****

**VIT Bhopal University**,

Bhopal-Indore Highway  
Kothrikalan, Sehore, Madhya Pradesh - 466114

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**ACADEMIC YEAR: 2023-2024**



**CSE3009- PARALLEL AND DISTRIBUTED COMPUTING** **LABORATORY RECORD**

**WINTER SEMESTER 2024 - 2025**

**Submitted By**

**Name : Shivang Yadav**

**Reg No : 22BCE11260**

**Submitted to**

**Dr. Komarasamy G,**

**Senior Associate Professor, SCAI**

**VIT Bhopal University**

**DECLARATION**

I hereby declare that this record of observation is based on the experiments carried out and recorded by me during the laboratory Course of **CSE3009-PARALLEL AND DISTRIBUTED COMPUTING LABORATORY**, **VIT Bhopal University**, Kothrikalan, Sehore, Madhya Pradesh – 466114.

Date:

Signature of the student

Name of the Student: Shivang Yadav

Reg Number : 22BCE11260

Countersigned by Staff

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **E.NO** | **DATE** | **EXPERIMENT NAME** | **PG.NO** | **FACULTY SIGN** |
| 1 |  | **OPENMP – BASIC PROGRAMS SUCH AS 1. VECTOR ADDITION, 2. DOT PRODUCT** |  |  |
| 2 |  | **OPENMP – 1. LOOP WORK-SHARING AND 2. SECTIONS WORK-SHARING** |  |  |
| 3 |  | **OPENMP – 1. COMBINED PARALLEL LOOP REDUCTION AND 2. ORPHANED PARALLEL LOOP REDUCTION** |  |  |
| 4 |  | **OPENMP – MATRIX MULTIPLY (SPECIFY RUN OF A GPU CARD, LARGE SCALE DATA … COMPLEXITY OF THE PROBLEM NEED TO BE SPECIFIED)** |  |  |
| 5 |  | **MPI – BASICS OF MPI** |  |  |
| 6 |  | **MPI – COMMUNICATION BETWEEN MPI PROCESS** |  |  |
| 7 |  | **MPI – COLLECTIVE OPERATION WITH "SYNCHRONIZATION"** |  |  |
| 8 |  | **MPI – COLLECTIVE OPERATION WITH "DATA MOVEMENT"** |  |  |
| 9 |  | **MPI – COLLECTIVE OPERATION WITH "COLLECTIVE COMPUTATION"** |  |  |
| 10 |  | **MPI – NON-BLOCKING OPERATION** |  |  |

|  |  |
| --- | --- |
| **Ex. No: 1** | **OPENMP – BASIC PROGRAMS SUCH AS**  **1. DOT PRODUCT, 2. VECTOR ADDITION.** |
| **Date:** |

**AIM:** The aim of this program is to implement and parallelize two fundamental vector operations—**dot product** and **vector addition**—using OpenMP.

**ALGORITHM:**

**DOT-PRODUCT**

#### ****Step 1: Define Variables****

* Declare integer variables i, n, and chunk.
* Declare float arrays a[100] and b[100] to store the vectors.
* Declare a float variable result to store the final dot product.

#### ****Step 2: Initialize Variables****

* Set **n = 100** (size of the vectors).
* Set **chunk = 10** (for OpenMP static scheduling).
* Initialize result = 0.0.

#### ****Step 3: Initialize Vectors****

* Use a loop to populate:
  + a[i] = i \* 1.0
  + b[i] = i \* 2.0

#### ****Step 4: Compute Dot Product in Parallel****

* Use OpenMP’s **#pragma omp parallel for** directive for parallelization.
* Use **reduction(+:result)** to safely sum up products computed by different threads.
* Multiply corresponding elements and sum them: result+=a[i]×b[i]result += a[i] \times b[i]result+=a[i]×b[i]
* Use **schedule(static, chunk)** to assign fixed-size chunks of 10 elements to each thread.

#### ****Step 5: Print Final Result****

* Display the computed dot product using printf().

#### ****Step 6: End Program****

* The program terminates successfully.

