Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

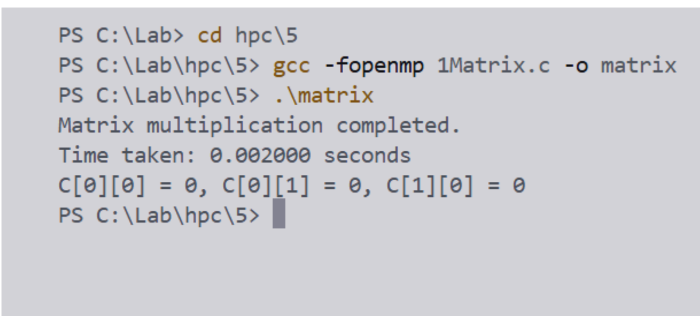
**Practical No. 5**

**Exam Seat No: 22510021**

**Title of practical: Implementation of OpenMP programs.**

Implement following Programs using OpenMP with C:

**Problem Statement 1:** Implementation of Matrix-Matrix Multiplication.
**Screenshots:**



```
PS C:\Lab> cd hpc\5
PS C:\Lab\hpc\5> gcc -fopenmp 1Matrix.c -o matrix
PS C:\Lab\hpc\5> .\matrix
Matrix multiplication completed.
Time taken: 0.002000 seconds
C[0][0] = 0, C[0][1] = 0, C[1][0] = 0
PS C:\Lab\hpc\5>
```

**Information:**

**Program Steps:**

1. Declare three N×N matrices: A, B, and C.
2. Initialize A[i][j] = i, B[i][j] = j, and C[i][j] = 0.
3. Set number of threads using omp_set_num_threads(4).
4. Use OpenMP parallel for with collapse(2) to compute each element of C[i][j] in parallel.
5. Measure start and end time using omp_get_wtime().
6. Print the total time taken and a few sample elements of C to verify correctness.

**Analysis:**

- Threads compute complete elements independently → no race conditions.
- Example (2 threads, 4×4 matrix):
  - Thread 0: C[0][0], C[0][2], C[1][0], C[1][2]

  - Thread 1: C[0][1], C[0][3], C[1][1], C[1][3]
- Parallel execution reduces time significantly.
- Small matrices may show less improvement due to thread overhead.
- Increasing threads improves speedup up to the number of CPU cores.

**Problem Statement 2:** Implementation of Matrix-scalar Multiplication.
**Screenshots:**

```
PS C:\Lab\hpc\5> gcc -fopenmp 2scaler.c -o scaler
PS C:\Lab\hpc\5> .\scaler
PS C:\Lab\hpc\5> gcc -fopenmp 2scaler.c -o scaler
PS C:\Lab\hpc\5> .\scaler
Scalar multiplication completed.
Time taken: 0.012000 seconds
C[0][0] = 0, C[0][1] = 0, C[1][0] = 5
PS C:\Lab\hpc\5>
```

**Information:**

- Scalar Multiplication Formula: C[i][j]=scalar×A[i][j]
- OpenMP Parallelization:
  - #pragma omp parallel for collapse(2) → Distributes the row-column iterations among multiple threads.
  - omp_set_num_threads() → Sets the number of threads globally.

- omp_get_wtime() → Measures the wall-clock time for execution.

**Program Steps:**

1. Declare matrices A and C of size N×N.
2. Initialize matrix A with values.
3. Set the number of threads using omp_set_num_threads().
4. Multiply each element by the scalar using OpenMP parallel loops.
5. Measure and print the time taken and a few elements of the result matrix.

**Analysis:**
Time Complexity:

- Sequential: O(N2)O(N^2)O(N2)

- Parallel: The i and j loops are split among threads, so effective time decreases roughly by the number of threads.

Thread Work Distribution:

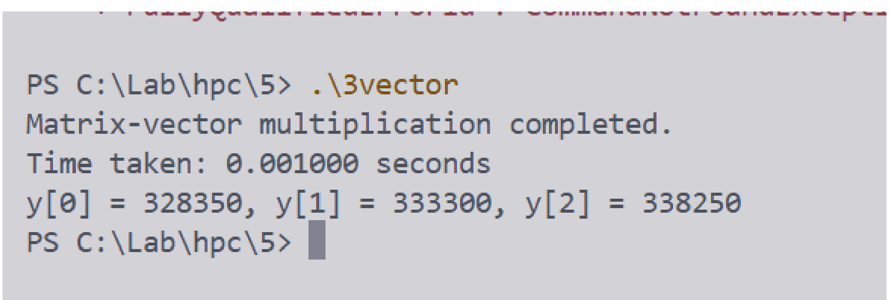
- Using collapse 2 threads are assigned blocks of i and j pairs.
- Each thread multiplies its assigned elements independently → no race conditions.

Observations:

- Parallel execution significantly reduces computation time, especially for large matrices.
- Small matrices may show minor improvement due to thread overhead.
- Increasing threads improves speedup up to the number of CPU cores.

**Problem Statement 3:** Implementation of Matrix-Vector Multiplication.

**Screenshots:**




**Information:**

**Program Steps:**

1. Declare a matrix A[N][N], input vector x[N], and result vector y[N].
2. Initialize matrix and vector with values.
3. Set the number of threads using omp_set_num_threads().
4. Parallelize the computation of each y[i] using #pragma omp parallel for.
5. Compute each element as the dot product of a row of A and vector x.
6. Measure execution time using omp_get_wtime() and print sample elements of y.

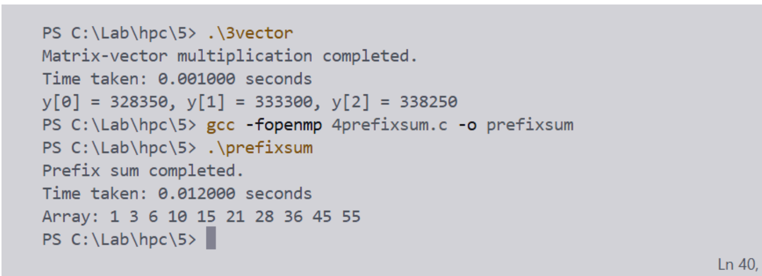
**Analysis:**

- $O(N^2)$

- Parallel: Each row computation is independent, so rows are distributed among threads for faster execution.
- Each thread computes a subset of rows completely.
- No race conditions occur because each thread writes to its own element of y.

**Problem Statement 4:** Implementation of Prefix sum.

**Screenshots:**



```
PS C:\Lab\hpc\5> .\3vector
Matrix-vector multiplication completed.
Time taken: 0.001000 seconds
y[0] = 328350, y[1] = 333300, y[2] = 338250
PS C:\Lab\hpc\5> gcc -fopenmp 4prefixsum.c -o prefixsum
PS C:\Lab\hpc\5> .\prefixsum
Prefix sum completed.
Time taken: 0.012000 seconds
Array: 1 3 6 10 15 21 28 36 45 55
PS C:\Lab\hpc\5>
```



**Information:**

**Program Steps:**

1. Declare input array A[N] and result array B[N].
2. Initialize A with values.
3. Set number of threads using omp_set_num_threads().
4. Compute prefix sum in parallel (simple version).
5. Measure execution time using omp_get_wtime() and print the result array.

**Analysis:**

1. Time Complexity: Sequentia O(N)
2. Parallel: Basic version has overhead; optimal version achieves O(N/p+logp), where p = number of threads.
3. Work Distribution :
    a. Each thread computes partial sums of assigned chunk.
    b. Then, offsets are added to adjust the sums across threads.

**Github Link:** [https://github.com/22510021-Shrikrishna/HPC.git](https://github.com/22510021-Shrikrishna/HPC.git)