



Name: Tazmeen Afroz

Roll No: 22P-9252

Section: BAI-6A

Instructor: Sir Ali Sayyed

Course: Parallel and Distributed Computing

Assignment: 1

Campus: FAST-NUCES Peshawar

Date: 27 April 2025

Task 1

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

- `omp_get_thread_num()` - Returns the thread ID
- `omp_get_num_threads()` - Returns the number of threads in current team
- `omp_set_num_threads()` - Sets the number of threads for parallel regions
- `omp_get_num_procs()` - Determines the number of processors available to the program
- `omp_get_thread_limit()` - Returns the maximum number of OpenMP threads available to participate in the current contention group

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

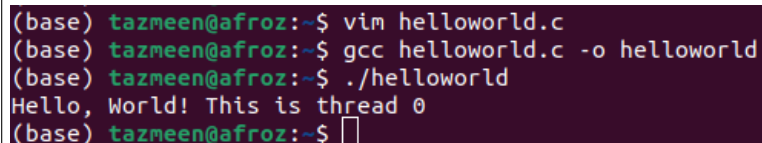
The thread sequence is disordered because the code is non-deterministic. Each thread executes independently with no synchronization, and the operating system scheduler decides which thread runs when, resulting in unpredictable output order on each run.

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

When `thread_id` is declared outside the parallel region, it becomes a shared variable among all threads, creating a race condition where threads compete to write to the same memory location, potentially causing unpredictable results.

4. Convert the code to serial code.

```
1 #include <stdio.h>
2
3 int main() {
4     int thread_id = 0; // In serial code, there's only thread 0
5     printf("Hello, World! This is thread %d\n", thread_id);
6     return 0;
7 }
```

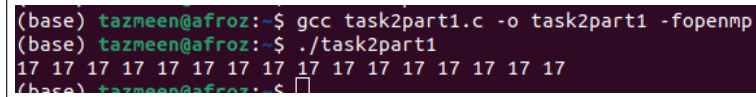


```
(base) tazmeen@afroz:~$ vim helloworld.c
(base) tazmeen@afroz:~$ gcc helloworld.c -o helloworld
(base) tazmeen@afroz:~$ ./helloworld
Hello, World! This is thread 0
(base) tazmeen@afroz:~$
```

Figure 1: Output of serial code

Task 2

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

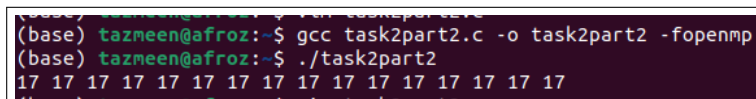


```
(base) tazmeen@afroz:~$ gcc task2part1.c -o task2part1 -fopenmp
(base) tazmeen@afroz:~$ ./task2part1
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
(base) tazmeen@afroz:~$
```

Figure 2: Original array addition code

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

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
6     16};
7     int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
8     1};
9     int result[16];
10
11     // This directive combines thread creation and loop distribution.
12     // It's a cleaner syntax when you only need to parallelize a single
13     // loop.
14     // Since we are only parallelizing the addition loop, we use this
15     // combined directive.
16
17     #pragma omp parallel for
18     for (int i = 0; i < 16; i++) {
19         result[i] = array1[i] + array2[i];
20     }
21
22     // This loop runs serially after all threads complete the addition.
23     for (int i = 0; i < 16; i++) {
24         printf("%d ", result[i]);
25     }
26     printf("\n");
27
28     return 0;
29 }
```



```
(base) tazmeen@afroz:~$ gcc task2part2.c -o task2part2 -fopenmp
(base) tazmeen@afroz:~$ ./task2part2
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
(base) tazmeen@afroz:~$
```

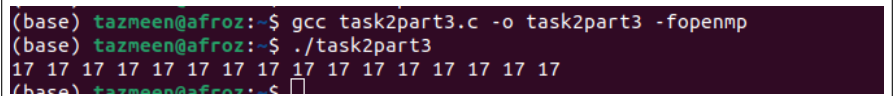
Figure 3: Output of parallel addition code

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.

```

1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
6     16};
7     int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
8     1};
9     int result[16];
10
11     // This creates a team of threads that will stay active for the whole
12     block
13
14     #pragma omp parallel
15     {
16         // This distributes the loop iterations among the existing threads
17         // Each thread gets a chunk of the array to process
18         #pragma omp for
19         for (int i = 0; i < 16; i++) {
20             result[i] = array1[i] + array2[i];
21         }
22
23         // Every thread reaches this point, but we only want thread 0 to
24         print
25
26         if (omp_get_thread_num() == 0) {
27             // Only the thread with ID 0 executes this code
28             for (int i = 0; i < 16; i++) {
29                 printf("%d ", result[i]);
30             }
31             printf("\n");
32         }
33     }
34
35     return 0;
36 }

```