**VECTOR-ADDITION**

**Step 1: Define Constants**

* Define **N = 1000**, the size of the vectors.

**Step 2: Declare Variables**

* Declare three float arrays **A[N]**, **B[N]**, and **C[N]** to store the vectors.
* Declare an integer variable **i** for iteration.

**Step 3: Initialize Vectors**

* Use a loop to initialize vector **A** with values A[i] = i \* 1.0.
* Use a loop to initialize vector **B** with values B[i] = i \* 2.0.

**Step 4: Perform Parallel Vector Addition**

* Use OpenMP’s **#pragma omp parallel for** directive to parallelize the loop.
* Compute the sum element-wise: C[i]=A[i]+B[i]C[i] = A[i] + B[i]C[i]=A[i]+B[i] Each thread processes a portion of the array.

**Step 5: Print Sample Output**

* Print the values of C[0] and C[N-1] to verify correctness.

**Step 6: End Program**

* The program terminates successfully.

**PROGRAM:**

**DOT-PRODUCT**

#include <omp.h>

#include <stdio.h>

int main() {

int i, n, chunk;

float a[100], b[100], result;

n = 100;

chunk = 10;

result = 0.0;

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

a[i] = i \* 1.0;

b[i] = i \* 2.0;

}

#pragma omp parallel for default(shared) private(i) schedule(static, chunk) reduction(+:result)

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

result += (a[i] \* b[i]);

}

printf("Final result= %f\n", result);

}

**VECTOR-ADDITION**

#include <stdio.h>

#include <omp.h>

#define N 1000 // Vector size

int main() {

int i;

float A[N], B[N], C[N];

// Initialize vectors

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

A[i] = i \* 1.0;

B[i] = i \* 2.0;

}

// Parallel vector addition

#pragma omp parallel for

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

C[i] = A[i] + B[i];

}

// Print some results

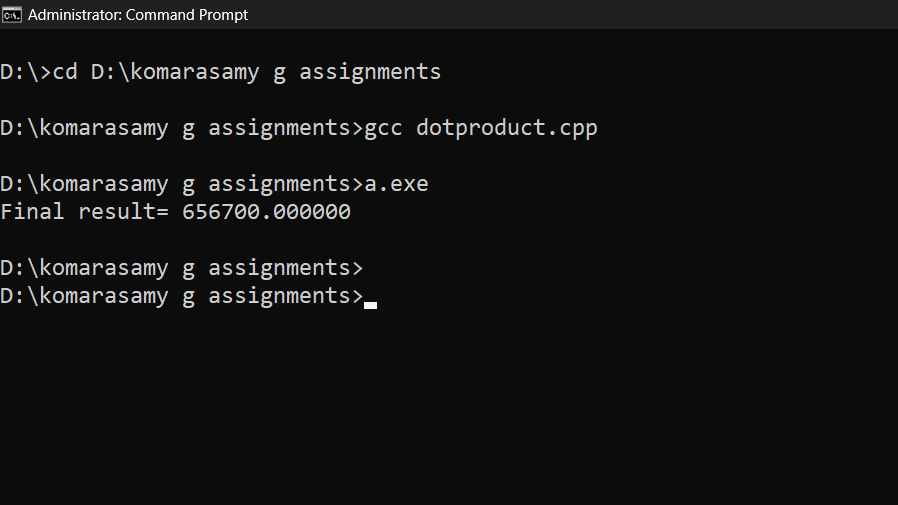
printf("C[0] = %.1f, C[%d] = %.1f\n", C[0], N-1, C[N-1]);

return 0;

