Assignment – 1

Title: Implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP.

Problem Statement: Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS.

Objectives: Students should be able to perform Parallel Breadth First Search based on existing algorithms using OpenMP.

Theory:

BFS: BFS stands for Breadth-First Search. It is a graph traversal algorithm used to explore all the nodes of a graph or tree systematically, starting from the root node or a specified starting point, and visiting all the neighbouring nodes at the current depth level before moving on to the next depth level. The algorithm uses a queue data structure to keep track of the nodes that need to be visited, and marks each visited node to avoid processing it again. The basic idea of the BFS algorithm is to visit all the nodes at a given level before moving on to the next level, which ensures that all the nodes are visited in breadth-first order. BFS is commonly used in many applications, such as finding the shortest path between two nodes, solving puzzles, and searching through a tree or graph.

Example of BFS:

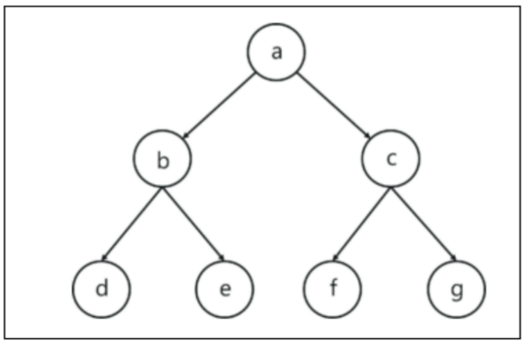
Now let us look at the steps involved in traversing a graph by using Breadth-First Search:

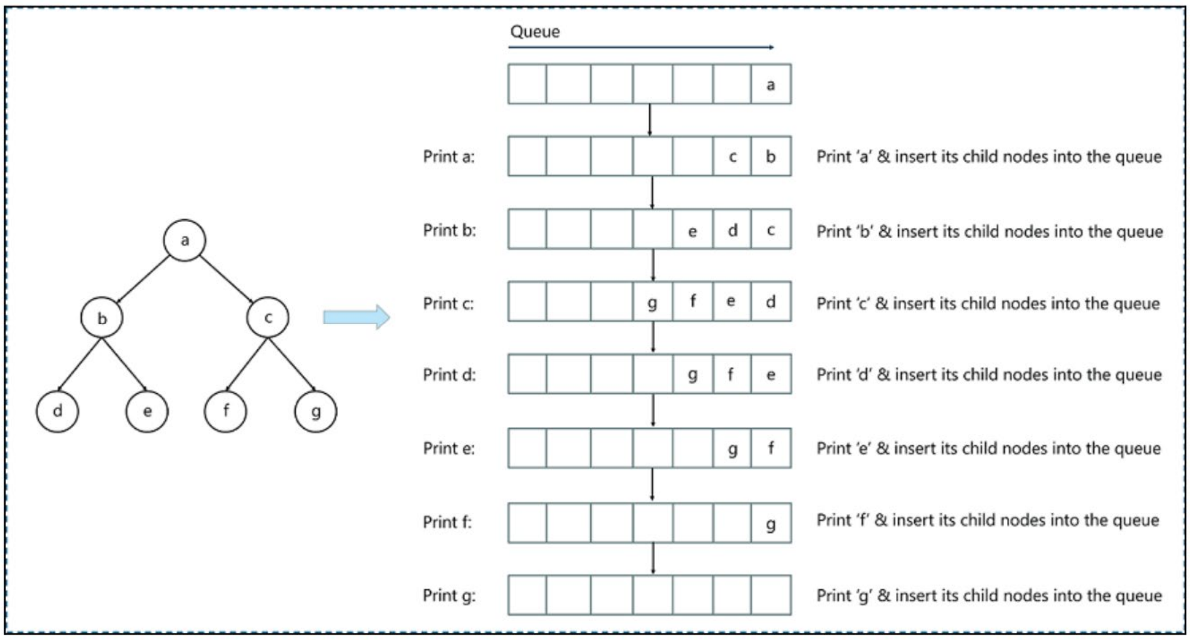
Step 1: Take an Empty Queue.

Step 2: Select a starting node (visiting a node) and insert it into the Queue.

Step 3: Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (exploring a node) into the Queue.

Step 4: Print the extracted node.




Queue
a
Print a:
c
b
Print 'a' & insert its child nodes into the queue
Print b:
e
d
c
Print 'b' & insert its child nodes into the queue
Print c:
g
f
e
d
Print 'c' & insert its child nodes into the queue
Print d:
g
f
e
Print 'd' & insert its child nodes into the queue
Print e:
g
f
Print 'e' & insert its child nodes into the queue
Print f:
g
Print 'f' & insert its child nodes into the queue
Print g:
Print 'g' & insert its child nodes into the queue
a
b
c
d
e
f
g

DFS:

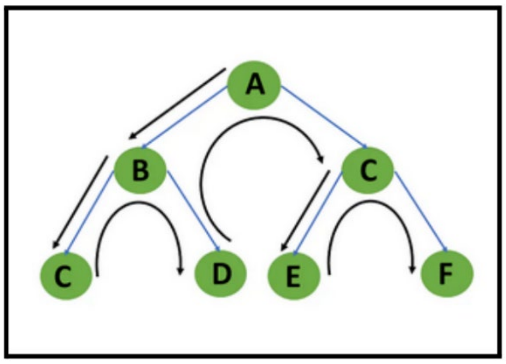
DFS stands for Depth-First Search. It is a popular graph traversal algorithm that explores as far as possible along each branch before backtracking. This algorithm can be used to find the shortest path between two vertices or to traverse a graph in a systematic way. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. The backtracking is done to explore the next branch that has not been explored yet. DFS can be implemented using either a recursive or an iterative approach. The recursive approach is simpler to implement but can lead to a stack overflow error for very large graphs. The iterative approach uses a stack to keep track of nodes to be explored and is preferred for larger graphs. DFS can also be used to detect cycles in a graph. If a cycle exists in a graph, the DFS algorithm will eventually reach a node that has already been visited, indicating that a cycle exists.

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited

2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.



Example of DFS:

To implement DFS traversal, you need to take the following stages.

Step 1: Create a stack with the total number of vertices in the graph as the size.

Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to the stack.

Step 3 - Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.

