

Class: Final Year B.Tech(Computer Science and Engineering)

Year: 2025-26 Semester: 1

Course: High Performance Computing Lab

## Practical No. 2

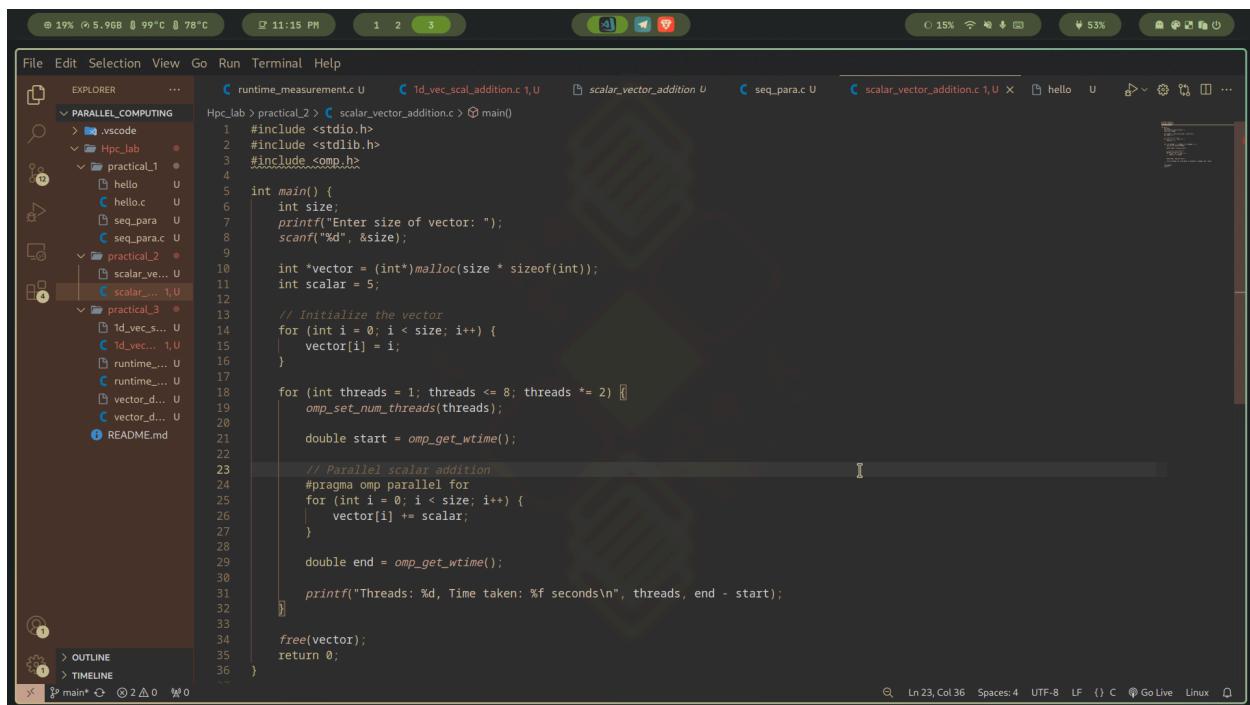
Exam Seat No: 23520012

Title of practical: Study and implementation of basic OpenMP clauses

### Problem Statement 1:

Implement following Programs using OpenMP with C:

#### Vector Scalar Addition:



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a project structure under "PARALLEL COMPUTING". The "practical\_3" folder is expanded, showing files: scalar\_vector\_addition.c, 1d\_vec\_scal\_addition.c, seq\_para.c, runtime\_measurement.c, 1d\_vec\_s... (truncated), scalar\_ve... (truncated), and README.md.
- Code Editor:** Displays the content of `scalar_vector_addition.c`. The code implements a parallel scalar addition using OpenMP threads.
- Status Bar:** Shows system information like battery level (19%), temperature (5.96B, 99°C, 78°C), time (11:15 PM), and disk usage (15%, 53%).
- Bottom Bar:** Includes icons for search, file operations, and terminal.