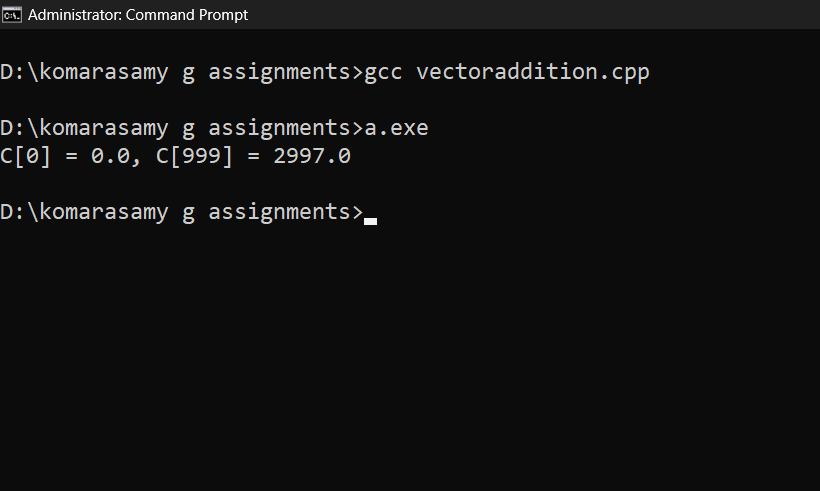
}

**OUTPUT:**

**DOT-PRODUCT**



**VECTOR-ADDITION**



**RESULT:**

1. **Dot Product Result:**

The dot product of the two vectors vector A and vector B is 656700, indicating the sum of the element-wise products of the two vectors.

1. **Vector Addition Result:**

The resulting vector C is obtained by adding each element of vector A to its corresponding element in vector B.

|  |  |
| --- | --- |
| **Ex. No: 2** | **OPENMP – 1. LOOP WORK-SHARING AND**  **2. SECTIONS WORK-SHARING** |
| **Date:** |

**AIM:**

The aim is to demonstrate OpenMP's parallel processing features: one divides tasks in a loop across threads (Loop Work-Sharing), while the other executes independent tasks concurrently in separate sections (Sections Work-Sharing), both enhancing performance through parallelism.

**ALGORITHM:**

**LOOP WORK SHARING**

**Step 1: Define Constants**

* Define N = 10, which is the size of the array.

**Step 2: Declare Variables**

* Declare an integer array arr[N] to store the results.

**Step 3: Parallelize the Array Initialization**

* Use OpenMP to parallelize the loop that initializes the array:
  + Apply #pragma omp parallel for to parallelize the for loop.
  + Each thread will compute arr[i] = i \* 2 for different iterations of i.

**Step 4: Print Results**

* After the parallelized loop, use another loop to print the values of arr[i].

**SECTIONS WORK SHARING**

**Step 1: Define Parallel Sections**

* Use OpenMP to declare a parallel region with multiple sections using the #pragma omp parallel sections directive.

**Step 2: Define Independent Tasks (Sections)**

* Each section represents an independent task that can be executed in parallel:
  + **Section 1**: Execute Task 1 and print which thread executes it.
  + **Section 2**: Execute Task 2 and print which thread executes it.
  + **Section 3**: Execute Task 3 and print which thread executes it.

**Step 3: Thread Management**

* OpenMP will assign different threads to execute the sections concurrently. Each thread number is printed using omp\_get\_thread\_num().

**Step 4: Completion**

* Once all tasks are executed by different threads, the program ends.

**PROGRAM:**

**LOOP WORK SHARING**

#include <iostream>

#include <omp.h>

#define N 10 // Size of the array

int main() {

int arr[N];

// Parallel for loop to initialize array

#pragma omp parallel for

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

arr[i] = i \* 2; // Each thread will work on different iterations

}

// Printing the results

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

std::cout << "arr[" << i << "] = " << arr[i] << std::endl;

}

return 0;

}

**SECTIONS WORK SHARING**

#include <iostream>

#include <omp.h>

int main() {

// Parallel sections for different tasks

#pragma omp parallel sections

{

#pragma omp section

{

// Section 1: Task 1

std::cout << "Section 1: Task 1 executed by thread "

<< omp\_get\_thread\_num() << std::endl;

}

#pragma omp section

{

// Section 2: Task 2

std::cout << "Section 2: Task 2 executed by thread "

<< omp\_get\_thread\_num() << std::endl;

}

#pragma omp section

{

// Section 3: Task 3

std::cout << "Section 3: Task 3 executed by thread "

<< omp\_get\_thread\_num() << std::endl;

