Assignment no 1

Name: **Muhammad Abdullah**
Roll no: **22P-9371**
Section: **BCS-6B**
Submitted to: **Dr. Ali Sayyed**

Task1:

```
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel
{ int thread_id = omp_get_thread_num();
printf("Hello, World! This is thread %d\n", thread_id);
}
return 0;
}
```

Output:

```
Hello, World! This is thread 3
Hello, World! This is thread 1
Hello, World! This is thread 2
Hello, World! This is thread 0
```

1. **Provide a list of run-time routines that are used in OpenMP.**

Solution:

omp_get_num_threads(): Returns the number of threads in the current parallel region.
omp_get_thread_num(): Returns the ID of the calling thread.
omp_set_num_threads(int num_threads): Sets the number of threads for parallel regions.
omp_get_wtime(): Returns wall-clock time in seconds.
omp_set_lock(omp_lock_t *lock): Acquires a lock for synchronization.

2. **Why aren't you seeing the Hello World output thread sequence as 0, 1, 2, 3 etc. Why are they disordered?**

Solution:
Thread outputs are disordered because the OS schedules threads non-deterministically, and printf calls from multiple threads interleave without synchronization.

3. **What happens to the thread_id if you change its scope to before the pragma?**

Solution:

```c
#include <stdio.h>
#include <omp.h>      // Include OpenMP header for parallel programming

int main()
{
   int thread_id; // Declare a variable to store thread ID (shared among all threads)

   // Start of the parallel region
   #pragma omp parallel
   {
       // Each thread will execute this block independently

       thread_id = omp_get_thread_num(); // Get the unique ID of the current thread

       printf("Hello, World! This is thread %d\n", thread_id); // Print thread-specific message
   }
   // End of parallel region

   return 0;
}
```

Moving thread_id outside makes it shared, causing a race condition where threads overwrite its value, leading to incorrect or duplicated thread IDs in output.

4. **Convert the code to serial code.**

Solution:

```c
#include <stdio.h>

int main()
{
   // Loop from 0 to 3 to simulate 4 "threads"
   for (int thread_id = 0; thread_id < 4; thread_id++)
   {
       // Print a message including the current "thread" ID
       printf("Hello, World! This is thread %d\n", thread_id);
   }

   return 0;
}
```

Output:

```
Hello, World! This is thread 0
Hello, World! This is thread 1
Hello, World! This is thread 2
Hello, World! This is thread 3
```

Task 2:

1. The following code adds two arrays of size 16 together and stores answer in result array.

```c
#include <stdio.h>
int main() {
int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ..., 16};
int array2[16] = {16, ..., 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
int result[16];
for (int i = 0; i < 16; i++)
{ result[i] = array1[i] + array2[i];
}
for (int i = 0; i < 16; i++)
{ printf("%d ", result[i]);
```

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

return 0;
}
```

Output:

```
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
```

2. Convert it into Parallel, such that only the addition part is parallelized.

Solution:

```c
#include <stdio.h>
#include <omp.h> // Include OpenMP header for parallel programming

int main()
{
   // Declare and initialize two arrays with 16 elements each
   int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16};
   int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
1};
   int result[16]; // Array to store the addition result

// Start of parallel region
#pragma omp parallel
   {
// Each thread will run part of this for loop in parallel
#pragma omp for
       for (int i = 0; i < 16; i++)
       {
           // Each thread computes one or more elements independently
           result[i] = array1[i] + array2[i];
       }
   }
   // End of parallel region
   for (int i = 0; i < 16; i++)
   {
```

```
        printf("%d ", result[i]); // Print each result element
    }
    printf("\n"); // New line after all numbers are printed

    return 0;
}
```

Output:

```
 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
```

3. The display loop at the end displays the result. Modify the code such that this is also parallel, but only thread of id 0 is able to display the entire loop. The others should not do anything. When making it parallel, make sure its the old threads and new threads are not created. What output do you see?

Solution:

```
#include <stdio.h>
#include <omp.h>
int main() {
   // Initialize two arrays of size 16 for addition
   int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
   int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
   // Array to store the sum of array1 and array2
   int result[16];

   // Start a parallel region, creating a team of threads
   #pragma omp parallel
   {
       // Parallelize the addition loop: distribute iterations across threads
       #pragma omp for
       for (int i = 0; i < 16; i++) {
           // Each thread computes a portion of the result array
           result[i] = array1[i] + array2[i];
       }
```

```
        // Implicit barrier here: all threads wait until the addition is
complete

