VIT Bhopal University,

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

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ACADEMIC YEAR: 2024-2025

CSE3009- PARALLEL AND DISTRIBUTED COMPUTING LABORATORY RECORD

WINTER SEMESTER 2024 - 2025

Submitted By

Name	**:**	**Saransh Prajapati**
Reg No	**:**	**22BCE10795**

Submitted to

Dr. Komarasamy G,
Senior Associate Professor, SCAI
VIT Bhopal University

DECLARATION

I **Saransh Prajapati (22BCE10795)** 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: Saransh Prajapati

Reg Number : 22BCE10795

Countersigned by Staff

INDEX

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

AIM: OpenMP – Basic programs such as Vector addition, Dot Product

ALGORITHM:

1. Dot Product:

- Input Vectors: Obtain two vectors a and b of equal length n.
- Initialize Result
- Iterate Over Elements:
    - For each index i from 0 to $n-1$:
        - Multiply $a[i]$ by $b[i]$ and add the result to result.
- Output Result.

2. Vector Addition:

- Define Constants: Set the size of the arrays and the number of threads.
- Allocate Memory: Dynamically allocate memory for the input arrays and the result array.
- Initialize Arrays with values.
- Parallelize Addition:
    - Use #pragma omp parallel for to distribute the addition operation across multiple threads.
- Free the dynamically allocated memory.

PROGRAM:

DOT-PRODUCT

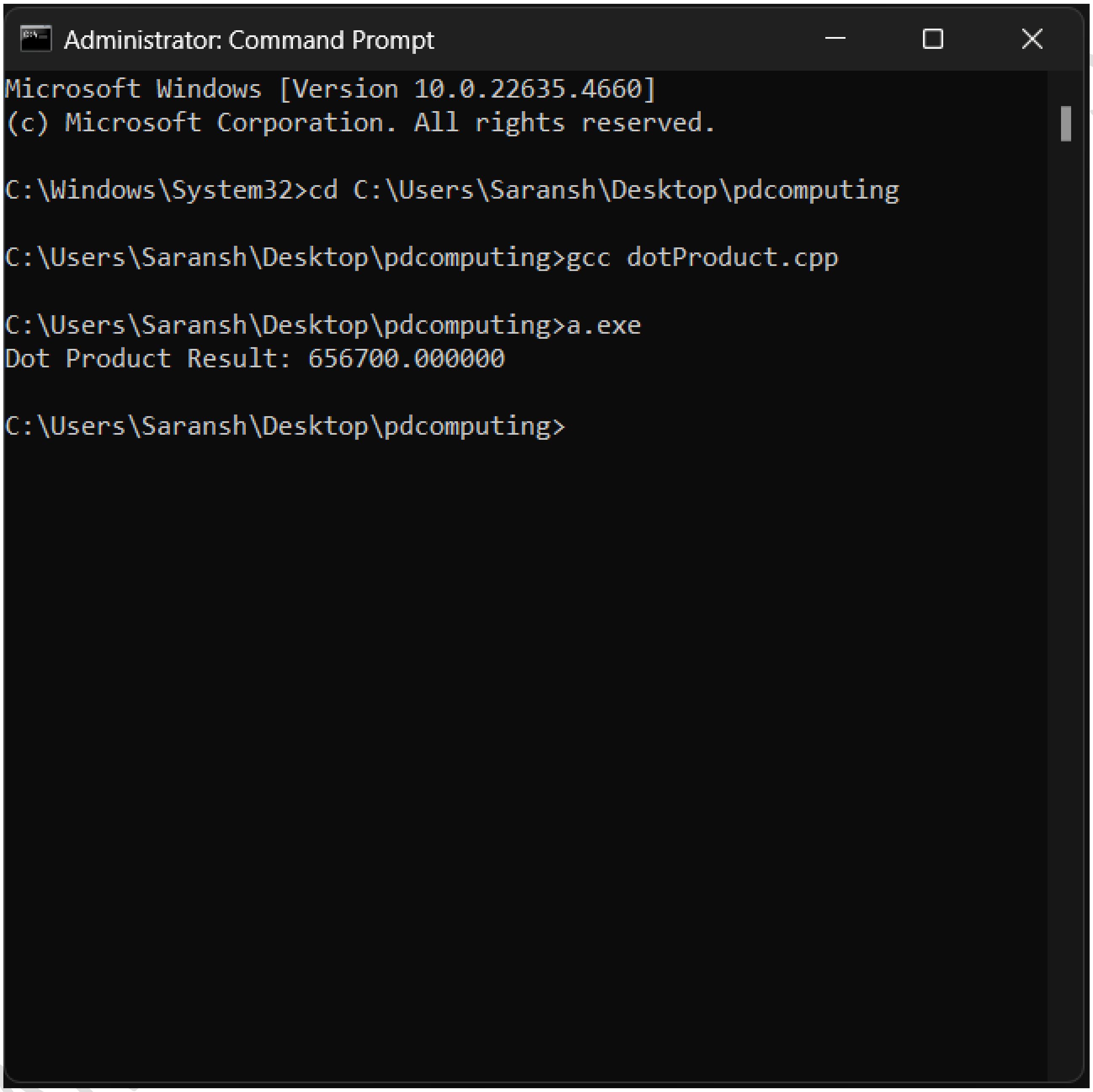
```
#include <omp.h>
#include <stdio.h>

#define SIZE 100
#define CHUNK_SIZE 10

int main() {

   int index, total_elements;
   float vector_a[SIZE], vector_b[SIZE], dot_product = 0.0;
   total_elements = SIZE;

   for (index = 0; index < total_elements; index++) {
   vector_a[index] = index * 1.0f;
   vector_b[index] = index * 2.0f;

   }

   #pragma omp parallel for default(none) shared(vector_a, vector_b, total_elements) private(index)
schedule(static, CHUNK_SIZE) reduction(+:dot_product)

   for (index = 0; index < total_elements; index++) {
        dot_product += (vector_a[index] * vector_b[index]);
   }
   printf("Dot Product Result: %f\n", dot_product);

}
```