}

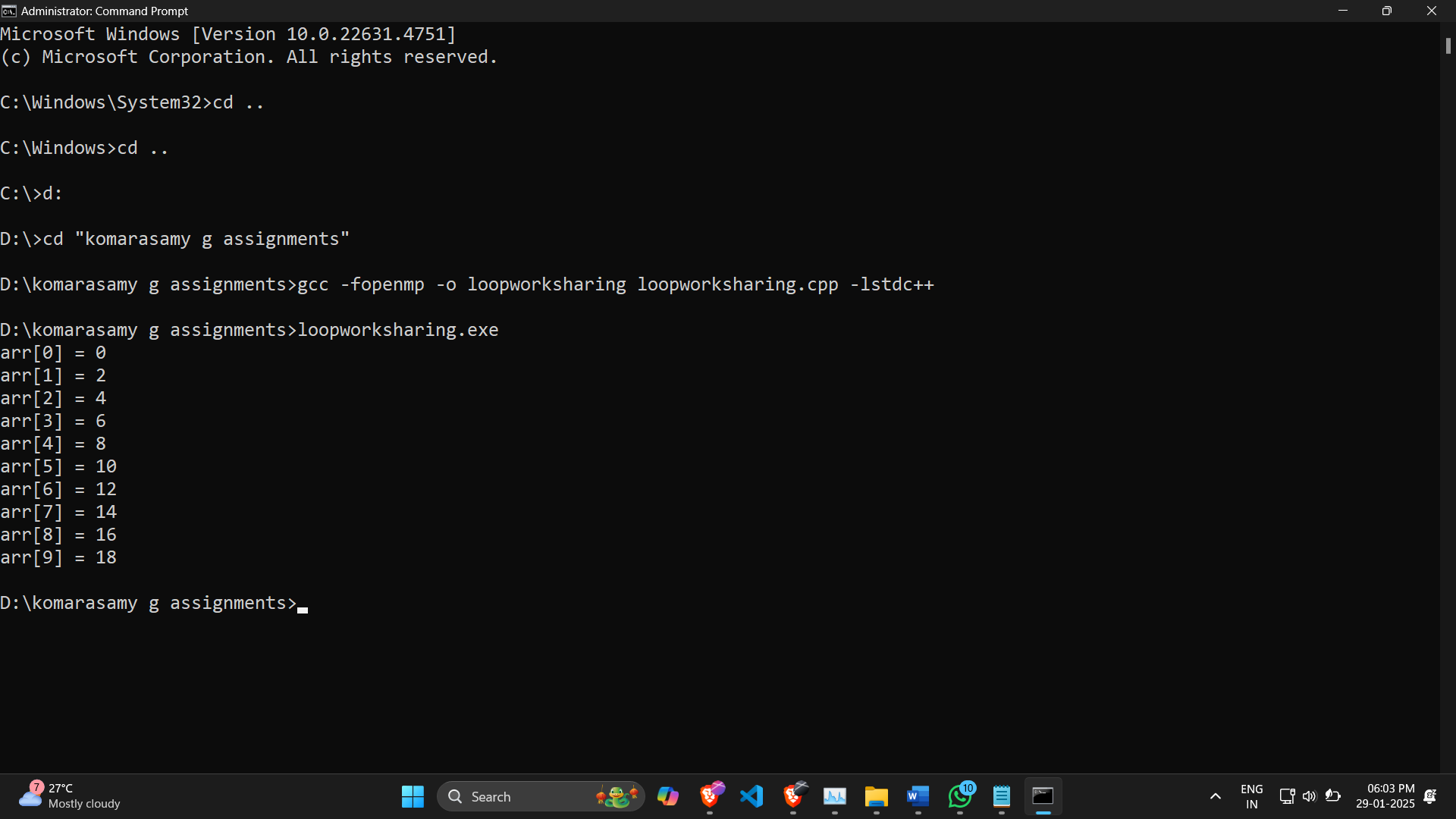
}

return 0;