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

int main() {
    int size;
    printf("Enter size of vector: ");
    scanf("%d", &size);

    int *vector = (int*)malloc(size * sizeof(int));
    int scalar = 5;

    // Initialize the vector
    for (int i = 0; i < size; i++) {
        vector[i] = i;
    }

    for (int threads = 1; threads <= 8; threads *= 2) {
        omp_set_num_threads(threads);

        double start = omp_get_wtime();

        // Parallel scalar addition
        #pragma omp parallel for
        for (int i = 0; i < size; i++) {
            vector[i] *= scalar;
        }

        double end = omp_get_wtime();
        printf("Threads: %d, Time taken: %f seconds\n", threads, end - start);
    }

    free(vector);
    return 0;
}
```

The screenshot shows a terminal window within the VS Code interface. The terminal output displays the execution of a C program named `scalar_vector_addition.c`. The user enters the size of the vector (4) and the program prints the execution time for different thread counts: 1 thread (0.000011 seconds), 2 threads (0.000092 seconds), 4 threads (0.000063 seconds), and 8 threads (0.008345 seconds). The terminal also shows the command to compile the code using `gcc -fopenmp`.

```

Hpc_lab > practical_2 > scalar_vector_addition.c > main()
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main() {
6     int size;
7     printf("Enter size of vector: ");
8     scanf("%d", &size);
9
10    int *vector = (int*)malloc(size * sizeof(int));
11    int scalar = 5;
12
13    #pragma omp parallel for
14    for (int i = 0; i < size; i++) {
15        vector[i] += scalar;
16    }
17
18    for (int i = 0; i < size; i++) {
19        if (vector[i] != i + 5) {
20            printf("Error: vector[%d] = %d, expected %d\n", i, vector[i], i + 5);
21            return 1;
22        }
23    }
24
25    free(vector);
26    return 0;
27}

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS
NOWAIT used. Time: 0.018885 seconds

● > cd ..
● > cd practical_2
● > gcc -fopenmp scalar_vector_addition.c -o scalar_vector_addition
● > ./scalar_vector_addition
Enter size of vector: 4
Threads: 1, Time taken: 0.000011 seconds
Threads: 2, Time taken: 0.000092 seconds
Threads: 4, Time taken: 0.000063 seconds
Threads: 8, Time taken: 0.008345 seconds

● > ./scalar_vector_addition
Enter size of vector: 4
Threads: 1, Time taken: 0.000011 seconds
Threads: 2, Time taken: 0.000092 seconds
Threads: 4, Time taken: 0.000063 seconds
Threads: 8, Time taken: 0.008345 seconds

```

Threads	Time (seconds)
1	0.000011
2	0.000092
4	0.000063
8	0.008345

## Why performance got worse with more threads?

Here's a **simple explanation**:

- For **very small tasks**, using many threads **adds more overhead** than benefit.
- Thread creation, context switching, and managing synchronization takes time.
- That overhead **outweighs** the tiny job of adding a scalar to 4 elements.

## What this teaches us:

- **Parallelism only helps when the workload is large enough** to divide meaningfully among threads.
- For small data (like 4 elements), it's **better to use fewer threads or just one**.
- For large vectors (e.g., 100,000+ elements), you'll see much better **scaling and speedup** with more threads.

The screenshot shows a terminal window in VS Code with the following output:

```
Threads: 4, Time taken: 0.000063 seconds
Threads: 8, Time taken: 0.008345 seconds
● > ./scalar_vector_addition
Enter size of vector: 1000
Threads: 1, Time taken: 0.000012 seconds
Threads: 2, Time taken: 0.000095 seconds
Threads: 4, Time taken: 0.000084 seconds
Threads: 8, Time taken: 0.030078 seconds