        // Display loop: only thread 0 prints the result
        if (omp_get_thread_num() == 0) {
            // Thread 0 prints the entire result array
            for (int i = 0; i < 16; i++) {
                printf("%d ", result[i]);
            }
            printf("\n");
        }
        // Other threads skip the printing loop and do nothing
    } // End of parallel region

    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_assignment#1$ ./task2_3
 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
```

Task 3

1. Modify the code of Task 2 and do the job in half of the threads.

Solution:

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

int main() {
    // Initialize arrays
    int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16};
    int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
1};
    int result[16];

    // Get the default number of threads and set to half
```

```
   int default_threads = omp_get_max_threads(); // Default number of
threads
   int half_threads = (default_threads + 1) / 2; // Half of default,
rounded up if odd
   omp_set_num_threads(half_threads); // Set the number of threads to half

   // Print the number of threads being used for clarity
   printf("Using %d threads (half of default %d)\n", half_threads,
default_threads);

   // Single parallel region for both addition and display
   #pragma omp parallel
   {
       // Parallelize the addition loop
       #pragma omp for
       for (int i = 0; i < 16; i++) {
           result[i] = array1[i] + array2[i];
       }
       // Implicit barrier ensures all threads finish addition

       // Display loop in parallel region, only thread 0 prints
       if (omp_get_thread_num() == 0) {
           for (int i = 0; i < 16; i++) {
               printf("%d ", result[i]);
           }
           printf("\n");
       }
       // Other threads skip the printing loop
   } // End of parallel region

   return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_assignment#1$ ./task3
Using 2 threads (half of default 4)
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
```

Task 4

2. Convert it into Parallel using 16 threads.

Solution:

```
#include <stdio.h>
#include <omp.h>      // Include OpenMP header for parallel programming

int main() {
   // Define two arrays of size 16 and initialize them with values
   int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16};
   int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
1};

   // Initialize two result variables to 0
   int result1 = 0, result2 = 0;

   // Create a single parallel region with 16 threads
   #pragma omp parallel num_threads(16)
   {
       // First parallelized for loop to compute sum of array1
       #pragma omp for reduction(+:result1)
       for (int i = 0; i < 16; i++) {
           result1 += array1[i];
       }

       // Only one thread should check and handle the if condition
       #pragma omp single
       {
           if (result1 > 10) {
               result2 = result1;
           }
       }

       // Second parallelized for loop to add elements of array2 if
condition was true
       #pragma omp for reduction(+:result2)
       for (int i = 0; i < 16; i++) {
           if (result1 > 10) { // Check again inside parallel loop
```

```
                result2 += array2[i];
            }
        }
    }
    // Parallel region ends here

    // Print the final result2 value
    printf("%d\n", result2);

    return 0;
}
```

Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_assignment#1$ ./task4
272
```

3. Try removing the reduction() clause and add #pragma omp atomic just beore the +=. What is the effect on result? Explain.

Solution:

```
#include <stdio.h>
#include <omp.h>        // Include OpenMP library for parallel programming

int main() {
    // Initialize two arrays with 16 elements each
    int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16};
    int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
1};

    // Initialize result variables to 0
    int result1 = 0, result2 = 0;

    // Start a parallel for loop with 16 threads
    #pragma omp parallel num_threads(16)
    {
```

```
        // First parallelized for loop to compute sum of array1
        #pragma omp for
        for (int i = 0; i < 16; i++) {
            #pragma omp atomic
            result1 += array1[i];  // Ensure safe addition of array1[i] to
result1
        }
        // Only one thread should check and handle the if condition
        #pragma omp single
        {
            if (result1 > 10) {
                result2 = result1;  // Copy result1 to result2 if condition
is met
            }
        }

        // Second parallelized for loop to add elements of array2 if
result1 > 10
        #pragma omp for
        for (int i = 0; i < 16; i++) {
            if (result1 > 10) {  // Only process if result1 is greater than
10
                #pragma omp atomic
                result2 += array2[i];  // Ensure safe addition of array2[i]
to result2
            }
        }
    }

    // Print the final result2 value
    printf("%d\n", result2);

    return 0;
}
```
Output:

```
muhammadabdullah@muhammad-abdullah:~/Documents/pdc_assignment#1$ ./task4_1
272
```

Explanation:

When you **remove reduction(+:result)** and instead **use #pragma omp atomic** before result += ...,
the result is still **correct**, because #pragma omp atomic ensures that the updates to result happen safely, **one at a time**.

Effect on result:

- The final **numerical result stays the same** (correct sum).
- No wrong answers or race conditions occur.

But performance becomes slower.

- **Reason**: In atomic mode, each thread **waits its turn** to update result, causing delays.
- In reduction mode, each thread works **independently** and combines results at the end, so it's faster.