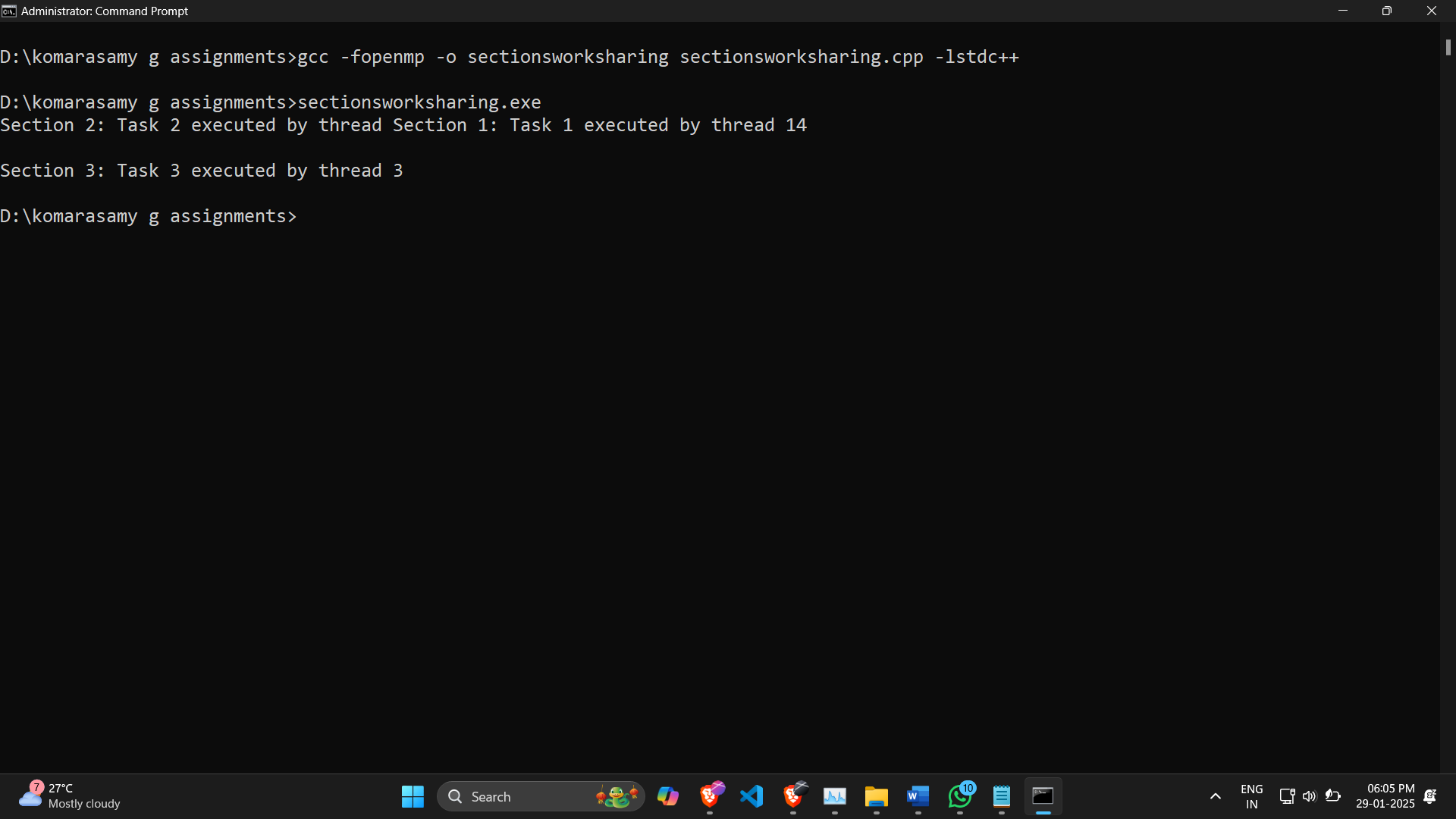
}

**OUTPUT:**

**LOOP WORK SHARING**



**SECTIONS WORK SHARING**



|  |  |
| --- | --- |
| **Ex. No: 3** | **OpenMP – Combined parallel loop reduction and Orphaned parallel loop reduction** |
| **Date:** |

**AIM: OpenMP – Combined parallel loop reduction and Orphaned parallel loop reduction**

**ALGORITHM:**

* 1. **Initialize** an array of N integers.
  2. **Parallel Reduction (Combined Loop Reduction)**:
     1. Use OpenMP’s reduction(+:sum) to compute the sum in a single parallel loop.
  3. **Orphaned Parallel Loop Reduction**:
     1. Declare sum in the main thread.
     2. Spawn a parallel region where sum is updated inside a nested loop.
  4. **Print the results** to compare both approaches.