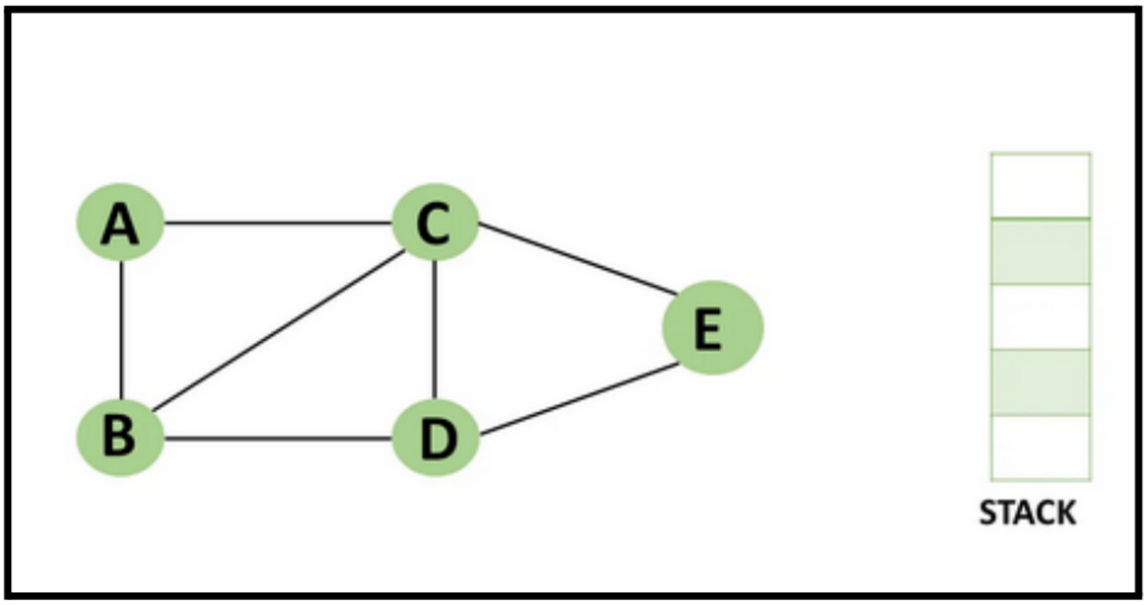
Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.

Step 5 - If there are no new vertices to visit, go back and pop one from the stack using backtracking.

Step 6 - Continue using steps 3, 4, and 5 until the stack is empty.

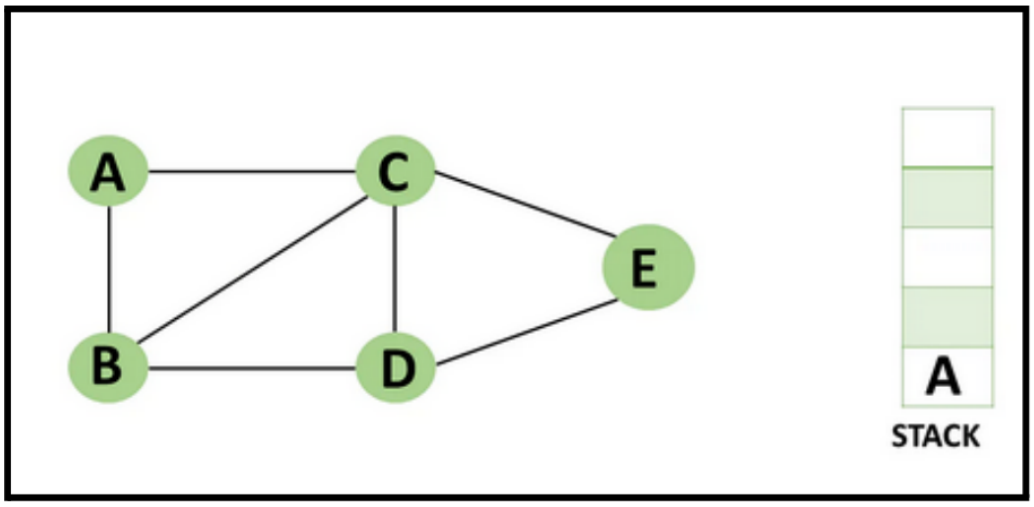
Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

Consider the following graph as an example of how to use the DFS algorithm.



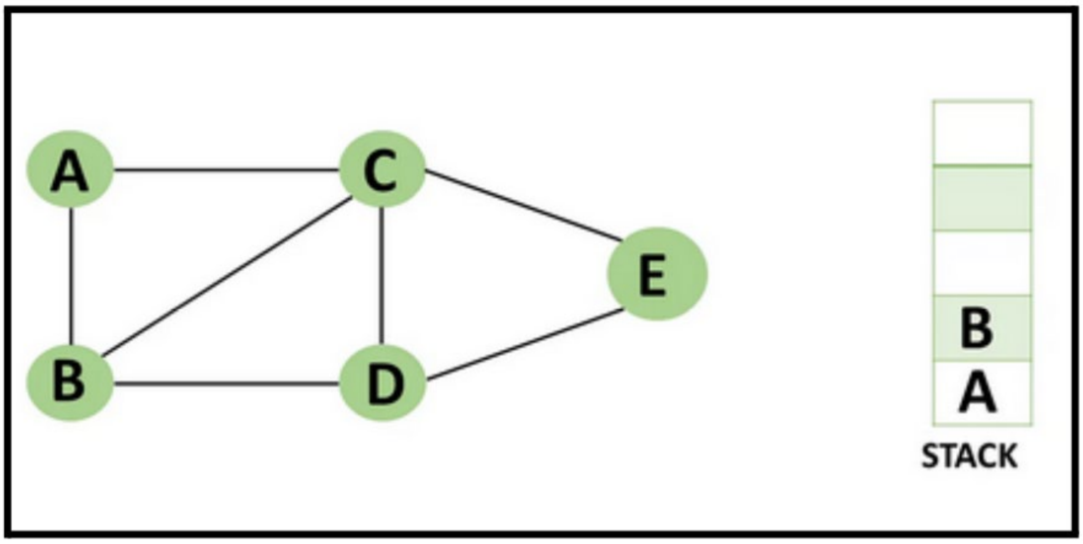
Step 1: Mark vertex A as a visited source node by selecting it as a source node.

● You should push vertex A to the top of the stack.



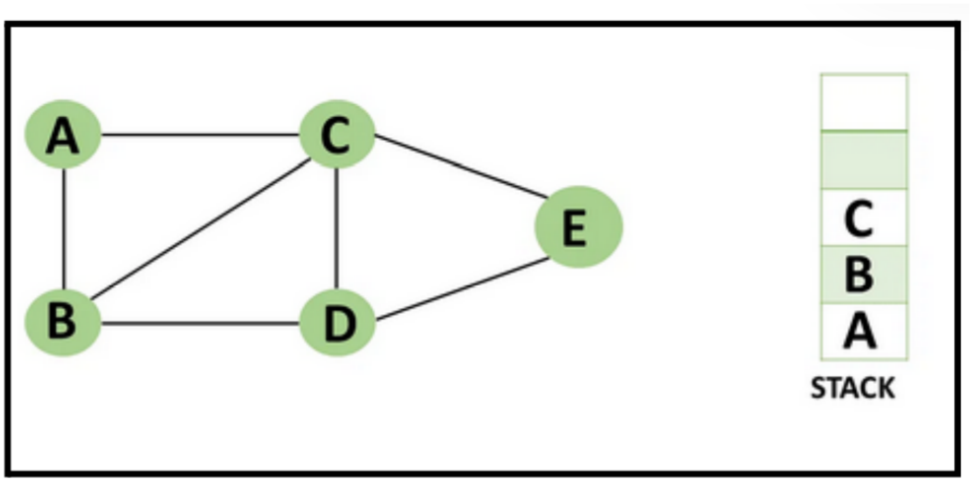
Step 2: Any nearby unvisited vertex of vertex A, say B, should be visited.

● You should push vertex B to the top of the stack.