VECTOR-ADDITION

```
#include <omp.h>
#include <stdio.h>

int main() {
   int vector_a[5] = {1, 4, 6, 7, 8};
   int vector_b[5] = {2, 3, 5, 9, 0};
   int vector_c[5] = {0, 0, 0, 0, 0};

   omp_set_num_threads(3);

   #pragma omp parallel for shared(vector_a, vector_b, vector_c) schedule(static, 1)
   for (int idx = 0; idx < 5; idx++) {
      printf("Thread %d works on element %d\n", omp_get_thread_num(), idx);
      vector_c[idx] = vector_a[idx] + vector_b[idx];
   }

   printf("\n");
   printf("i \t vector_a[i] \t + \t vector_b[i] \t = \t vector_c[i] \n");
   for (int idx = 0; idx < 5; idx++) {
      printf("%d \t%d \t \t%d \t \t %d \n", idx, vector_a[idx], vector_b[idx], vector_c[idx]);
   }

   return 0;
}
```

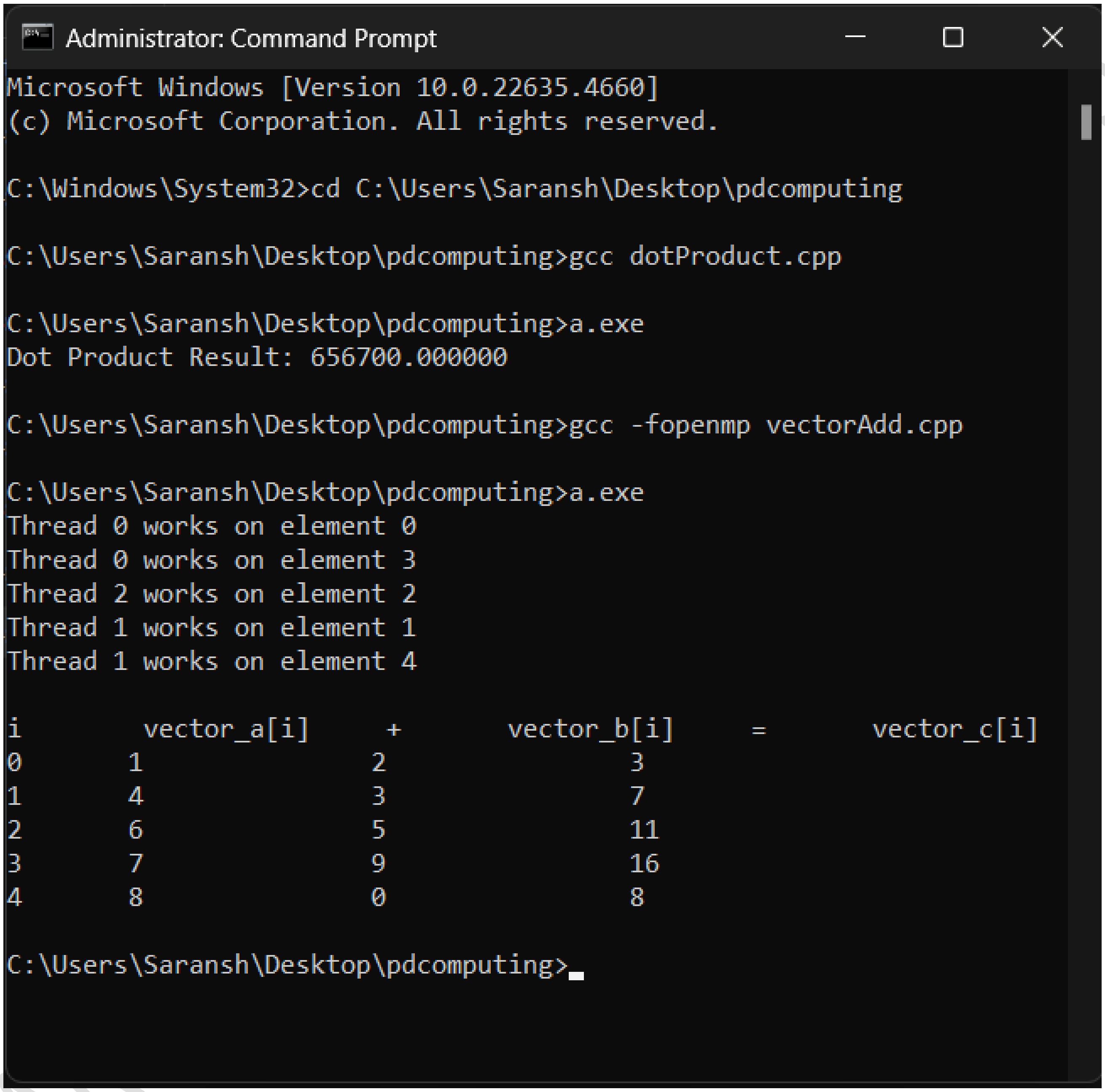
OUTPUT:

1. **Dot Product Output:**



```
Microsoft Windows [Version 10.0.22635.4660]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>cd C:\Users\Saransh\Desktop\pdcomputing

C:\Users\Saransh\Desktop\pdcomputing>gcc dotProduct.cpp

C:\Users\Saransh\Desktop\pdcomputing>a.exe
Dot Product Result: 656700.000000

C:\Users\Saransh\Desktop\pdcomputing>
```

2. Vector Addition Output:



Administrator: Command Prompt

```
Microsoft Windows [Version 10.0.22635.4660]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>cd C:\Users\Saransh\Desktop\pdcomputing

C:\Users\Saransh\Desktop\pdcomputing>gcc dotProduct.cpp

C:\Users\Saransh\Desktop\pdcomputing>a.exe
Dot Product Result: 656700.000000

C:\Users\Saransh\Desktop\pdcomputing>gcc -fopenmp vectorAdd.cpp

C:\Users\Saransh\Desktop\pdcomputing>a.exe
Thread 0 works on element 0
Thread 0 works on element 3
Thread 2 works on element 2
Thread 1 works on element 1
Thread 1 works on element 4

i        vector_a[i]     +       vector_b[i]     =       vector_c[i]
0       1               2                3
1       4               3                7
2       6               5                11
3       7               9                16
4       8               0                8

C:\Users\Saransh\Desktop\pdcomputing>
```

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.