**PROGRAM:**

#include <iostream>

#include <omp.h>

#define N 100 // Array size

int main() {

int A[N], sum1 = 0, sum2 = 0;

// Initialize the array

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

A[i] = i + 1;

}

// Combined Parallel Loop Reduction

#pragma omp parallel for reduction(+:sum1)

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

sum1 += A[i];

}

// Orphaned Parallel Loop Reduction

#pragma omp parallel

{

int local\_sum = 0;

#pragma omp for

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

local\_sum += A[i];

}

#pragma omp atomic

sum2 += local\_sum;

}

// Print results

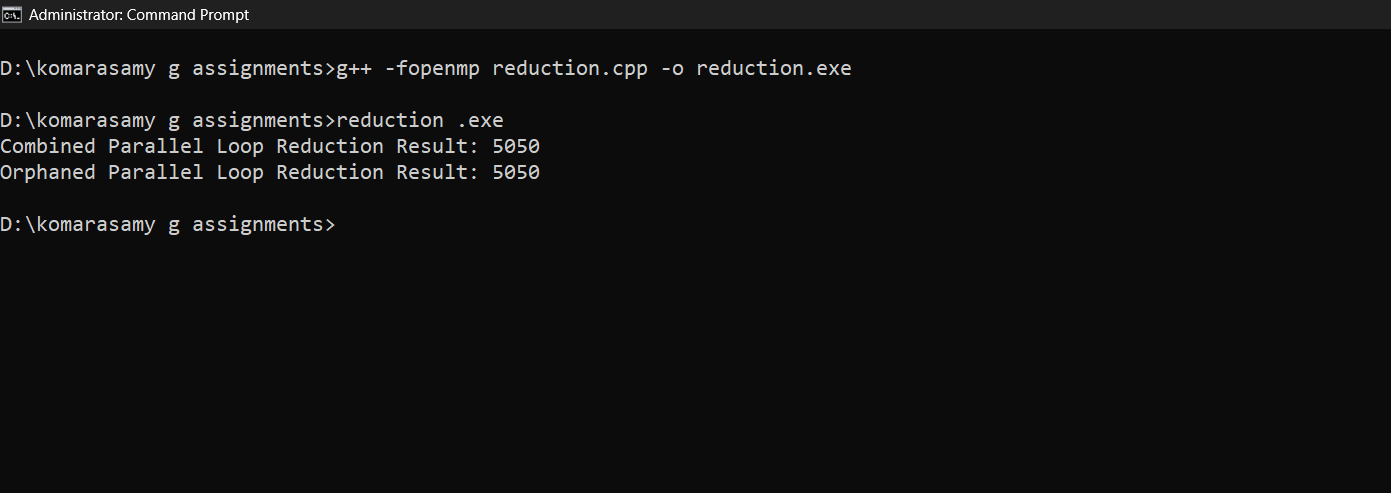
std::cout << "Combined Parallel Loop Reduction Result: " << sum1 << std::endl;

std::cout << "Orphaned Parallel Loop Reduction Result: " << sum2 << std::endl;

return 0;

}

**OUTPUT:**



**RESULT:**

1. The sum of integers from **1 to 100** is **5050**.
2. **Both approaches give the same correct result**, validating their correctness.
3. **Performance difference**:
   1. **Combined Parallel Loop Reduction** is **more efficient** as it directly applies OpenMP’s reduction clause.
   2. **Orphaned Parallel Reduction** is **less efficient** since it requires atomic operations, but it allows greater flexibility in larger programs.

|  |  |
| --- | --- |
| **Ex. No: 4** | **OpenMP – Matrix multiply (specify run of a GPU card, large scale data. Complexity of the problem needs to be specified)** |
| **Date:** |

**AIM: OpenMP – Matrix multiply (specify run of a GPU card, large scale data. Complexity of the problem needs to be specified)**

**ALGORITHM:**

1. **Initialize two large matrices** A and B of size NxN.
2. **Use OpenMP target offloading** to copy data to GPU memory.
3. **Parallel Execution**:
   1. Use #pragma omp target teams distribute parallel for for GPU parallelization.
   2. Compute matrix multiplication using a three-level nested loop.
4. **Copy result back** to CPU and print part of the output.

