**Assignment 5**

**PRN**: 22510012

**Name**: Jai Chowgule

**Batch**: B6

Problem Statement-1:

Implementation of Matrix-Matrix Multiplication.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 500

int main() {

static int A[N][N], B[N][N], C[N][N];

double start, end;

// Initialize matrices

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

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

A[i][j] = i + j;

B[i][j] = i - j;

C[i][j] = 0;

}

}

start = omp\_get\_wtime();

#pragma omp parallel for collapse(2) schedule(static)

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

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

int sum = 0;

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

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

}

C[i][j] = sum;

}

}

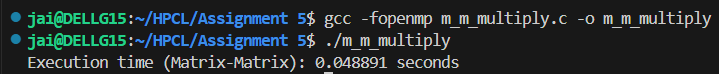
end = omp\_get\_wtime();

printf("Execution time (Matrix-Matrix): %f seconds\n", end - start);

return 0;

}

Output:



Analysis:

#pragma omp parallel for collapse(2) parallelizes the i and j loops together.

schedule(static) divides work evenly among threads.

Time measured using omp\_get\_wtime().

Problem Statement-2:

Implementation of Matrix-scalar Multiplication.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define ROWS 500

#define COLS 500

int main() {

static int A[ROWS][COLS];

int scalar = 5;

double start, end;

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

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

A[i][j] = i + j;

}

}

start = omp\_get\_wtime();

#pragma omp parallel for collapse(2) schedule(static)

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

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

A[i][j] = A[i][j] \* scalar;

}

}

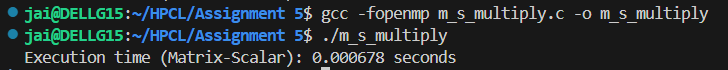
end = omp\_get\_wtime();

printf("Execution time (Matrix-Scalar): %f seconds\n", end - start);

return 0;

}

Output:



Analysis:

Uses collapse(2) to parallelize both row and column loops together.

Very little computation per element, so speedup may be small for small sizes due to thread overhead.

Demonstrates data parallelism where each element update is independent.

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

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define ROWS 500

#define COLS 500

int main() {

static int A[ROWS][COLS];

static int x[COLS], y[ROWS];

double start, end;

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

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

A[i][j] = i + j;

}

}

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

x[j] = j;

}

start = omp\_get\_wtime();

#pragma omp parallel for schedule(static)

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

int sum = 0;

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

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

}

y[i] = sum;

}

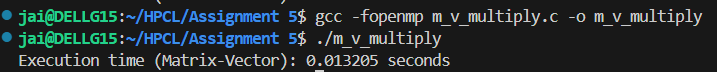
end = omp\_get\_wtime();

printf("Execution time (Matrix-Vector): %f seconds\n", end - start);

return 0;

}

Output:



Analysis:

Each thread computes a row of the result vector independently.

Inner loop over columns is serial to avoid thread coordination overhead.

schedule(static) ensures even distribution of rows among threads.

Problem Statement-4

Implementation of Prefix sum.

Source Code:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 1000000

int main() {

static int arr[N], prefix[N];

double start, end;

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

arr[i] = 1;

}

start = omp\_get\_wtime();

int num\_threads;

#pragma omp parallel

{

int id = omp\_get\_thread\_num();

int i, chunk\_size;

int local\_sum = 0;

#pragma omp single

num\_threads = omp\_get\_num\_threads();

chunk\_size = N / num\_threads;

int start\_idx = id \* chunk\_size;

int end\_idx = (id == num\_threads - 1) ? N : start\_idx + chunk\_size;

// Step 1: Local prefix sum in each chunk

for (i = start\_idx; i < end\_idx; i++) {

local\_sum += arr[i];

prefix[i] = local\_sum;

}

// Step 2: Store total sum of each chunk's last element

static int chunk\_sums[100];

chunk\_sums[id] = local\_sum;

#pragma omp barrier

// Step 3: Accumulate sums from previous chunks

int offset = 0;

for (i = 0; i < id; i++) {

offset += chunk\_sums[i];

}

// Step 4: Add offset to local chunk

for (i = start\_idx; i < end\_idx; i++) {

prefix[i] += offset;

}

}

end = omp\_get\_wtime();

printf("Execution time (Prefix Sum): %f seconds\n", end - start);

printf("First 10 prefix sums: ");

for (int i = 0; i < 10; i++) {

printf("%d ", prefix[i]);

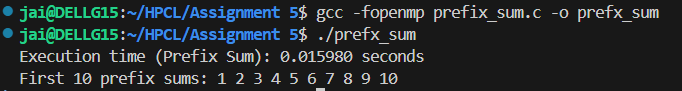
}

printf("\n");

return 0;

}

Output:



Analysis:

Phase 1: Each thread computes a local prefix sum for its chunk.

Phase 2: Each thread records the total sum of its chunk.

Phase 3: Each thread computes the sum of all previous chunks and adds it to its local results.

barrier ensures all threads finish local sums before adding offsets.

Github Link: [**https://github.com/Jai-173/HPCL**](https://github.com/Jai-173/HPCL)