Class: Final Year B.Tech(Computer Science and Engineering)
Year: 2025-26 Semester: 1
Course: High Performance Computing Lab

Practical No. 2

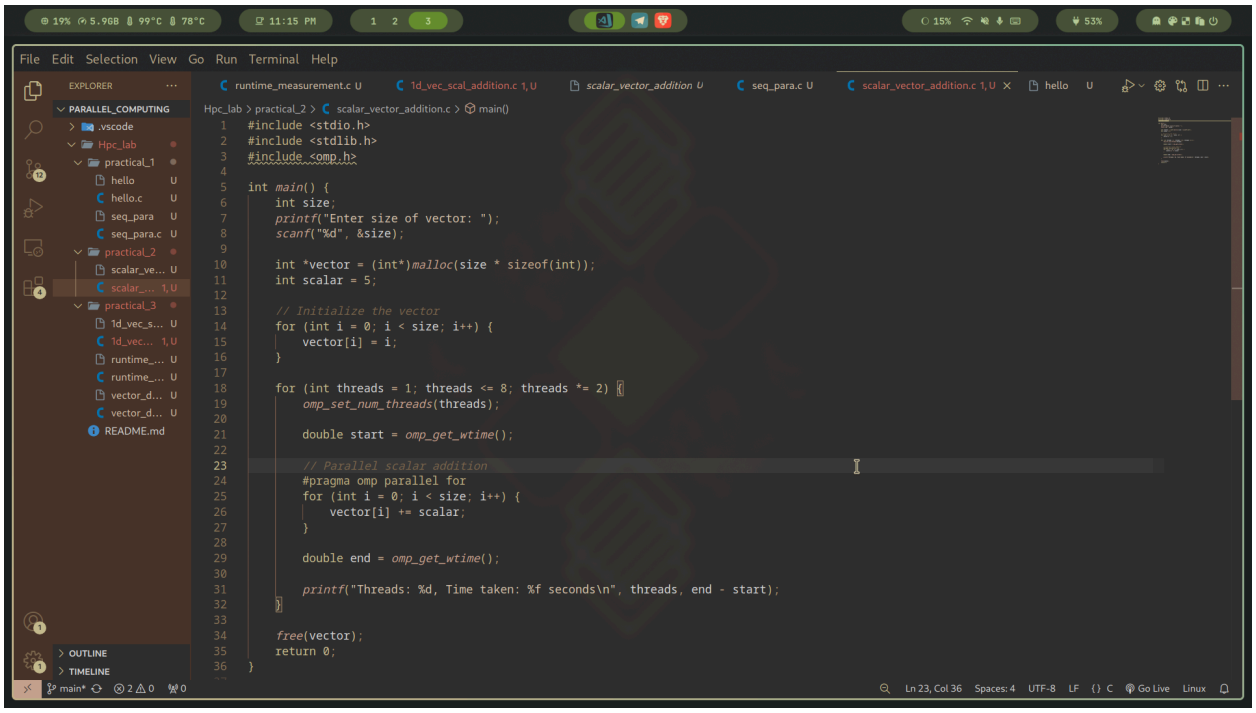
Exam Seat No: 23520012

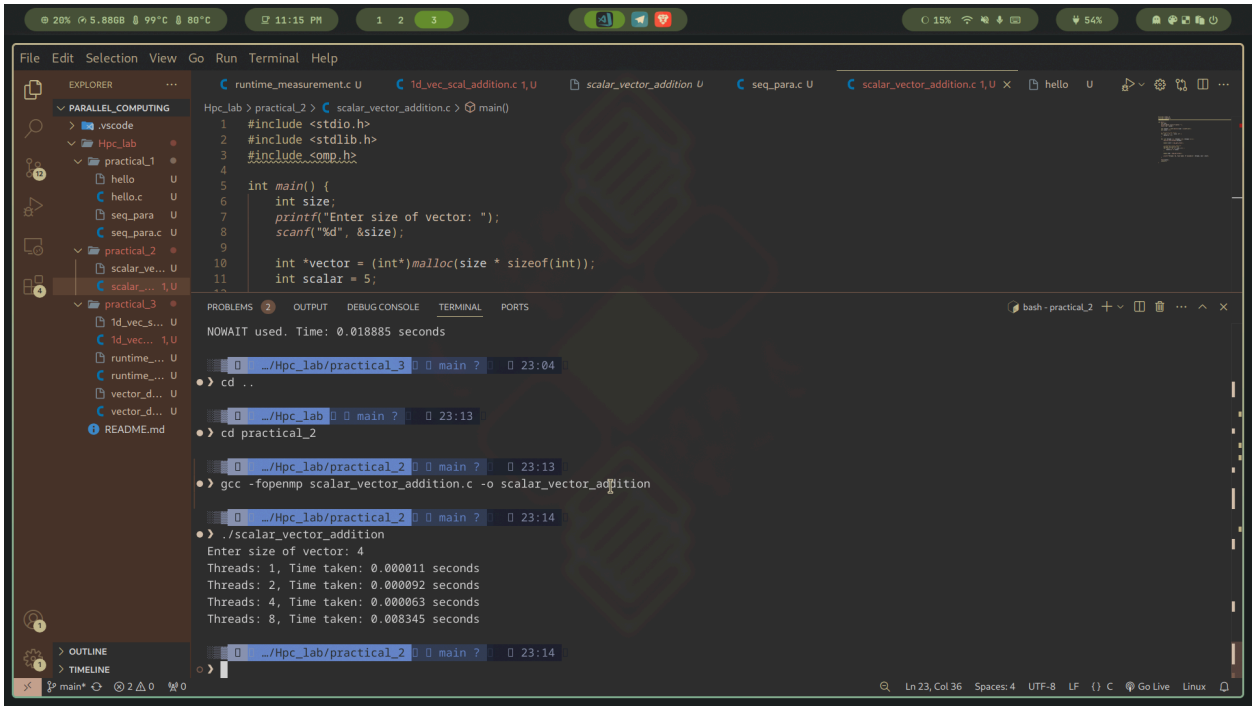