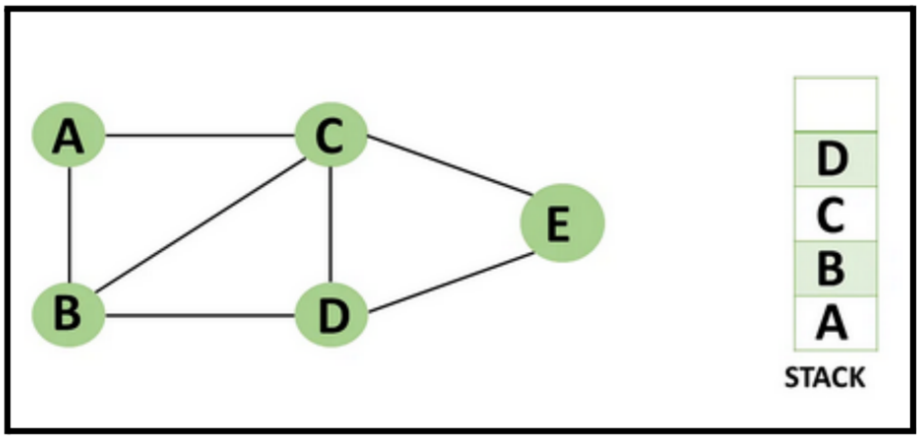
Step 3: From vertex C and D, visit any adjacent unvisited vertices of vertex B. Imagine you have chosen vertex C, and you want to make C a visited vertex.

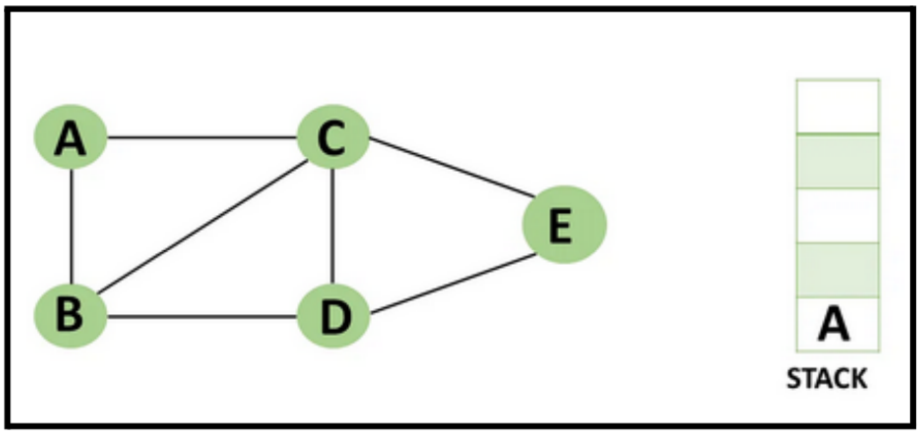
● Vertex C is pushed to the top of the stack.



Step 4: You can visit any nearby unvisited vertices of vertex C; you need to select vertex D and designate it as a visited vertex.

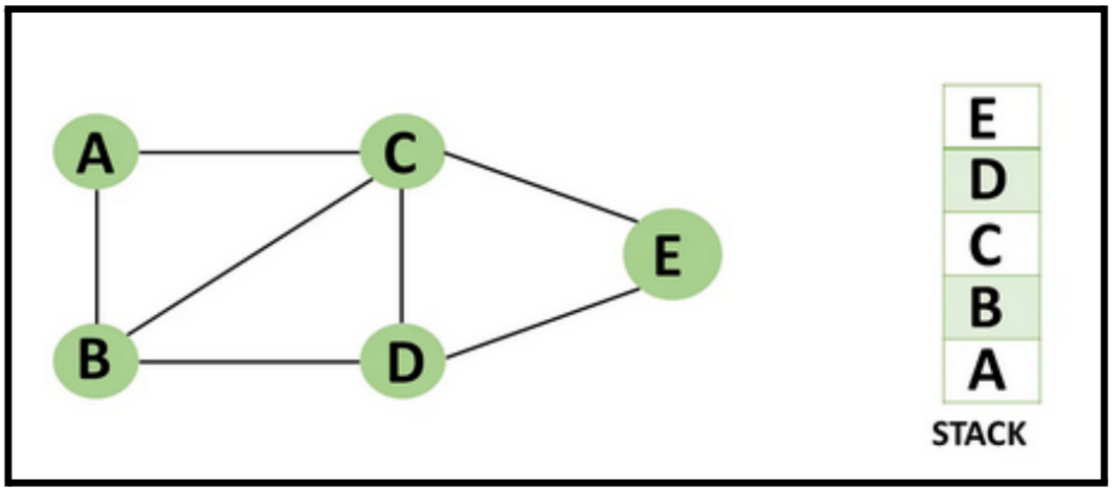
● Vertex D is pushed to the top of the stack.



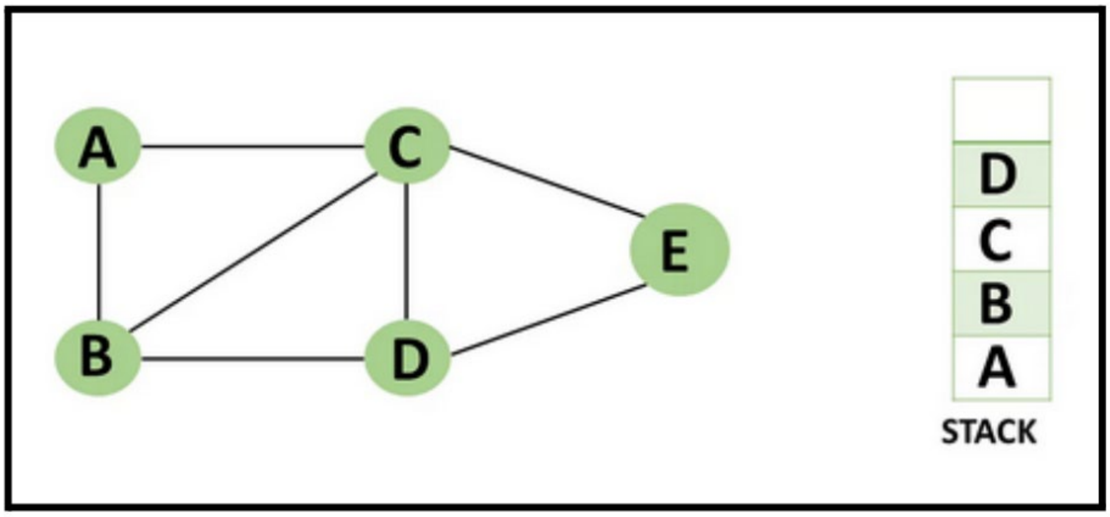


Step 5: Vertex E is the lone unvisited adjacent vertex of vertex D, thus marking it as visited.