2. **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 – Loop work-sharing and sections work-sharing
Date: 27/1/25	

AIM: OpenMP – Loop work-sharing and sections work-sharing

ALGORITHM:

1. Loop work-sharing:

- Uses #pragma omp parallel to create a parallel region
- Distribute iterations: Assign a starting and ending iteration to each thread. This can be done statically (pre-determined ranges) or dynamically (work is assigned as threads finish).
- Parallel Execution: Each thread executes the loop iterations assigned to it.

2. Sections Work-Sharing:

- Assign Sections: Each section of code is assigned to a separate thread.
- Parallel Execution: Each thread executes the section assigned to it.
- Implicit Barrier: All threads wait until all sections have completed execution.
- Continue execution.

PROGRAM:

Loop Work-Sharing

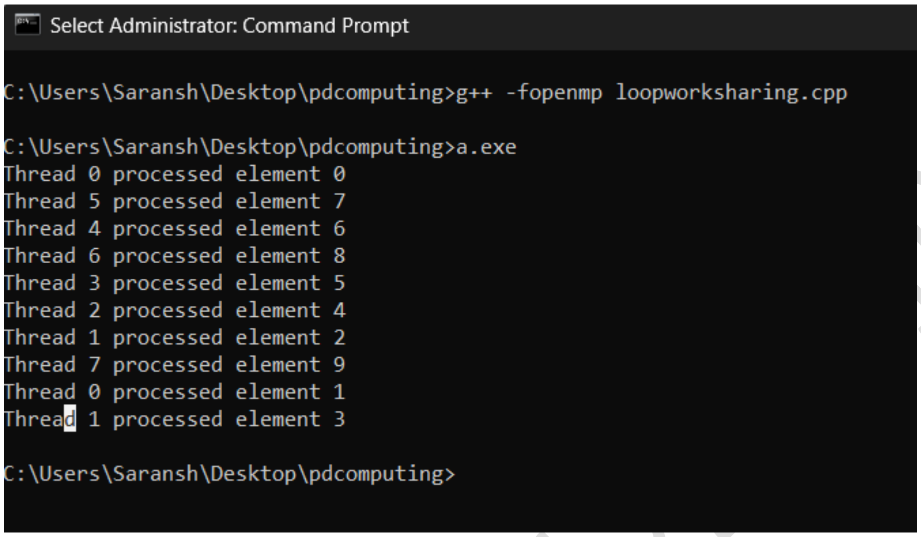
```
#include <iostream>
#include <omp.h>
#include <stdio.h>

int main() {
   int n = 10;
   int arr[n];
   #pragma omp parallel for
   for (int i = 0; i < n; i++) {
      arr[i] = i * 2;
      cout << "Thread " << omp_get_thread_num() << " processed element " << i << endl;

   }
   return 0;
}
```

Sections Work-Sharing

```
#include <iostream>
#include <stdio.h>
#include <omp.h>

void function1() {
  cout << "Thread " << omp_get_thread_num() << " executing function1" << endl;
}

void function2() {
  cout << "Thread " << omp_get_thread_num() << " executing function2" << endl;
}

int main() {
  #pragma omp parallel sections
  {
    #pragma omp section
    function1();

    #pragma omp section
    function2();
  }

  return 0;
}
```

OUTPUT:

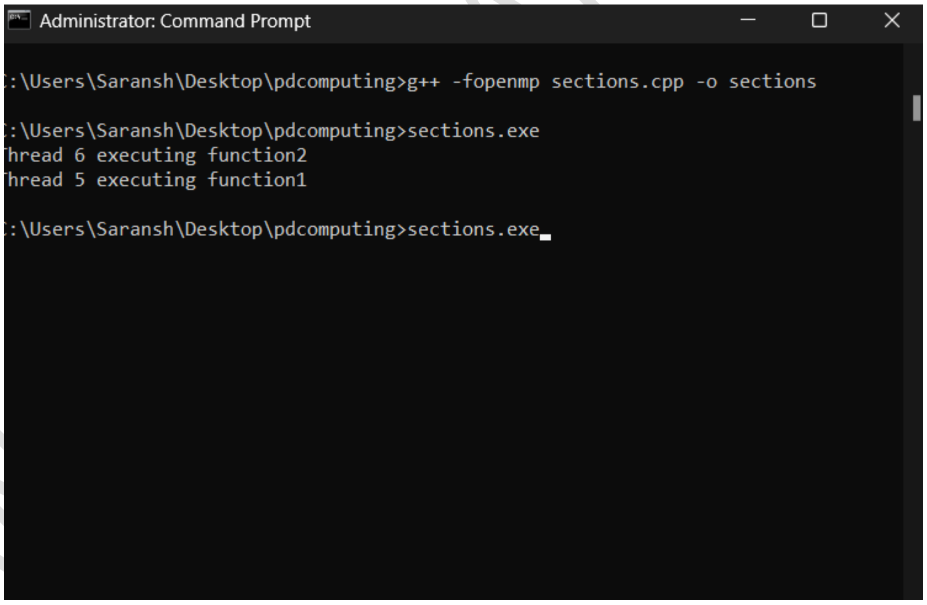
Loop Work-Sharing:



Select Administrator: Command Prompt

```
C:\Users\Saransh\Desktop\pdcomputing>g++ -fopenmp loopworksharing.cpp

C:\Users\Saransh\Desktop\pdcomputing>a.exe
Thread 0 processed element 0
Thread 5 processed element 7
Thread 4 processed element 6
Thread 6 processed element 8
Thread 3 processed element 5
Thread 2 processed element 4
Thread 1 processed element 2
Thread 7 processed element 9
Thread 0 processed element 1
Thread 1 processed element 3

C:\Users\Saransh\Desktop\pdcomputing>
```

Sections Work-Sharing:



Administrator: Command Prompt

```
:\Users\Saransh\Desktop\pdcomputing>g++ -fopenmp sections.cpp -o sections

:\Users\Saransh\Desktop\pdcomputing>sections.exe
hread 6 executing function2
hread 5 executing function1

:\Users\Saransh\Desktop\pdcomputing>sections.exe
```

RESULT:

Both Loop work-sharing & sections work sharing are successfully implemented.

Ex. No: 3	**OpenMP – Combined parallel loop reduction and Orphaned parallel loop reduction**
Date: 12/2/25	

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

ALGORITHM:

1. Combined parallel loop reduction:

- The sum variable is initialized before the parallel region. This is important.
- #pragma omp parallel for reduction(+: sum): This directive tells OpenMP to parallelize the loop and perform a reduction on the sum variable using the + operator (addition).
- Inside the loop, each thread adds its portion of the array to its private copy of sum.
- After the loop, OpenMP automatically adds all the private sum values together and stores the final result in the original sum variable

2. Orphaned parallel loop reduction:

- Assign Sections: Each section of code is assigned to a separate thread.
- Data Initialization: Create and initialize the data that will be processed in parallel.
- Parallel Region (Orphaned): Start a parallel region using #pragma omp parallel. The loop that does the parallel work will be *outside* this region (hence "orphaned").
- Parallel Loop (#pragma omp for): Use #pragma omp for *outside* the parallel region to distribute the loop iterations among the threads. This is the "orphaned" part.
- Thread-Local Variables: Inside the parallel region, declare any variables that need to be private to each thread. This is where you'd perform thread-specific setup.
- Computation (Inside the Loop): The actual computation happens within the loop, which is being executed in parallel. Each thread works on its assigned chunk of the loop.

- Reduction (If Needed): If you need to combine results from each thread (like summing up values), use a reduction clause or a critical section (although reduction is generally preferred for performance).
- Synchronization (If Needed): If threads need to coordinate or wait for each other, use OpenMP synchronization constructs (e.g., barriers).
- Result Collection (If Needed): Gather the results from each thread after the parallel loop has finished.
- Cleanup: Perform any necessary cleanup.