Title of practical: Study and implementation of basic OpenMP clauses

Problem Statement 1:

Implement following Programs using OpenMP with C:

Vector Scalar Addition:





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

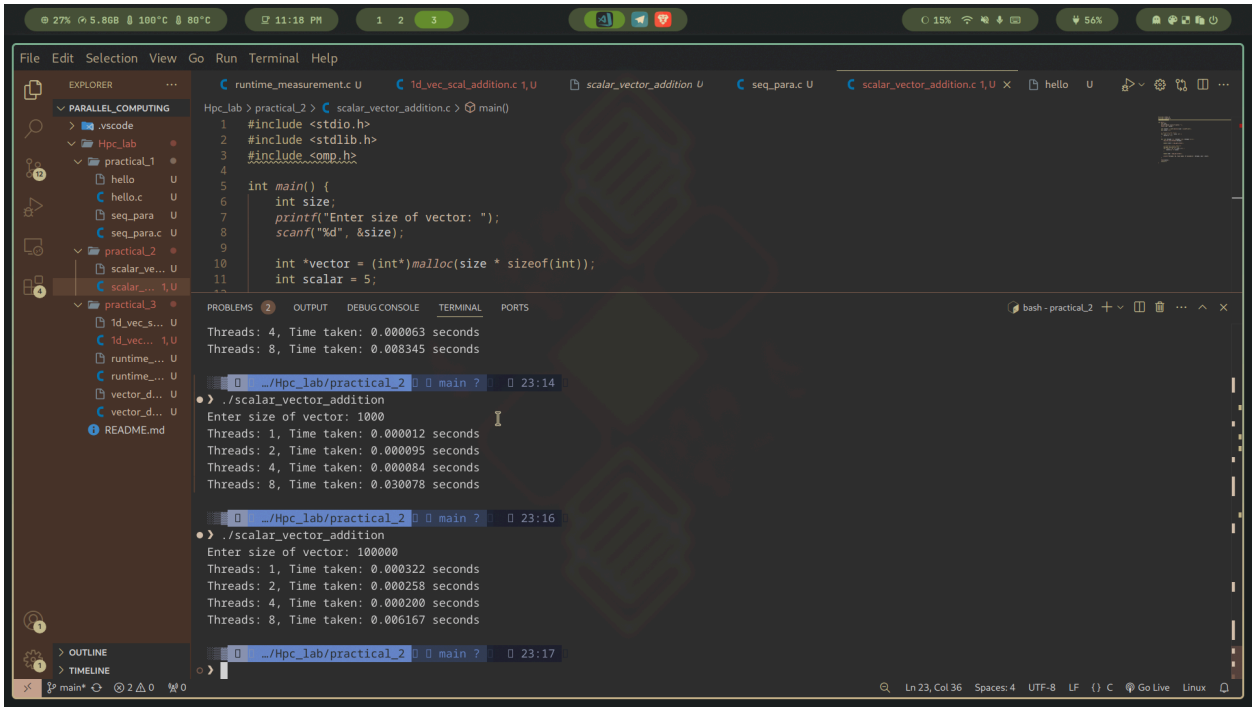
Why performance got worse with more threads?