```

(base) tazmeen@afroz:~$ gcc task2part3.c -o task2part3 -fopenmp
(base) tazmeen@afroz:~$ ./task2part3
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
(base) tazmeen@afroz:~$

```

Figure 4: Output with thread 0 displaying results

Explanation: In both cases, each element position ($\text{array1}[i] + \text{array2}[i]$) equals 17, so the output is sixteen 17s. The key difference is not in the calculation but in who does the printing. The first version has parallel addition but serial printing. The second version has parallel addition and only thread 0 handles all printing.

Task 3

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

Explanation:

Solution 1: Uses `omp_get_max_threads()` to determine the maximum available threads and sets only half that number before execution, creating exactly the threads needed for computation.

Solution 2: Creates all available threads but uses `omp_get_num_threads()` at runtime to determine which threads should work, allowing only half to perform calculations while the others remain idle.

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
6     16};
7     int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
8     1};
9     int result[16];
10
11     // Get max threads and use half
12     int max_threads = omp_get_max_threads();
13     omp_set_num_threads(max_threads / 2);
14
15     // Parallel for with half threads
16     #pragma omp parallel for
17     for (int i = 0; i < 16; i++) {
18         result[i] = array1[i] + array2[i];
19     }
20
21     // Print results
22     for (int i = 0; i < 16; i++) {
23         printf("%d ", result[i]);
24     }
25     printf("\n");
26
27     return 0;
28 }
```

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
6     16};
7     int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
8     1};
9     int result[16];
10
11     #pragma omp parallel
12     {
13         // Only use half the threads based on thread ID
```

```

12     int thread_id = omp_get_thread_num();
13     int total_threads = omp_get_num_threads();
14     int active_threads = total_threads / 2;
15
16     if (thread_id < active_threads) {
17         for (int i = thread_id; i < 16; i += active_threads) {
18             result[i] = array1[i] + array2[i];
19         }
20     }
21 }
22
23 // Print results
24 for (int i = 0; i < 16; i++) {
25     printf("%d ", result[i]);
26 }
27 printf("\n");
28
29 return 0;
30 }

```

```

(base) tazmeen@afroz:~$ gcc task3.c -o task3 -fopenmp
(base) tazmeen@afroz:~$ ./task3
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
(base) tazmeen@afroz:~$

```

Figure 5: Output using half the number of threads

Task 4

2. Convert it into Parallel using 16 threads. 3. Try removing the reduction() clause and add #pragma omp atomic

Version 1: Using reduction clause

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
6         16};
7     int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
8         1};
9     int result1 = 0, result2 = 0;
10
11     // Version 1: Using reduction clause
12     #pragma omp parallel num_threads(16)
13     {
14         // First loop with reduction
15         // here result1 is private to each thread
16         // and will be combined at the end of the loop
17
18         #pragma omp for reduction(+:result1)
19         for (int i = 0; i < 16; i++) {
20             result1 += array1[i];
21         }
22
23         // Barrier here is implicit at the end of the for construct
24
25         // Only proceed with second loop if condition is met
26
27         #pragma omp single
28         {
29             if (result1 > 10) {
30                 result2 = result1;
31             }
32         }
33
34         // Second loop with reduction if condition was met
35         // here result2 is private to each thread
36         // and will be combined at the end of the loop
37         if (result1 > 10) {
38             #pragma omp for reduction(+:result2)
39             for (int i = 0; i < 16; i++) {
40                 result2 += array2[i];
41             }
42         }
43     }
44
45     printf("Using reduction: %d\n", result2);
46 }
```

```

45
46     return 0;
47 }

```

Version 2: Using atomic operations

```

1  #include <stdio.h>
2  #include <omp.h>
3
4  int main() {
5      int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
6                          16};
7      int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,
8                          1};
9      int result1 = 0, result2 = 0;
10
11     // Version 2: Using atomic operations
12     #pragma omp parallel num_threads(16)
13     {
14         // First loop with atomic
15         // atomic operations are used to ensure that
16         // the updates to result1 are done safely
17         // it is different from reduction as it does not
18         // create a private copy of result1 for each thread
19         // but rather updates the shared variable directly
20         // this can be less efficient than reduction
21         #pragma omp for
22         for (int i = 0; i < 16; i++) {
23             #pragma omp atomic
24             result1 += array1[i];
25         }
26
27         // Barrier here is implicit at the end of the for construct
28
29         // Only proceed with second loop if condition is met
30         #pragma omp single
31         {
32             if (result1 > 10) {
33                 result2 = result1;
34             }
35         }
36
37         // Second loop with atomic if condition was met
38         if (result1 > 10) {
39             #pragma omp for
40             for (int i = 0; i < 16; i++) {
41                 #pragma omp atomic
42                 result2 += array2[i];
43             }
44         }
45     }
46
47     printf("Using atomic: %d\n", result2);

```



```
46  
47     return 0;  
48 }
```

```
(base) tazmeen@afroz:~$ gcc task4.c -o task4 -fopenmp
(base) tazmeen@afroz:~$ ./task4
Using reduction: 272
(base) tazmeen@afroz:~$ gcc task4b.c -o task4b -fopenmp
(base) tazmeen@afroz:~$ ./task4b
Using atomic: 272
(base) tazmeen@afroz:~$
```

Figure 6: Reduction vs Atomic operations output

Explanation of the effect on result:

Atomic Operations:

- Each update to result1 or result2 requires locking/unlocking the memory location
- Only one thread can update the variable at a time, creating a bottleneck
- Other threads must wait their turn, reducing parallelism
- Creates high memory contention and cache coherence traffic

Reduction:

- Each thread maintains a private copy of the reduction variable
- Threads perform additions to their private copies without interference
- At the end of the parallel region, the private copies are combined efficiently
- Much less contention and better scalability

[illegible]

Figure 7: Outputs of All codes