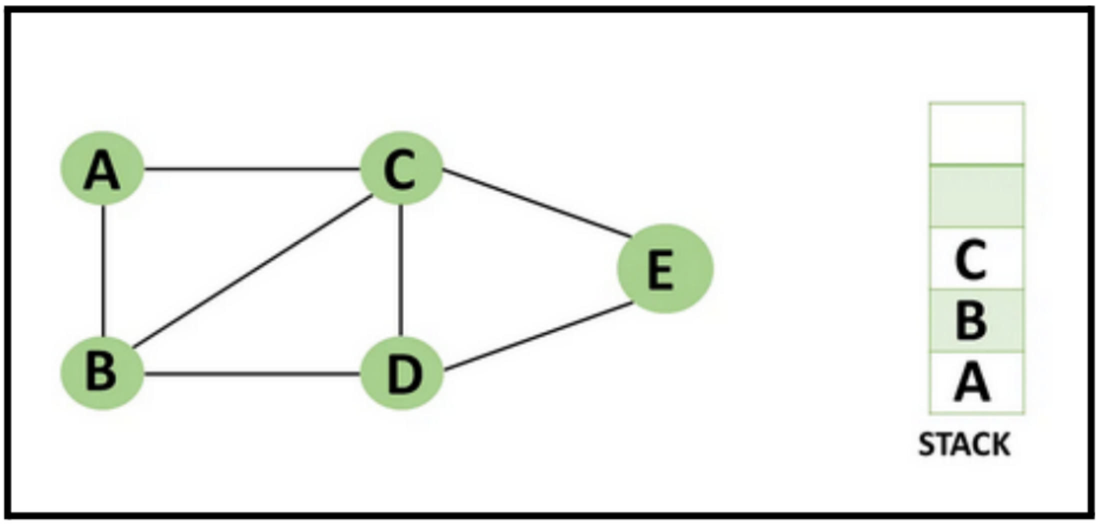
●Vertex E should be pushed to the top of the stack.



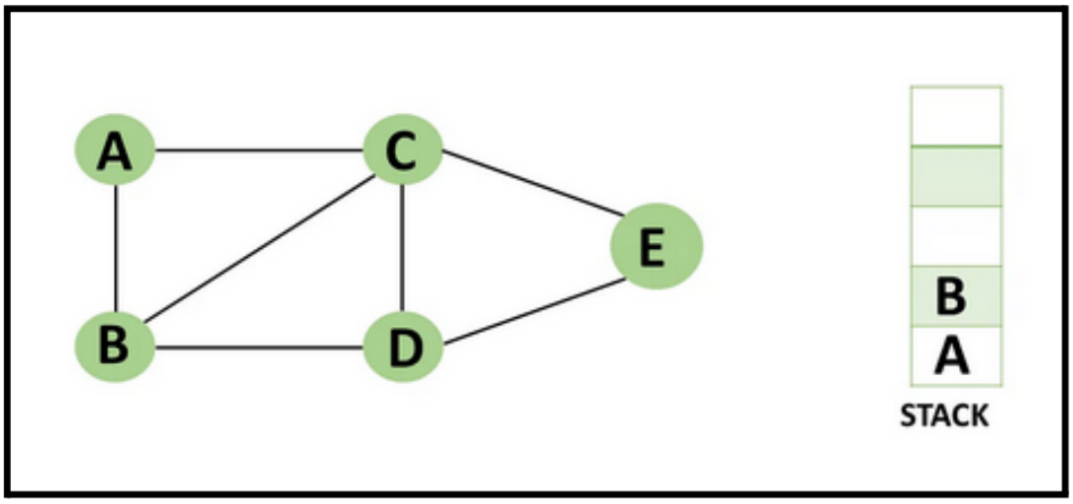
Step 6: Vertex E's nearby vertices, namely vertex C and D have been visited, pop vertex E from the stack.



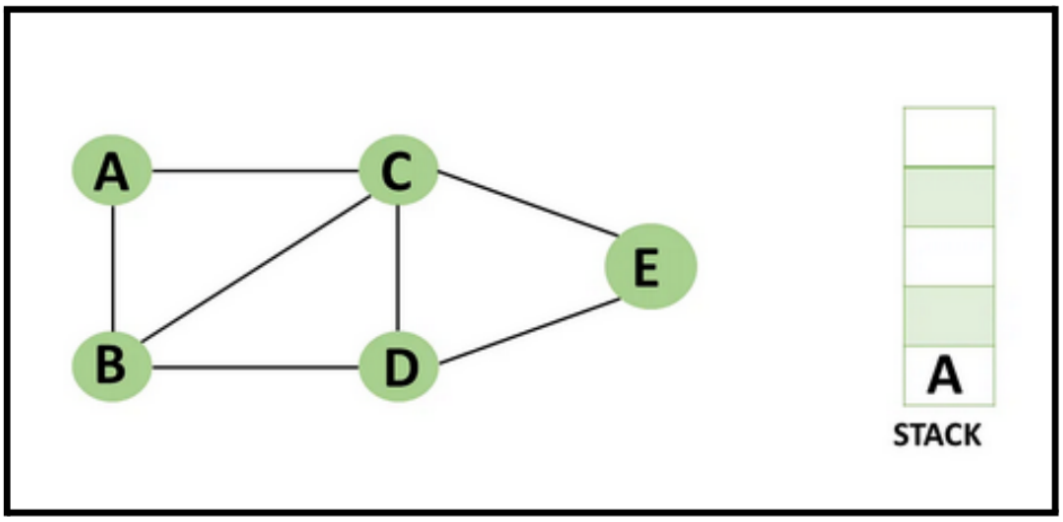
Step 7: Now that all of vertex D's nearby vertices, namely vertex B and C, have been visited, pop vertex D from the stack.



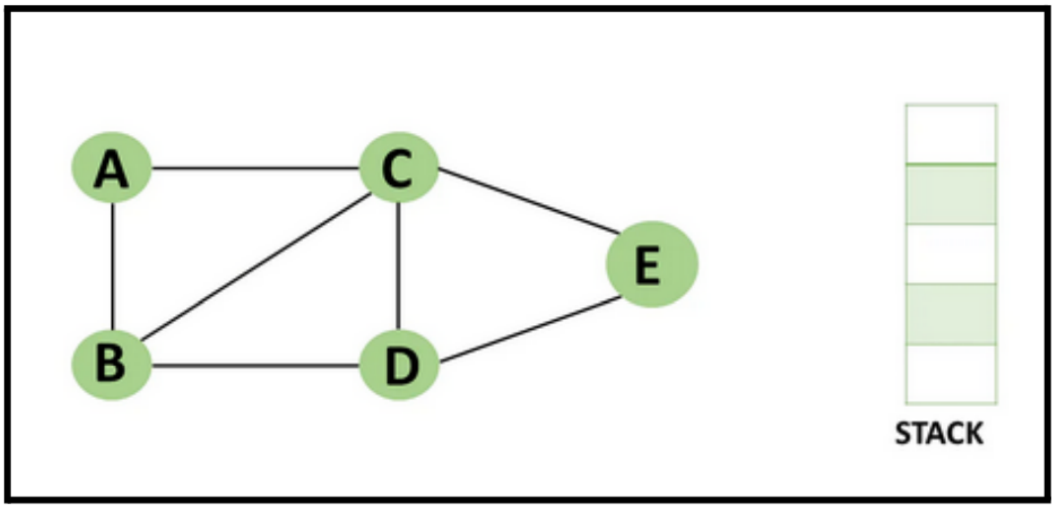
Step 8: Similarly, vertex C's adjacent vertices have already been visited; therefore, pop it from the stack.



Step 9: There is no more unvisited adjacent vertex of b, thus pop it from the stack.



Step 10: All the nearby vertices of Vertex A, B, and C, have already been visited, so pop vertex A from the stack as well.