● > ./scalar_vector_addition
Enter size of vector: 10000
Threads: 1, Time taken: 0.000322 seconds
Threads: 2, Time taken: 0.000258 seconds
Threads: 4, Time taken: 0.000200 seconds
Threads: 8, Time taken: 0.006167 seconds
```

Threads	Time (seconds)
1	0.000012
2	0.000095
4	0.000084

8	0.030078
---	----------

Threads	Time (seconds)
1	0.000322
2	0.000258
4	0.000200
8	0.006167

#### **Analysis:**

**Up to 4 threads**, the program is becoming **faster**—because the CPU divides work nicely.

**At 8 threads**, performance **drops a lot**—too much overhead, not enough work for each thread.

For real speedup using OpenMP, you need:

- **Larger vectors** (e.g., 1 million elements or more)
- **Optimal number of threads** (not too many)

#### **Calculation of value of Pi:**

Analyse the performance of your programs for different number of threads and Data size.

```

File Edit Selection View Go Run Terminal Help
.c U C runtime_measurement.c U C 1d_vec_scal_addition.c U scalar_vector_addition U C seq_para.c U C scalar_vector_addition.c U C pi_calculation.c 1,U X C hello U D> V S E I ...
Hpc_lab > practical_2 > C pi_calculation.c > main()
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     long num_steps = 100000000; // More steps = more accurate
6     double step = 1.0 / (double)num_steps;
7     double pi = 0.0;
8     int num_threads;
9
10    printf("Enter number of threads: ");
11    scanf("%d", &num_threads);
12
13    omp_set_num_threads(num_threads);
14    double start_time = omp_get_wtime();
15
16    #pragma omp parallel
17    {
18        int id = omp_get_thread_num();
19        int nthreads = omp_get_num_threads();
20        double x, sum = 0.0;
21
22        for (long i = id; i < num_steps; i += nthreads) {
23            x = (i + 0.5) * step;
24            sum += 4.0 / (1.0 + x * x);
25
26        #pragma omp atomic
27        pi += sum;
28    }
29
30    pi *= step;
31
32    double end_time = omp_get_wtime();
33    printf("Approximate value of pi: %.15f\n", pi);
34    printf("Time taken with %d thread(s): %f seconds\n", num_threads, end_time - start_time);
35    return 0;
36

```

Ln 34, Col 94 Spaces:4 UTF-8 LF () C Go Live Linux

```

File Edit Selection View Go Run Terminal Help
.c U C runtime_measurement.c U C 1d_vec_scal_addition.c U scalar_vector.addition U C seq_para.c U C scalar_vector.addition.c U C pi_calculation.c 1,U X C hello U D> V S E I ...
Hpc_lab > practical_2 > C pi_calculation.c > main()
1 #include <stdio.h>
2
3 PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS bash-practical_2 + ...
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

```

Ln 34, Col 94 Spaces:4 UTF-8 LF () C Go Live Linux

Threads	Time Taken (s)	Approx. $\pi$
1	0.376596	3.141592653590426

4	0.186257	3.141592653590217
8	0.130433	3.141592653589614
10	0.131590	3.141592653589745

Threads	Time (s)	Speedup
1	0.376596	1.00
4	0.186257	2.02 ×
8	0.130433	2.89 ×
10	0.131590	2.86 ×

At 4 threads, the speedup is  $\sim 2 \times$  (not  $4 \times$ ) due to:

- Overheads of thread management
- Uneven load distribution
- Shared memory access

At 8–10 threads, you observe a **plateau in performance**:

- Likely due to **hyper-threading**, or
- You have fewer physical cores (say, 4 or 6), and more threads are not giving additional benefit.

**Conclusion:**

- OpenMP successfully parallelizes the work.
- Speedup improves with more threads up to a point, then levels off.
- You can visualize this with a plot if needed.

Github Link: [https://github.com/bhaktimore18/hpc\\_lab](https://github.com/bhaktimore18/hpc_lab)