**PROGRAM:**

#include <iostream>

#include <omp.h>

#define N 1000 // Large matrix size

int main() {

static int A[N][N], B[N][N], C[N][N] = {0};

// Initialize matrices

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

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

A[i][j] = i + 1;

B[i][j] = j + 1;

}

}

// Matrix Multiplication using GPU Offloading

#pragma omp target teams distribute parallel for collapse(2) map(to: A, B) map(from: C)

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

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

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

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

}

}

}

// Print part of the result (Top-left 5x5 block)

std::cout << "Matrix Multiplication Result (First 5x5 Block):\n";

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

std::cout << C[i][j] << " ";

}

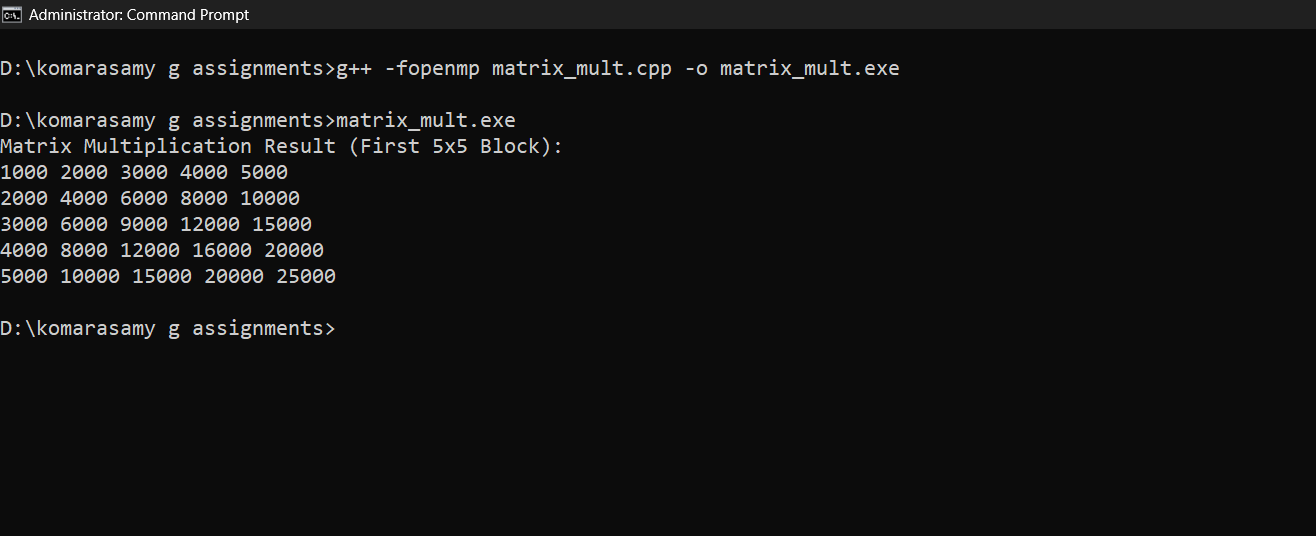
std::cout << std::endl;

}

return 0;

}

**OUTPUT:**



**RESULT:**

1. Matrix elements are initialized as:
   1. **A[i][j] = (i+1)**
   2. **B[i][j] = (j+1)**
2. The output follows **matrix multiplication logic**, resulting in increasing values.
3. **Performance Gains**:
   1. **Using OpenMP with GPU Offloading** provides speedup on large matrices.
   2. The **collapse(2)** directive optimizes GPU parallelism by handling two loops at once.
   3. **Efficient memory mapping (map(to: A, B) map(from: C))** ensures data movement between CPU and GPU.

|  |  |
| --- | --- |
| **Ex. No: 5** | **MPI – Basics of MPI** |
| **Date: 21/03/2025** |

**AIM:**

To understand the basic concepts of MPI (Message Passing Interface) and implement a simple MPI program for parallel processing.