Algorithm:

Parallel Breadth First Search (BFS):

Initialize:

- Create a queue data structure to store nodes to be visited.
- Mark all nodes as not visited.
- Enqueue the starting node into the queue and mark it as visited.

Parallel Exploration:

While the queue is not empty:

- Use OpenMP parallel constructs to parallelize the exploration of nodes at each level of the BFS tree.
- Each thread dequeues a node from the queue and explores its neighbours in parallel.
- Mark each neighbour node as visited and enqueue it into the queue if it has not been visited before.

Terminate:

- When all nodes have been visited, terminate the BFS algorithm.

Parallel Depth First Search (DFS):

Initialize:

- Create a stack data structure to store nodes to be visited.
- Mark all nodes as not visited.
- Push the starting node onto the stack and mark it as visited.

Parallel Exploration:

While the stack is not empty:

- Use OpenMP parallel constructs to parallelize the exploration of nodes at each level of the DFS tree.
- Each thread pops a node from the stack and explores its neighbours in parallel.
- Mark each neighbour's node as visited and push it onto the stack if it has not been visited before.

Terminate:

- When all nodes have been visited, terminate the DFS algorithm.

Concept of OpenMP:

● OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.

● OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

● OpenMP is widely used in scientific computing, engineering, and other fields that require high-performance computing. It is supported by most modern compilers and is available on a wide range of platforms, including desktops, servers, and supercomputers.

Conclusion:

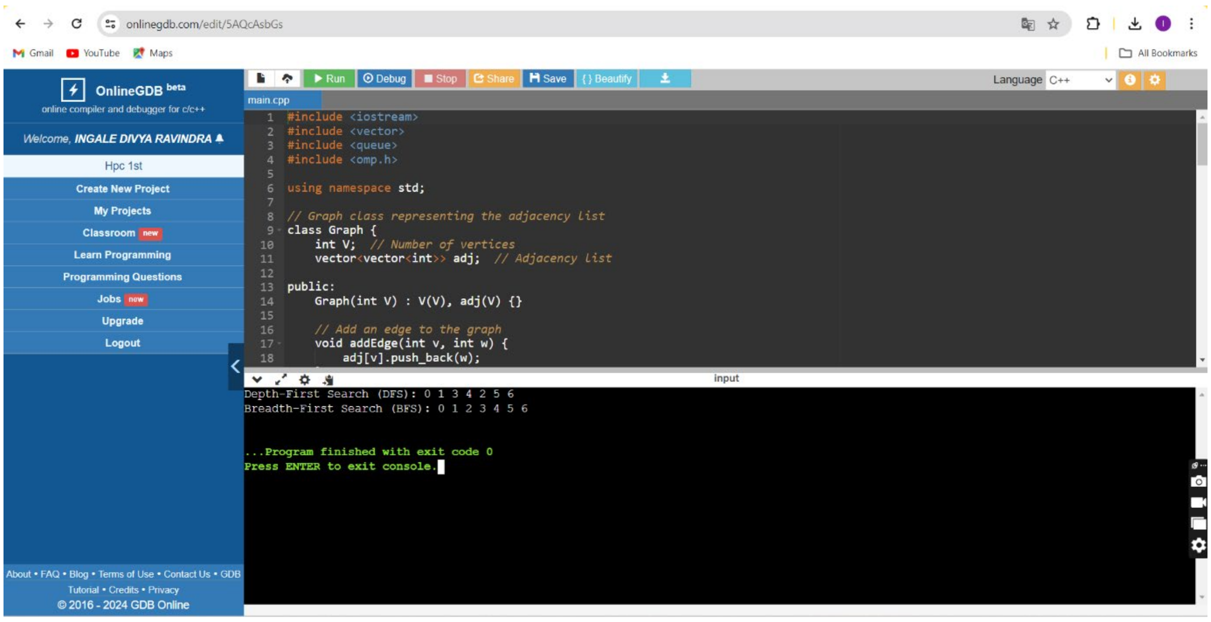
In this way we have implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP.

CODE:

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>

using namespace std;

// Graph class representing the adjacency list
class Graph {
   int V;  // Number of vertices
   vector<vector<int>> adj;  // Adjacency list

public:
   Graph(int V) : V(V), adj(V) {}

   // Add an edge to the graph
   void addEdge(int v, int w) {
      adj[v].push_back(w);
   }

   // Parallel Depth-First Search
   void parallelDFS(int startVertex) {
      vector<bool> visited(V, false);
      parallelDFSUtil(startVertex, visited);
   }

   // Parallel DFS utility function
   void parallelDFSUtil(int v, vector<bool>& visited) {
      visited[v] = true;
      cout << v << " ";
```

```
    #pragma omp parallel for
    for (int i = 0; i < adj[v].size(); ++i) {
        int n = adj[v][i];
        if (!visited[n])
            parallelDFSUtil(n, visited);
    }
}

// Parallel Breadth-First Search
void parallelBFS(int startVertex) {
    vector<bool> visited(V, false);
    queue<int> q;

    visited[startVertex] = true;
    q.push(startVertex);

    while (!q.empty()) {
        int v = q.front();
        q.pop();
        cout << v << " ";

        #pragma omp parallel for
        for (int i = 0; i < adj[v].size(); ++i) {
            int n = adj[v][i];
            if (!visited[n]) {
                visited[n] = true;
                q.push(n);
            }
        }
    }
    }
};

int main() {
    // Create a graph
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);

    /*
        0 -------->1
        |         / \
        |        /   \
        |       /     \
        v       v       v
        2 ----> 3       4
        |       |
        |       |
        v       v
        5       6
    */

    cout << "Depth-First Search (DFS): ";
    g.parallelDFS(0);
    cout << endl;

    cout << "Breadth-First Search (BFS): ";
    g.parallelBFS(0);
    cout << endl;


    return 0;
}
```

OUTPUT:



```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>

using namespace std;

// Graph class representing the adjacency list
class Graph {
    int V;  // Number of vertices
    vector<vector<int>> adj;  // Adjacency list

public:
    Graph(int V) : V(V), adj(V) {}

    // Add an edge to the graph
    void addEdge(int v, int w) {
        adj[v].push_back(w);
```

```
Depth-First Search (DFS): 0 1 3 4 2 5 6
Breadth-First Search (BFS): 0 1 2 3 4 5 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Assignment – 2

Title: Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Problem Statement: Design and implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Objectives: Students should be able to Write a program to implement Parallel Bubble Sort and Merge Sort and can measure the performance of sequential and parallel algorithms.

Theory:

Bubble Sort: Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It is called "bubble" sort because the algorithm moves the larger elements towards the end of the array in a manner that resembles the rising of bubbles in a liquid.

The basic algorithm of Bubble Sort is as follows:

1. Start at the beginning of the array.

2. Compare the first two elements. If the first element is greater than the second element, swap them.

3. Move to the next pair of elements and repeat step 2.

4. Continue the process until the end of the array is reached.

5. If any swaps were made in step 2-4, repeat the process from step 1.

The time complexity of Bubble Sort is O(n^2), which makes it inefficient for large lists. However, it has the advantage of being easy to understand and implement, and it is useful for educational purposes and for sorting small datasets.