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.



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

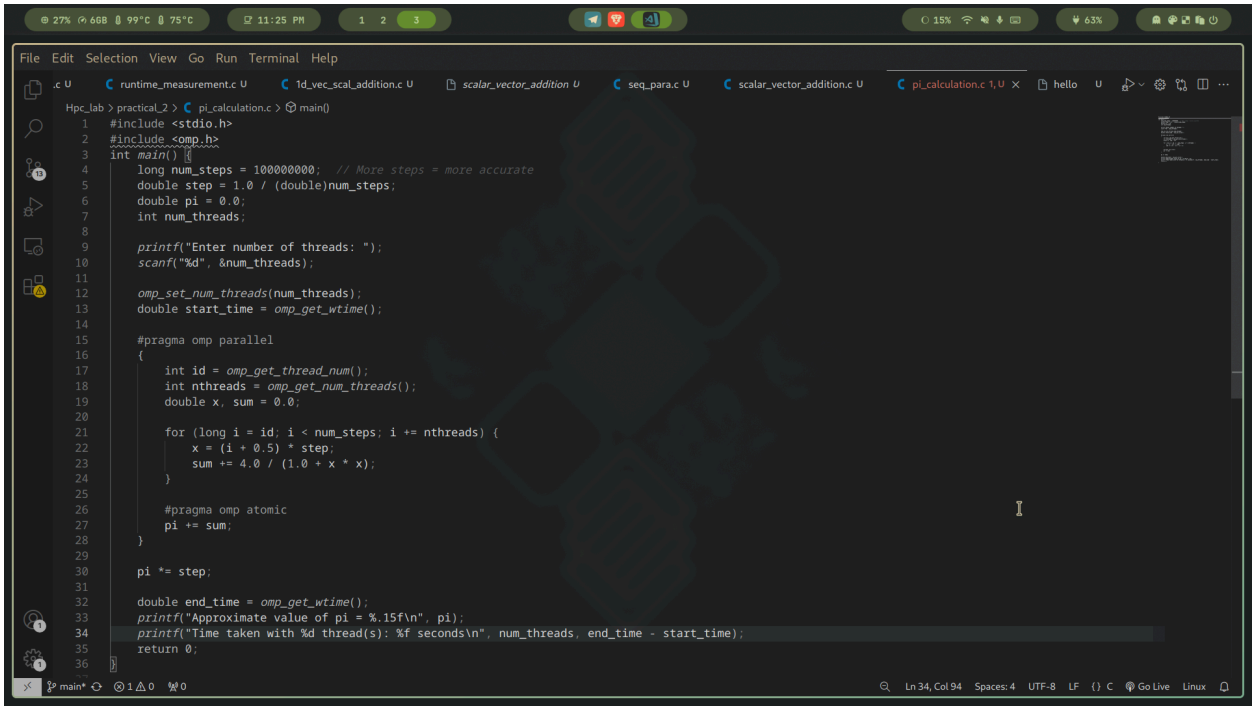
Analysis:
Up to 4 threads, the program is becoming **faster**—because the CPU divides work nicely.