PROGRAM:

Combined parallel loop reduction:

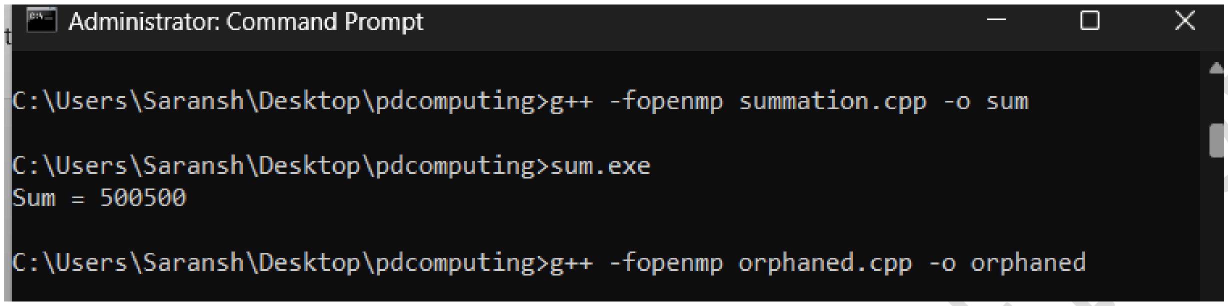
```
#include <iostream>
#include <omp.h>
#include <stdio.h>
using namespace std;


int main() {
   int n = 1000;
   int arr[n];
   for (int i = 0; i < n; i++) {
       arr[i] = i + 1;
   }

   int sum = 0;
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < n; i++) {
       sum += arr[i];
   }

  cout << "Sum = " << sum << endl;
   return 0;
}
```

Orphaned parallel loop reduction:

```
#include <iostream>
#include <omp.h>
#include <vector>
#include <cmath>
using namespace std;


int main() {
   int n = 100000;
   vector<double> data(n);

   // 1. Data Initialization
   for (int i = 0; i < n; ++i) {
      data[i] = i * 0.01; // Example data
   }
   double sum_of_sin = 0.0;
   // 2. Parallel Region (Orphaned)
   #pragma omp parallel
   {
      // 4. Thread-Local Variables (if needed)
      double local_sum = 0.0;
      // 3. Parallel Loop (Orphaned)
      #pragma omp for
      for (int i = 0; i < n; ++i) {
         // 5. Computation (Inside the Loop)
         local_sum += sin(data[i]);
      }

      // 6. Reduction (using critical section for illustration - reduction is better!)
      #pragma omp critical
      {
        sum_of_sin += local_sum;
      }

   } // End of parallel

   // 8. Result Collection (already done with reduction)

   cout << "Sum of sin(data[i]): " << sum_of_sin << endl;

   return 0;
}
```