Bubble Sort has limited practical use in modern software development due to its inefficient time complexity of O(n^2) which makes it unsuitable for sorting large datasets. However, Bubble Sort has some advantages and use cases that make it a valuable algorithm to understand, such as:

1. Simplicity: Bubble Sort is one of the simplest sorting algorithms, and it is easy to understand and implement. It can be used to introduce the concept of sorting to beginners and as a basis for more complex sorting algorithms.

2. Educational purposes: Bubble Sort is often used in academic settings to teach the principles of sorting algorithms and to help students understand how algorithms work.

3. Small datasets: For very small datasets, Bubble Sort can be an efficient sorting algorithm, as its overhead is relatively low.

4. Partially sorted datasets: If a dataset is already partially sorted, Bubble Sort can be very efficient. Since Bubble Sort only swaps adjacent elements that are in the wrong order, it has a low number of operations for a partially sorted dataset.

5. Performance optimization: Although Bubble Sort itself is not suitable for sorting large datasets, some of its techniques can be used in combination with other sorting algorithms to optimize their performance. For example, Bubble Sort can be used to optimize the performance of Insertion Sort by reducing the number of comparisons needed.

How Parallel Bubble Sort Work:

● Parallel Bubble Sort is a modification of the classic Bubble Sort algorithm that takes advantage of

parallel processing to speed up the sorting process.

● In parallel Bubble Sort, the list of elements is divided into multiple sub lists that are sorted

concurrently by multiple threads. Each thread sorts its sub list using the regular Bubble Sort

algorithm. When all sub lists have been sorted, they are merged to form the final sorted

list.

● The parallelization of the algorithm is achieved using OpenMP, a programming API that supports

parallel processing in C++, Fortran, and other programming languages. OpenMP provides a set of

compiler directives that allow developers to specify which parts of the code can be executed in parallel.

● In the parallel Bubble Sort algorithm, the main loop that iterates over the list of elements is divided into multiple iterations that are executed concurrently by multiple threads. Each thread sorts a subset of the list, and the threads synchronize their work at the end of each iteration to ensure that the elements are properly ordered.

● Parallel Bubble Sort can provide a significant speedup over the regular Bubble Sort algorithm, especially when sorting large datasets on multi-core processors. However, the speedup is limited by the overhead of thread creation and synchronization, and it may not be worth the effort for small datasets or when using a single-core processor.

Merge Sort: Merge sort is a sorting algorithm that uses a divide-and-conquer approach to sort an array or a list of elements. The algorithm works by recursively dividing the input array into two halves, sorting each half, and then merging the sorted halves to produce a sorted output.

The merge sort algorithm can be broken down into the following steps:

1. Divide the input array into two halves.

2. Recursively sort the left half of the array.

3. Recursively sort the right half of the array.

4. Merge the two sorted halves into a single sorted output array.

● The merging step is where the bulk of the work happens in merge sort. The algorithm compares the first elements of each sorted half, selects the smaller element, and appends it to the output array. This process continues until all elements from both halves have been appended to the output array.

● The time complexity of merge sort is O(n log n), which makes it an efficient sorting algorithm for large input arrays. However, merge sort also requires additional memory to store the output array, which can make it less suitable for use with limited memory resources.

● In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

● One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

● Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

How Parallel Merge Sort Work:

● Parallel merge sort is a parallelized version of the merge sort algorithm that takes advantage of multiple processors or cores to improve its performance. In parallel merge sort, the input array is divided into smaller subarrays, which are sorted in parallel using multiple processors or cores. The sorted subarrays are then merged together in parallel to produce the final sorted output.

● The parallel merge sort algorithm can be broken down into the following steps:

● Divide the input array into smaller subarrays.

● Assign each subarray to a separate processor or core for sorting.

● Sort each subarray in parallel using the merge sort algorithm.

● Merge the sorted subarrays together in parallel to produce the final sorted output.

● The merging step in parallel merge sort is performed in a similar way to the merging step in the sequential merge sort algorithm. However, because the subarrays are sorted in parallel, the merging step can also be performed in parallel using multiple processors or cores. This can significantly reduce the time required to merge the sorted subarrays and produce the final output.

● Parallel merge sort can provide significant performance benefits for large input arrays with many elements, especially when running on hardware with multiple processors or cores. However, it also requires additional overhead to manage the parallelization, and may not always provide performance improvements for smaller input sizes or when run on hardware with limited parallel processing capabilities.

How to measure the performance of sequential and parallel algorithms:

There are several metrics that can be used to measure the performance of sequential and parallel merge sort algorithms:

1. Execution time: Execution time is the amount of time it takes for the algorithm to complete its sorting operation. This metric can be used to compare the speed of sequential and parallel merge sort algorithms.

2. Speedup: Speedup is the ratio of the execution time of the sequential merge sort algorithm to the execution time of the parallel merge sort algorithm. A speedup of greater than 1 indicates that the parallel algorithm is faster than the sequential algorithm.

3. Efficiency: Efficiency is the ratio of the speedup to the number of processors or cores used in the parallel algorithm. This metric can be used to determine how well the parallel algorithm is utilizing the available resources.

4. Scalability: Scalability is the ability of the algorithm to maintain its performance as the input size and number of processors or cores increase. A scalable algorithm will maintain a consistent speedup and efficiency as more resources are added.

Concept of OpenMP:

● OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.

● OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

Conclusion: In this way we can implement Bubble Sort and Merge Sort in parallel way using OpenMP also come to know how to how to measure performance of serial and parallel algorithm.

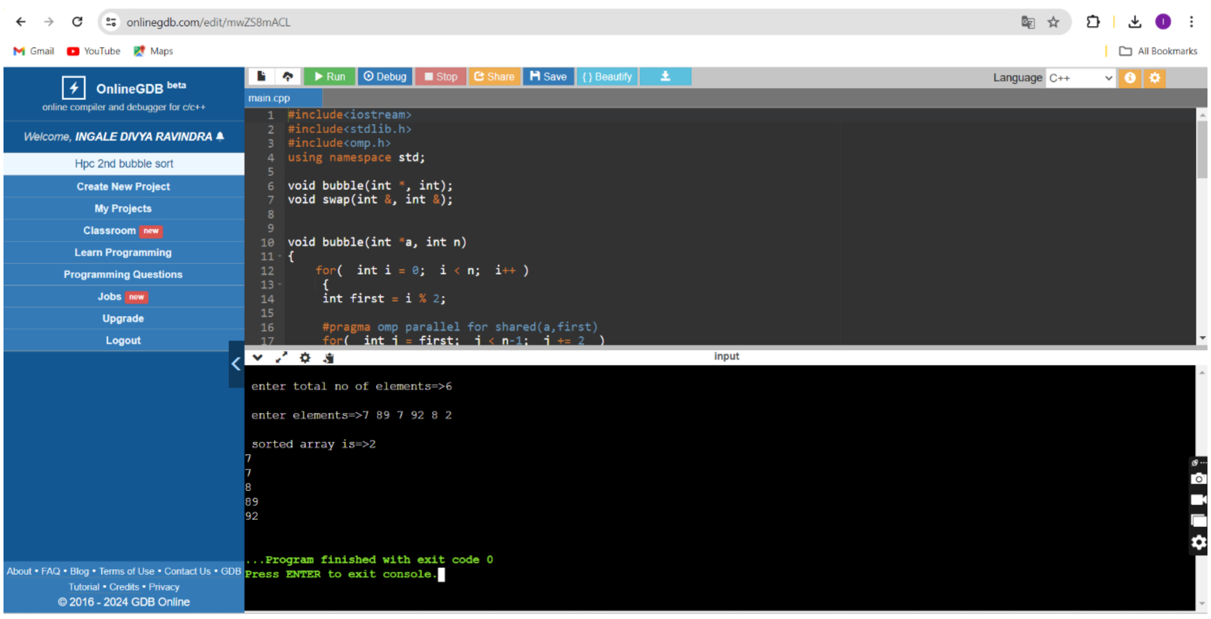
BUBBLE SORT:

CODE:

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

void bubble(int *, int);
void swap(int &, int &);



void bubble(int *a, int n)
{
   for(  int i = 0;  i < n;  i++ )
    {
          int first = i % 2;

          #pragma omp parallel for shared(a,first)
          for(  int j = first;  j < n-1;  j += 2  )
            {
                    if(  a[ j ]  >  a[ j+1 ]  )
                     {
                            swap(  a[ j ],  a[ j+1 ]  );
                     }
                      }
    }
}



void swap(int &a, int &b)
{
   int test;
   test=a;
   a=b;
   b=test;

}

int main()
{

   int *a,n;
   cout<<"\n enter total no of elements=>";
   cin>>n;
   a=new int[n];
   cout<<"\n enter elements=>";
   for(int i=0;i<n;i++)
   {
          cin>>a[i];
   }

   bubble(a,n);

   cout<<"\n sorted array is=>";
   for(int i=0;i<n;i++)
   {
          cout<<a[i]<<endl;
   }


return 0;
```

}

OUTPUT:



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

void bubble(int *, int);
void swap(int &, int &);


void bubble(int *a, int n)
{
    for(  int i = 0;  i < n;  i++ )
     {
     int first = i % 2;

     #pragma omp parallel for shared(a,first)
     for(  int j = first;  j < n-1;  j += 2  )
```

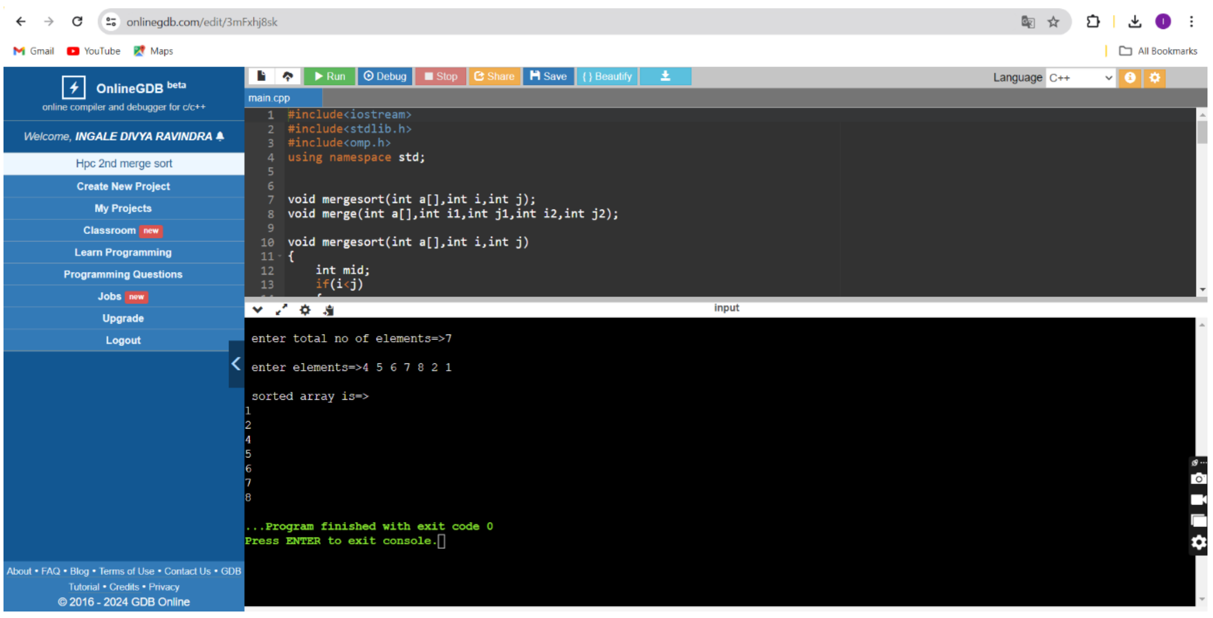
```
enter total no of elements=>6

enter elements=>7 89 7 92 8 2

sorted array is=>2
7
7
8
89
92


...Program finished with exit code 0
Press ENTER to exit console.
```

MERGE SORT:

CODE:

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



void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

void mergesort(int a[],int i,int j)
{
        int mid;
        if(i<j)
        {
        mid=(i+j)/2;

        #pragma omp parallel sections
        {

        #pragma omp section
        {
                  mergesort(a,i,mid);
        }

        #pragma omp section
        {
                  mergesort(a,mid+1,j);
        }
        }

        merge(a,i,mid,mid+1,j);
        }

}

void merge(int a[],int i1,int j1,int i2,int j2)
{
        int temp[1000];
        int i,j,k;
        i=i1;
        j=i2;
        k=0;

        while(i<=j1 && j<=j2)
        {
        if(a[i]<a[j])
        {
        temp[k++]=a[i++];
        }
        else
        {
        temp[k++]=a[j++];
    }
        }

        while(i<=j1)
        {
        temp[k++]=a[i++];
        }

        while(j<=j2)
        {
        temp[k++]=a[j++];
        }

        for(i=i1,j=0;i<=j2;i++,j++)
        {
        a[i]=temp[j];
        }
}


int main()
{
        int *a,n,i;
        cout<<"\n enter total no of elements=>";
        cin>>n;
        a= new int[n];

        cout<<"\n enter elements=>";
        for(i=0;i<n;i++)
        {
        cin>>a[i];
        }
//        start=.......
//#pragma omp…..
        mergesort(a, 0, n-1);
//          stop…….
        cout<<"\n sorted array is=>";
        for(i=0;i<n;i++)
        {
        cout<<"\n"<<a[i];
        }
        // Cout<<Stop-Start
        return 0;
}
```

OUTPUT:



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


void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

void mergesort(int a[],int i,int j)
{
    int mid;
    if(i<j)
```

```
enter total no of elements=>7

enter elements=>4 5 6 7 8 2 1

sorted array is=>
1
2
4
5
6
7
8

...Program finished with exit code 0
Press ENTER to exit console.
```

Assignment – 3

Title: Implement Min, Max, Sum, and Average operations using Parallel Reduction.

Problem Statement: Design and Implement Min, Max, Sum and Average operations using Parallel Reduction.

Objectives: To understand the concept of parallel reduction and how it can be used to perform basic mathematical operations on given data sets.

Theory:

Parallel Reduction.

Here's a function-wise manual on how to understand and run the sample C++ program that demonstrates

how to implement Min, Max, Sum, and Average operations using parallel reduction.