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

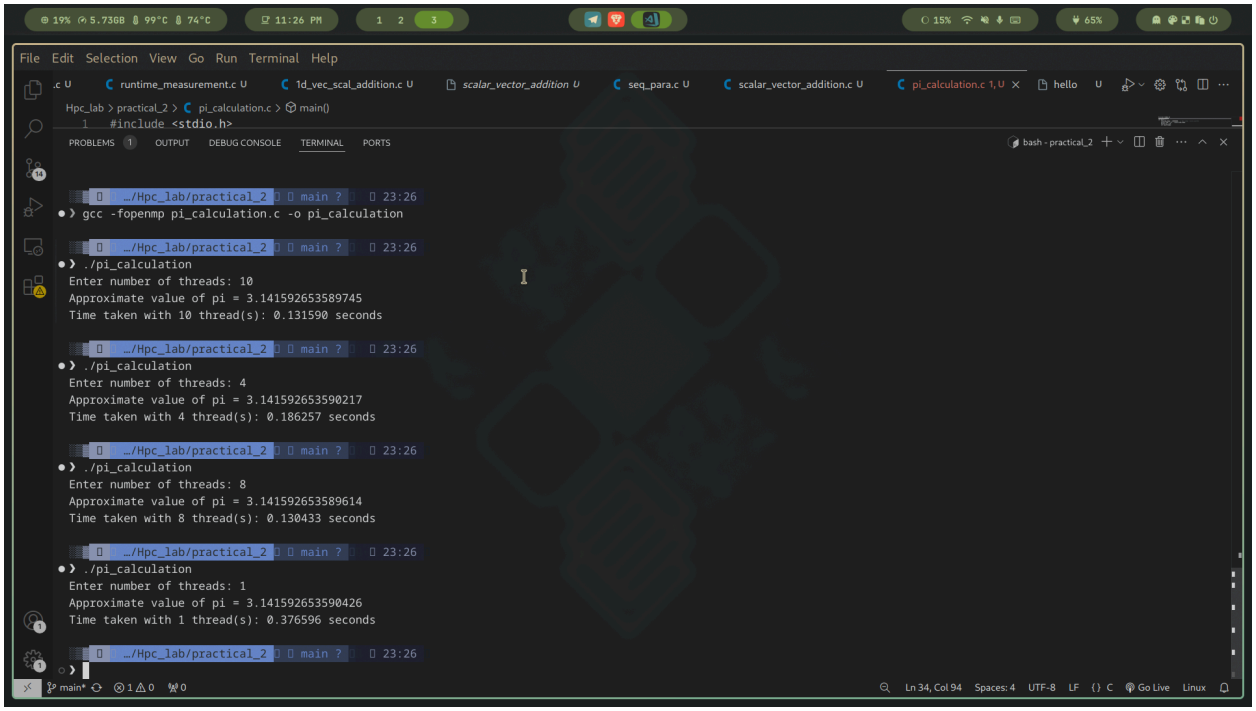
For real speedup using OpenMP, you need:

- **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.



```c
#include <stdio.h>
#include <omp.h>
int main() {
    long num_steps = 100000000;  // More steps = more accurate
    double step = 1.0 / (double)num_steps;
    double pi = 0.0;
    int num_threads;

    printf("Enter number of threads: ");
    scanf("%d", &num_threads);

    omp_set_num_threads(num_threads);
    double start_time = omp_get_wtime();

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        double x, sum = 0.0;

        for (long i = id; i < num_steps; i += nthreads) {
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }

        #pragma omp atomic
        pi += sum;
    }

    pi *= step;

    double end_time = omp_get_wtime();
    printf("Approximate value of pi = %.15f\n", pi);
    printf("Time taken with %d thread(s): %f seconds\n", num_threads, end_time - start_time);
    return 0;
}
```



```
…/Hpc_lab/practical_2  main ?  23:26
❯ gcc -fopenmp pi_calculation.c -o pi_calculation

…/Hpc_lab/practical_2  main ?  23:26
❯ ./pi_calculation
Enter number of threads: 10
Approximate value of pi = 3.141592653589745
Time taken with 10 thread(s): 0.131590 seconds

…/Hpc_lab/practical_2  main ?  23:26
❯ ./pi_calculation
Enter number of threads: 4
Approximate value of pi = 3.141592653590217
Time taken with 4 thread(s): 0.186257 seconds

…/Hpc_lab/practical_2  main ?  23:26
❯ ./pi_calculation
Enter number of threads: 8
Approximate value of pi = 3.141592653589614
Time taken with 8 thread(s): 0.130433 seconds

…/Hpc_lab/practical_2  main ?  23:26
❯ ./pi_calculation
Enter number of threads: 1
Approximate value of pi = 3.141592653590426
Time taken with 1 thread(s): 0.376596 seconds

…/Hpc_lab/practical_2  main ?  23:26
❯
```

Threads	Time Taken (s)	Approx. π
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 ~2× (not 4×) 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