources for limited time(or using some other strategy) so that it appears that all threads are running in parallel. These are managed by the operating system.

Q2. Which type of threads are supported by the processor?
Solution:
Generally the Hardware Threads are supported by the processor. The hardware threads are mostly based on the muti-core architecture which is latest architecture to achieve high performance.
A multi-threaded application running on a traditional single-core chip

would have to interleave the threads, as shown in Figure 4.3. On a multi-corechip, however, the threads could be spread across the available cores, allowing
true parallel processing.



Concurrent execution on single core


core 1
T_1 T_3 T_1 T_3 T_1 ...
core 2
T_2 T_4 T_2 T_4 T_2 ...
time

Parallel execution on multiple cores

Problem Statement: Program to print Hello World!

Source Code:

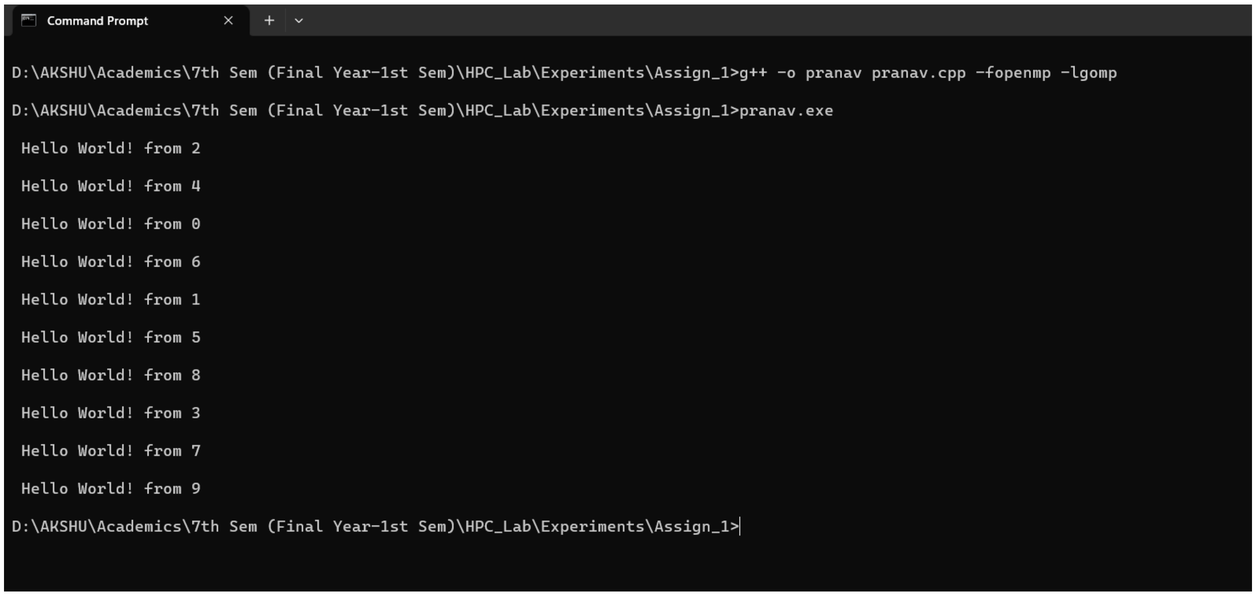
```
#include<stdio.h>
#include<omp.h>
void printMes();

int main(){
#pragma omp parallel num_threads(10)
printMes();
return 0;
}

void printMes(){
int tn;
tn = omp_get_thread_num();

printf("\n Hello World! from %d \n", tn);
```

Output:



```
D:\AKSHU\Academics\7th Sem (Final Year-1st Sem)\HPC_Lab\Experiments\Assign_1>g++ -o pranav pranav.cpp -fopenmp -lgomp

D:\AKSHU\Academics\7th Sem (Final Year-1st Sem)\HPC_Lab\Experiments\Assign_1>pranav.exe

 Hello World! from 2

 Hello World! from 4

 Hello World! from 0

 Hello World! from 6

 Hello World! from 1

 Hello World! from 5

 Hello World! from 8

 Hello World! from 3

 Hello World! from 7

 Hello World! from 9

D:\AKSHU\Academics\7th Sem (Final Year-1st Sem)\HPC_Lab\Experiments\Assign_1>
```

Information:

This is a simple example of using OpenMP (Open Multi-Processing) to create a parallel section of code that will be executed by multiple threads. OpenMP is a popular API for parallel programming in C and C++ that allows you to take advantage of multiple CPU cores to perform parallel computations.

Here's some information about the code:

- “#include<omp.h>”: This line includes the OpenMP header file, which provides the necessary functions and directives for parallel programming.
- “#pragma omp parallel num_threads(10)”: This line begins a parallel section of code using OpenMP. It specifies that we want to create a parallel region with 10 threads. Each thread will execute the code within the parallel region concurrently.
- “void printMes()”: This function is defined separately from “main” and is responsible for printing a message indicating which thread is executing it.
- “tn = omp_get_thread_num()”: “omp_get_thread_num()” is an OpenMP function that retrieves the unique ID of the calling thread. It assigns this ID to the”tn” variable.
- “printf("\n Hello World! from %d \n", tn)”: This line uses “printf” to print a message indicating the thread number (“tn”). Each thread will print its own thread number, allowing you to see which thread is executing which part of the code.
- When we run this program, we'll observe that the "Hello World!" message is printed multiple times, with each message indicating the thread number. The number of times the message is printed depends on the number of threads specified in the “num_threads()” clause of the “#pragma omp parallel directive (in

this case, 10 times). Each thread executes the “printMes” function concurrently, making it a simple parallel program.

Analysis:

The code demonstrates the use of OpenMP for parallel programming in C. Here's a short analysis of the code:

- The code utilizes OpenMP directives to create a parallel region with 10 threads.
- Within the parallel region, the \`printMes()\` function is called by each thread.
- The \`printMes()\` function retrieves and prints the thread's unique identifier.
- As a result, the "Hello World!" message is printed multiple times, each time indicating the thread number.
- This code serves as a basic example of parallelism, allowing multiple threads to execute the same function concurrently. It highlights the simplicity of using OpenMP to parallelize code in C.