Assignment 1

Student: Ubaid ur Rehman
Lecturer: Sir Ali Sayyad

This assignment is about PDC codes in which parallelism is achieved.

Task 1

- The list of run-time routines in OpenMP are listed below:
  - `omp_get_thread_num()`
  - `omp_get_num_threads()`
  - `omp_get_max_threads()`
- The reason why outputs may appear 'unordered' is that the OS/runtime schedules work on threads nondeterministically.
- If you print the thread ID outside of a parallel region (or only on thread0), you will only see one ID. Within a parallel region, each thread prints its own ID.
- The following is the serial version of the code:

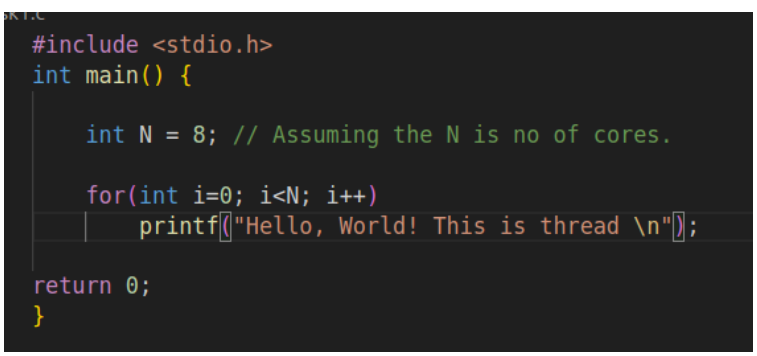

Figure 1: Serial code for iterative printing.

Task 2

- Part A

```c
#include <stdio.h>
#include<omp.h>

int main() {
    int array1[16],array2[16],result[16];            // Initialization of arrays
    int i,j,N=16;                                     // Some neccessary variables

    // Storing value 1-16 in array1

        for(i=0;i<N;i++){
            array1[i] = i+1;
        }

    // Storing values 16-1 in array2

        for(i=0;i<N;i++){
            array2[i] = N-i;
        }

    // Parallelized code for addition of two arrays
    #pragma omp parallel for
    {
        for(int i = 0; i < N; i++)
            result[i] = array1[i] + array2[i];
    }

    for (int i = 0; i < N; i++)
    {
        printf("%d ", result[i]);
    }
    printf("\n");
    return 0;
}
```

Figure 2: Parallel code for addition of arrays.

- Part B

```c
#include <stdio.h>
#include<omp.h>

int main() {
    int array1[16],array2[16],result[16];              // Initialization of arrays
    int i,N=16;                                        // Some neccessary variables

    #pragma omp parallel sections num_threads(2)       // Storing some values in arrays
    {

        // Storing value 1-16 in array1
        #pragma omp section
        {
            for(i=0;i<N;i++)
            {
                array1[i] = i+1;
            }
        }

        // Storing values 16-1 in array2
        #pragma omp section
        {
            for(i=0;i<N;i++)
            {
                array2[i] = N-i;
            }
        }
    }

    // Parallel code for addition of arrays
    #pragma omp parallel for num_threads(4)
    for(i = 0; i < N; i++)
    {
        result[i] = array1[i] + array2[i];
    }

    #pragma omp parallel num_threads(4)
    {
        // Condition for master-thread to execute this.
        if(omp_get_thread_num() == 0)
        {
            for(i=0;i<N;i++)
            {
                printf("%d ",result[i]);
            }
        }
    }
    printf("\n");
return 0;
}
```

Figure 3: Parallel code with master-thread check.

  - In this code, comments explain each section. We use ‘if (omp_get_thread_num() == 0)‘ to ensure only the master thread prints.

- The output for this code is shown below:

```
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop/subjects/PDC/Assignment/1$ gcc -fopenmp task2_b.c -o a.out
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop/subjects/PDC/Assignment/1$ ./a.out
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
```

Figure 4: Program output.

Task 3

Explanation:
In this code because the condition is to solve it using half of threads so first, I used 2 threads and create tasks below it. One task store values in array1 and same for task 2 to store in array2. Once they finish i use wait command to confirm both tasks finsish their work and then store values in result array and again before printing making sure that addition is complete before display the output.

Task 4

- Part A

task3.c

```c
#include <stdio.h>
#include<omp.h>

int main() {
    int array1[16],array2[16],result[16];              // Initialization of arrays
    int i,N=16;                                        // Some neccessary variables


    // Using only 2 threads for the whole code
    #pragma omp parallel num_threads(2)
    {
        // Single is used to execute below code using single thread
        #pragma omp single
        {
            // Now there is two tasks both are independent to eachother
            #pragma omp task
            {
                // Storing values in array1
                for(int i=0;i<N;i++)
                {
                    array1[i] = i+1;
                }
            }

            // Second task now it can be handle by seperate thread
            #pragma omp task
            {
                // Storing values in array2
                for(int i=0;i<N;i++)
                {
                    array2[i] = N-i;
                }
            }
        }

        // Barrier for tasks (to make sure they wait for others after finish)
        #pragma omp taskwait

        // Again single to create two more tasks
        #pragma omp single
        {
            // One task is for addition
```

```c
        }

        // Barrier for tasks (to make sure they wait for others after finish)
        #pragma omp taskwait

        // Again single to create two more tasks
        #pragma omp single
        {
            // One task is for addition
            #pragma omp task
            {
                for(int i=0;i<N;i++)
                {
                    result[i] = array1[i] + array2[i];
                }
            }

            // Barrier (to make sure display result after addition operation)
            #pragma omp taskwait


            // Second is for display the result
            #pragma omp task
            {
                for(int i=0;i<N;i++)
                {
                    printf("%d ", result[i]);
                }
            }
        }

    }
    printf("\nFinish");
}
```

Explaination:

This program demonstrates a simple pipeline of four dependent tasks using two OpenMP threads. Inside a ‘parallel‘ region, a single thread creates tasks to (1) fill ‘array1‘ with values 1–16, (2) fill ‘array2‘ with values 16–1, (3) sum the two arrays into ‘result‘, and (4) print the final ‘result‘. Between each stage, ‘pragma omp taskwait‘ ensures that all earlier tasks complete before the next group begins. Because only two threads are available, one thread always handles task creation and coordination while the other assists by executing queued tasks, achieving parallelism in computation and clear ordering of the pipeline.

- Part B After replacing reduction with pragma omp atomic, it will slow down the whole process in comparison with reduction because it works on hardware level and make sure only one thread will update value of a variable at same time.

```c
#include <stdio.h>
int main() {
    int array1[16],array2[16];
    int result1 = 0, result2 = 0,i,j,N=16;

    #pragma omp parallel sections num_threads(2)        // Storing some values in arrays
    {

        // Storing value 1-16 in array1
        #pragma omp section
        {
            for(i=0;i<N;i++)
            {
                array1[i] = i+1;
            }
        }

        // Storing values 16-1 in array2
        #pragma omp section
        {
            for(i=0;i<N;i++)
            {
                array2[i] = N-i;
            }
        }
    }

    // Sum array1 using atomic (it is used update variable only one thread at a time)
    #pragma omp parallel for num_threads(16)
    for(int i = 0; i < N; i++)
    {
        #pragma omp atomic
        result1 += array1[i];
    }

    // If result1 > 10, sum array2
    if(result1 > 10)
    {
        #pragma omp parallel for num_threads(16)
        for(int j = 0; j < N; j++)
        {
            #pragma omp atomic
            result2 += array2[j];
        }
    }

    printf("Total sum: %d \n",result2);
    return 0;
}
```

Figure 5: Using pragma with atomic