National University

of Computer & Emerging Sciences-Lahore

CS3006– Parallel and Distributed Computing

Assignment 3

Deadline: Friday 29th, March 2024

Question No. 1:

Write the program that works by testing each odd number (up to a specified limit) for divisibility by all the factors from 3 to the square root of that number. Your task is to parallelize this algorithm using the **OpenMP**. The program takes two main parameters, which are read in from the command line, **P**: the number of processors (numProcs in the code); and **N**: the problem size (size in the code).

In addition, the program takes arguments that output all of the primes generated, either to a file or standard out. This should assist you in ensuring that the parallel version of the algorithm works correctly.

The main data structure of the program is an array which holds boolean values indicating whether the corresponding numbers are prime. The array only holds odd numbers, since no even numbers (except 2) are prime. Good load balancing is crucial in achieving high performance for parallel workloads. You should be able to achieve very good speedups on this program (i.e., close to linear)

Question No. 2:

Parallelize the following piece of code using **OpenMP** with SIZE=1000. Report sequential execution time, parallel execution time (when using 2 threads, 3 threads and 4 threads), and the achieved speedup (when used 2 threads, 3 threads, and 4 threads).

Note: Your parallel formulation should provide the same answer as provided by the sequential code.

You will have to identify private variables, shared variables, critical section[s], and need for reduction to avoid semantic inconsistencies.

```
int key=111;
for (i=0; I < SIZE; i++) {
   for (j=0; j<SIZE; j++) {
```

```
      for (k=0; k<SIZE; k++) {
         x= (i*i*1000/35) % 1000;
         y= (j*j*1000/36) % 1000;
         z= (k*k*1000/37) % 1000;
         if (key == (x+y+z)) {
         win=win+1;
         }

      }
    }
}
printf("total wins=%ld\n",win);
```

Question No. 3:

Implement a simple parallel linear search algorithm using **OpenMP**. The program works on an integer array of size 'N' filled with random numbers and a key-value to be searched. Write the parallel program to divide the search space among the multiple threads. Your program should output the total number of times the occurrence of the key found in the array.

[HINT: Use reduction to sum-up local counts].

Question No. 4:

In this question you have to do experimentation on Matrix multiplication using **OpenMP**. The size of the N x N matrices in the program should be 100 x 100. You task is to parallelize this matrix multiplication program in different ways (course grain, fine grain) and report the execution time for each program. You can call the function omp_get_wtime() at the beginning and end of the program to find the elapsed time. The function omp_get_wtime() returns a double value.

1. At first, you will write a parallel program for matrix multiplication with default number of threads and default scheduling then run it with static and dynamic schedule for loop and report the execution times.

2. Change your program to explicitly create the number of threads equal to number of elements in the output matrix and each thread will be assigned the task to perform the calculation for one element (fine grain composition). Again, run it with default, static and dynamic schedule for routines and report the execution time for each one of them.

Question No. 5:

In this question, you have to write a multithreaded program using **OpenMP**. This task is exactly the same as you did in previous assignment but this time you have to implement it using OpenMP. You will perform Data Decomposition as well as Task Decomposition. Your program should provide the following functionality.

1. Take a matrix of size (m x n) where 'm' and 'n' values are taken as input from user. Initialize the matrix by some random values or user input.

2. Then take your student ID for the following.

- if the last digit of your student ID is (0 or 1 or 2) then create number of threads equal to number of columns 'n' and each thread will sort a column using Quick Sort in Ascending order.
- if the last digit of your student ID is (3, 4 or 5) then create number of threads equal to number of columns 'n' and each thread will sort a column using Quick Sort in Descending order.
- if the last digit of your student ID is (6 or 7) then create number of threads equal to number of rows 'm' and each thread will sort a row using Quick Sort in Ascending order.
- if the last digit of your student ID is (8 or 9) then create number of threads equal to number of rows 'm' and each thread will sort a row using Quick Sort in Descending order.

3. Quick Sort should be implemented using Recursive decomposition. Each subtask will be done by a new thread.

At the end each thread should add all values of the column/row that is being sorted and the resultant figure will be added to Matrix addition which is the sum of all values of the matrix.

National University

of Computer & Emerging Sciences-Lahore

CS3006– Parallel and Distributed Computing

Let's take the scenario of (4 x4) matrix for first batch of students.

3	5	32	80
23	11	9	15
20	7	14	26
4	17	2	7

(Input Matrix)

3	5	2	7
4	7	9	15
20	11	14	26
23	17	32	80

(Output Matrix)

Matrix Addition = 275

NOTE: Kindly submit your assignment on Google Classroom. Each file plus code file contains your roll number. Submit your assignment in Zip form with proper naming/roll no.