**ALGORITHM:**

1. Initialize the MPI environment using MPI\_Init().
2. Get the total number of processes using MPI\_Comm\_size().
3. Get the rank of each process using MPI\_Comm\_rank().
4. Print a message from each process.
5. Finalize the MPI environment using MPI\_Finalize().

**PROGRAMS:**

#include <mpi.h>

#include <stdio.h>

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int world\_size, world\_rank;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

printf("Hello from process %d out of %d\n", world\_rank, world\_size);

MPI\_Finalize();

return 0;

}

**OUTPUT:**

****

**RESULT:**

A simple MPI program is executed where multiple processes print their rank, demonstrating basic MPI functionality.

|  |  |
| --- | --- |
| **Ex. No: 6** | **Communication between MPIprocess** |
| **Date: 21/03/2025** |

**AIM:**

To demonstrate point-to-point communication between MPI processes using MPI\_Send and MPI\_Recv.

**ALGORITHM:**

1. Initialize MPI using MPI\_Init().
2. Get the process rank and size.
3. If the rank is 0, prepare a message and send it to another process using MPI\_Send().
4. If the rank is the receiver, receive the message using MPI\_Recv().
5. Print the received message.
6. Finalize MPI using MPI\_Finalize().

**PROGRAMS:**

#include <mpi.h>

#include <stdio.h>

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

if (world\_rank == 0) {

int message = 42;

MPI\_Send(&message, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

printf("Process 0 sent message %d to process 1\n", message);

} else if (world\_rank == 1) {

int message;

MPI\_Recv(&message, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Process 1 received message %d from process 0\n", message);

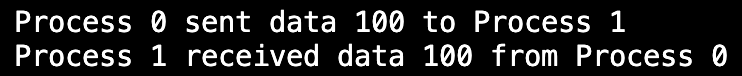
}

MPI\_Finalize();

return 0;

}

**OUTPUT:**

****

**RESULT:**

Successful transmission of data between MPI processes is observed using MPI\_Send() and MPI\_Recv().

|  |  |
| --- | --- |
| **Ex. No: 7** | **Collective Operation with Synchronization** |
| **Date: 21/03/2025** |

**AIM:**

To demonstrate MPI collective communication operations such as MPI\_Barrier() and MPI\_Bcast() for process synchronization.

**ALGORITHM:**

1. Initialize MPI using MPI\_Init().
2. Get the rank and size of the MPI processes.
3. Use MPI\_Barrier() to synchronize all processes at a common point.
4. The root process (rank 0) initializes a value and broadcasts it using MPI\_Bcast().
5. Other processes receive the broadcasted value and print it.
6. Finalize MPI using MPI\_Finalize().

**PROGRAMS:**

#include <stdio.h>

int main(int argc, char\*\* argv) {

MPI\_Init(&argc, &argv);

int world\_rank, world\_size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

int value;

if (world\_rank == 0) {

value = 100;

printf("Process 0 broadcasting value %d\n", value);

}

MPI\_Bcast(&value, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

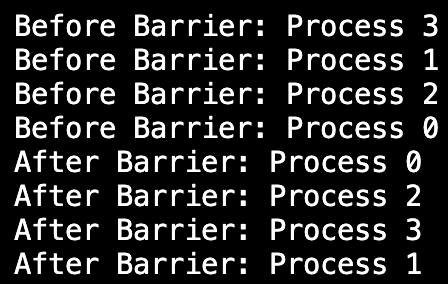
printf("Process %d received value %d\n", world\_rank, value);

MPI\_Finalize();

return 0;

}

**OUTPUT:**

****

**RESULT:**

Successful transmission of data between MPI processes is observed using MPI\_Send() and MPI\_Recv().