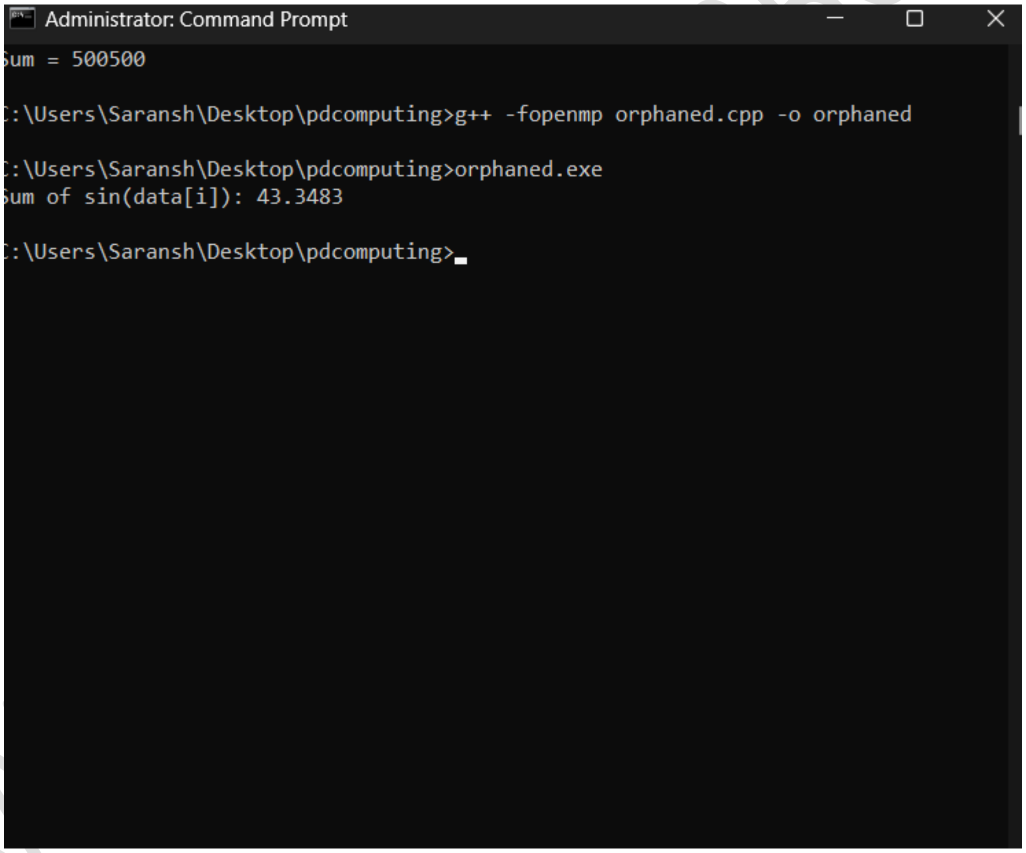
OUTPUT:

Combined parallel loop reduction:



```
C:\Users\Saransh\Desktop\pdcomputing>g++ -fopenmp summation.cpp -o sum

C:\Users\Saransh\Desktop\pdcomputing>sum.exe
Sum = 500500

C:\Users\Saransh\Desktop\pdcomputing>g++ -fopenmp orphaned.cpp -o orphaned
```

Orphaned parallel loop reduction:



```
um = 500500

:\Users\Saransh\Desktop\pdcomputing>g++ -fopenmp orphaned.cpp -o orphaned

:\Users\Saransh\Desktop\pdcomputing>orphaned.exe
um of sin(data[i]): 43.3483

:\Users\Saransh\Desktop\pdcomputing>
```

RESULT:

Both Combined parallel loop reduction & Orphaned parallel loop reduction are successfully implemented.

Ex. No: 4	OpenMP – Matrix multiply (specify run of a GPU card, large scale data ... Complexity of the problem need to be specified)
Date: 12/2/25	

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

ALGORITHM:

1. Combined parallel loop reduction:

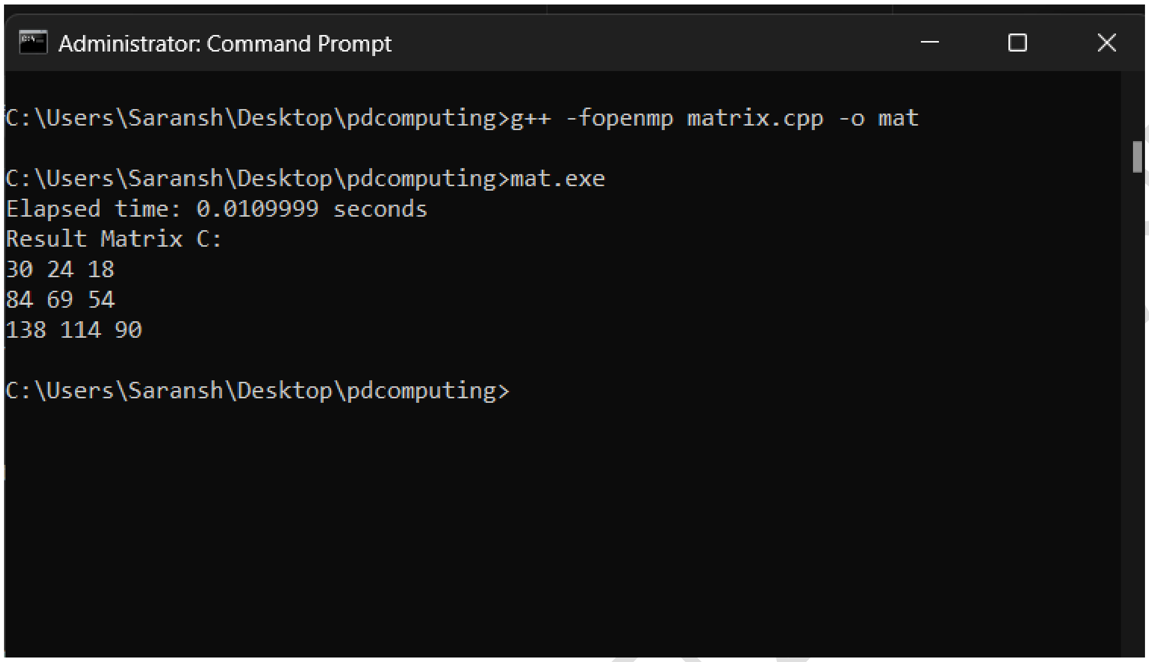
- The sum variable is initialized before the parallel region. This is important.
- **Input:** Two matrices A (n x m) and B (m x p).
- **Output:** Result matrix C (n x p), where C[i][j] is the dot product of the i-th row of A and the j-th column of B.
- **Initialization:** Create a matrix C of size n x p and initialize all its elements to 0.
- **Multiplication:**
  - Iterate through each row i of matrix A (from 0 to n-1).
  - Iterate through each column j of matrix B (from 0 to p-1).
  - Iterate through each element k in the i-th row of A and the k-th row of B(from 0 to m-1).
  - Calculate the dot product: C[i][j] += A[i][k] * B[k][j].
- **Parallelization:** The outermost loop (iterating through rows of A) is parallelized using OpenMP's #pragma omp parallel for directive. This distributes the computation of different rows of C among multiple threads.
- **Return:** The resulting matrix C.

PROGRAM:

Orphaned parallel loop reduction:

```
#include <iostream>
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;


void matrixMultiply(const vector<vector<int>>& A, const vector<vector<int>>& B, vector<vector<int>>& C) {
   int n = A.size(); // Rows of A and C
   int m = B.size(); // Rows of B
   int p = B[0].size(); // Columns of B and C

   if (m != n) {
      cerr << "Matrices are not compatible for multiplication." << endl;
      return;
   }

   C.resize(n, vector<int>(p, 0));

   // Parallelize the outer loop (rows of A)

   #pragma omp parallel for
   for (int i = 0; i < n; ++i) {
       for (int j = 0; j < p; ++j) {
          for (int k = 0; k < m; ++k) {
             C[i][j] += A[i][k] * B[k][j];

         }
      }
   }
}
int main() {
   vector<vector<int>> A = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
   };

   vector<vector<int>> B = {
       {9, 8, 7},
      {6, 5, 4},
      {3, 2, 1}
   };
   vector<vector<int>> C; // Result matrix
   double start_time = omp_get_wtime();
   matrixMultiply(A, B, C);
   double end_time = omp_get_wtime();
   double elapsed_time = end_time - start_time;
   cout << "Elapsed time: " << elapsed_time << " seconds" << endl;
   cout << "Result Matrix C:" << endl;

   for (const auto& row : C) {
      for (int val : row) {
         cout << val << " ";
      }
      cout << endl;
   }
   return 0;
}
```

OUTPUT:



```
C:\Users\Saransh\Desktop\pdcomputing>g++ -fopenmp matrix.cpp -o mat

C:\Users\Saransh\Desktop\pdcomputing>mat.exe
Elapsed time: 0.0109999 seconds
Result Matrix C:
30 24 18
84 69 54
138 114 90

C:\Users\Saransh\Desktop\pdcomputing>
```

RESULT: