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

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

**Course:** High Performance Computing Lab

**Practical No. 6**

**Exam Seat No: 21510073**

**Name: SAKSHI DHAMAPURKAR**

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

Implement following Programs using OpenMP with C:

1. Implementation of Matrix-Matrix Multiplication.
2. Implementation of Matrix-vector Multiplication.

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

**Screenshots:**

**#include <stdio.h>**

#include <stdlib.h>

#include <omp.h>

#define SIZE 1000

int main() {

double \*\*A = (double \*\*)malloc(SIZE \* sizeof(double \*));

double \*\*B = (double \*\*)malloc(SIZE \* sizeof(double \*));

double \*\*C = (double \*\*)malloc(SIZE \* sizeof(double \*));

for (int i = 0; i < SIZE; i++) {

A[i] = (double \*)malloc(SIZE \* sizeof(double));

B[i] = (double \*)malloc(SIZE \* sizeof(double));

C[i] = (double \*)malloc(SIZE \* sizeof(double));

}

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

A[i][j] = rand() % 10;

B[i][j] = rand() % 10;

C[i][j] = 0.0;

}

}

// Start the timer

double start\_time = omp\_get\_wtime();

#pragma omp parallel for collapse(3)

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

for (int k = 0; k < SIZE; k++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

// Stop the timer

double end\_time = omp\_get\_wtime();

printf("Time taken for matrix-matrix multiplication: %f seconds\n", end\_time - start\_time);

for (int i = 0; i < SIZE; i++) {

free(A[i]);

free(B[i]);

free(C[i]);

}

free(A);

free(B);

free(C);

return 0;

}

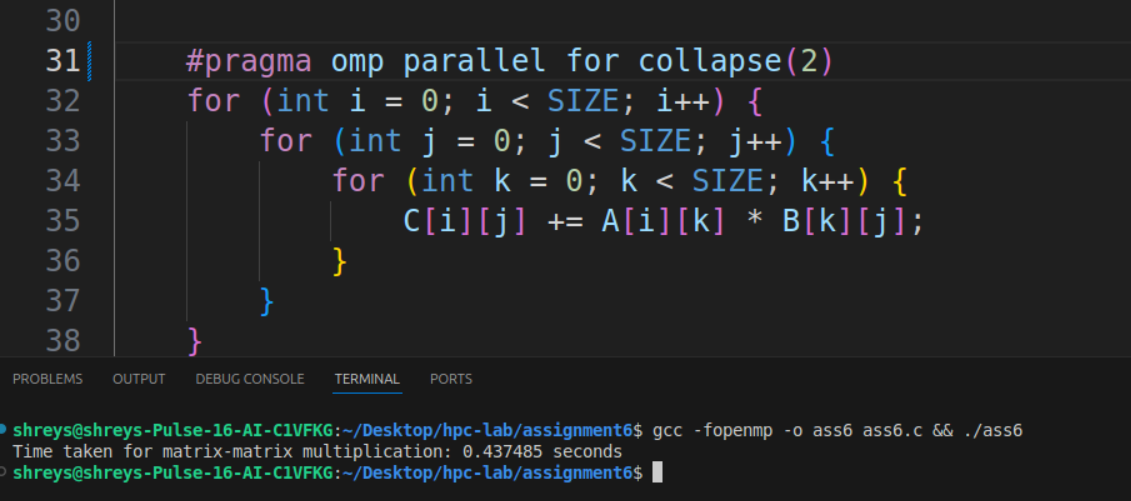
**Information:**

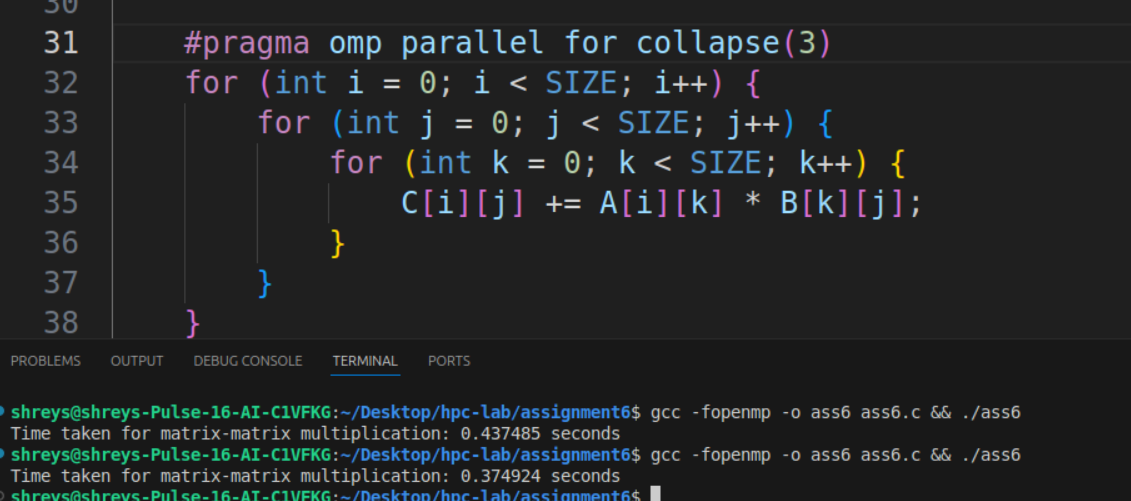
**Matrix Initialization:**

* The code dynamically allocates memory for matrices A, B, and C and initializes matrices A and B with random values, while initializing matrix C with zeros. This setup is typical for matrix-multiplication problems.
* **Matrix-Matrix Multiplication**:
* Using OpenMP, the outer three loops are collapsed into a single parallel loop using #pragma omp parallel for collapse(3). This loop assigns blocks of iterations to threads, helping utilize multiple threads more efficiently by reducing synchronization overhead.
* **Parallelization with** collapse:
* collapse(3) merges the three loops (i, j, k), allowing OpenMP to handle the combined iteration space. This strategy improves load balancing, as each thread processes multiple values of i, j, and k, reducing idle time.
* However, for very large matrices, it may result in memory contention because all threads are accessing C[i][j] concurrently.
* **Timing Analysis**:
* omp\_get\_wtime() measures the elapsed time of matrix multiplication, allowing you to evaluate the effectiveness of parallelization.

**Analysis:**

**For collapse 2**

**For Collapse 3 :**

 **Clear diffrence between the time execution in both the code**

**Problem Statement 2: Implementation of Matrix-vector Multiplication.**

**Screenshots:**

**#include <stdio.h>**

#include <stdlib.h>

#include <omp.h>

#define SIZE 1000

int main() {

double \*\*A = (double \*\*)malloc(SIZE \* sizeof(double \*));

double \*x = (double \*)malloc(SIZE \* sizeof(double));

double \*y = (double \*)malloc(SIZE \* sizeof(double));

for (int i = 0; i < SIZE; i++) {

A[i] = (double \*)malloc(SIZE \* sizeof(double));

}

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

A[i][j] = rand() % 10;

}

x[i] = rand() % 10;

y[i] = 0.0;

}

double start\_time = omp\_get\_wtime();

#pragma omp parallel for

for (int i = 0; i < SIZE; i++) {

for (int j = 0; j < SIZE; j++) {

y[i] += A[i][j] \* x[j];

}

}

double end\_time = omp\_get\_wtime();

printf("Time taken for matrix-vector multiplication: %f seconds\n", end\_time - start\_time);

for (int i = 0; i < SIZE; i++) {

free(A[i]);

}

free(A);

free(x);

free(y);

return 0;

}

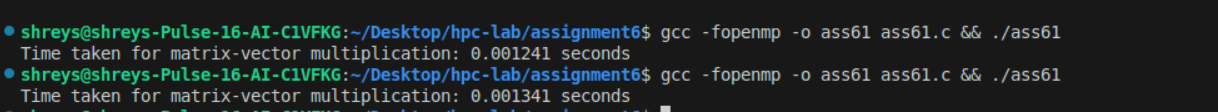
**Information:**

**Memory Allocation:**

* + **Matrix** A: Dynamically allocated as a 2D array (size SIZE x SIZE), where each element represents a value in the matrix.
  + **Vector** x: A single-dimensional array representing the vector to multiply with A.
  + **Result Vector** y: A single-dimensional array initialized to zero, which will store the resulting values of the matrix-vector multiplication.

1. **Initialization**:
   * The code initializes the elements of A and x with random values between 0 and 9. y is initialized to 0 for each element, which is necessary for the summation operation in matrix-vector multiplication.
2. **Matrix-Vector Multiplication**:
   * Each element in y[i] represents the dot product of row i of matrix A with vector x. For each row i, a sum is computed by multiplying each element A[i][j] with x[j] and adding it to y[i].
3. **Parallelization with OpenMP**:
   * The directive #pragma omp parallel for parallelizes the outer loop (for (int i = 0; i < SIZE; i++)). Each iteration of i is independently assigned to different threads, with each thread responsible for computing a row in the result vector y.
   * Since each thread updates a different y[i] index, there is no data race, making this code thread-safe.
4. **Timing**:
   * The code uses omp\_get\_wtime() to record the start and end times, allowing you to measure the time taken for the matrix-vector multiplication operation.

**Analysis:**

**This two output one with 100 size and another with 1000**

**clear diffrencce is there**

**Github Link:**