1. Min_Reduction function

•The function takes in a vector of integers as input and finds the minimum value in the vector using parallel reduction.

• The OpenMP reduction clause is used with the "min" operator to find the minimum value across all threads.

• The minimum value found by each thread is reduced to the overall minimum value of the entire array.

• The final minimum value is printed to the console.

2. Max_Reduction function

• The function takes in a vector of integers as input and finds the maximum value in the vector using parallel reduction.

• The OpenMP reduction clause is used with the "max" operator to find the maximum value across all threads.

• The maximum value found by each thread is reduced to the overall maximum value of the entire array.

• The final maximum value is printed to the console.

3. Sum_Reduction function

• The function takes in a vector of integers as input and finds the sum of all the values in the vector using parallel reduction.

• The OpenMP reduction clause is used with the "+" operator to find the sum across all threads.

• The sum found by each thread is reduced to the overall sum of the entire array.

• The final sum is printed to the console.

4. Average_Reduction function

• The function takes in a vector of integers as input and finds the average of all the values in the vector using parallel reduction.

• The OpenMP reduction clause is used with the "+" operator to find the sum across all threads.

• The sum found by each thread is reduced to the overall sum of the entire array.

• The final sum is divided by the size of the array to find the average.

• The final average value is printed to the console.

5. Main Function

• The function initializes a vector of integers with some values.

• The function calls the min_reduction, max_reduction, sum_reduction, and

average_reduction functions on the input vector to find the corresponding values.

• The final minimum, maximum, sum, and average values are printed to the console.

6. Compiling and running the program

Compile the program: You need to use a C++ compiler that supports OpenMP, such as g++ or clang. Open a terminal and navigate to the directory where your program is saved. Then, compile the program using the following command:

```
$ g++ -fopenmp program.cpp -o program
```

This command compiles your program and creates an executable file named "program". The "-fopenmp" flag tells the compiler to enable OpenMP.

Run the program: To run the program, simply type the name of the executable file in the terminal and press Enter:

```
$ ./program
```

Conclusion: We have implemented the Min, Max, Sum, and Average operations using parallel reduction in C++ with OpenMP. Parallel reduction is a powerful technique that allows us to perform these operations on large arrays more efficiently by dividing the work among multiple threads running in parallel. We presented a code example that demonstrates the implementation of these operations using parallel reduction in C++ with OpenMP. We also provided a manual for running OpenMP programs on the Ubuntu platform.

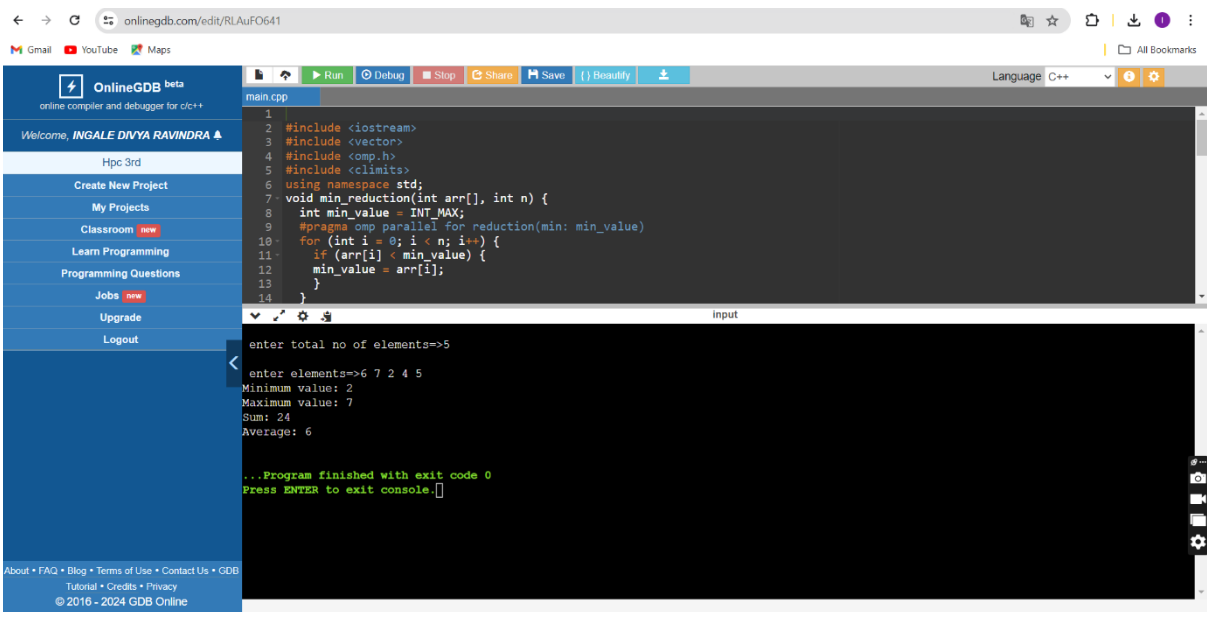
CODE:

```
#include <iostream>

#include <vector>

#include <omp.h>

#include <climits>

using namespace std;
```

```
void min_reduction(int arr[], int n) {
  int min_value = INT_MAX;
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < n; i++) {
            if (arr[i] < min_value) {
            min_value = arr[i];
            }
  }
  cout << "Minimum value: " << min_value << endl;
}

void max_reduction(int arr[], int n) {
  int max_value = INT_MIN;
  #pragma omp parallel for reduction(max: max_value)
  for (int i = 0; i < n; i++) {
            if (arr[i] > max_value) {
            max_value = arr[i];
            }
  }
  cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(int arr[], int n) {
  int sum = 0;
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < n; i++) {
            sum += arr[i];
  }
  cout << "Sum: " << sum << endl;
}
```

```
void average_reduction(int arr[], int n) {
  int sum = 0;
  #pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < n; i++) {
          sum += arr[i];
  }
  cout << "Average: " << (double)sum / (n-1) << endl;
}

int main() {
    int *arr,n;
    cout<<"\n enter total no of elements=>";
    cin>>n;
    arr=new int[n];
    cout<<"\n enter elements=>";
    for(int i=0;i<n;i++)
    {
           cin>>arr[i];
    }

//   int arr[] = {5, 2, 9, 1, 7, 6, 8, 3, 4};
//   int n = size(arr);

  min_reduction(arr, n);
  max_reduction(arr, n);
  sum_reduction(arr, n);
  average_reduction(arr, n);
}
```

OUTPUT:



```cpp

#include <iostream>
#include <vector>
#include <omp.h>
#include <climits>
using namespace std;
void min_reduction(int arr[], int n) {
  int min_value = INT_MAX;
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < n; i++) {
    if (arr[i] < min_value) {
    min_value = arr[i];
    }
  }
```

```
 enter total no of elements=>5

 enter elements=>6 7 2 4 5
Minimum value: 2
Maximum value: 7
Sum: 24
Average: 6


...Program finished with exit code 0
Press ENTER to exit console.
```

.

Assignment – 4

Title: Write a CUDA Program for:

1. Addition of two large vectors

2. Matrix Multiplication using CUDA C

Problem Statement: Write a CUDA Program for:

1. Addition of two large vectors

2. Matrix Multiplication using CUDA C

Objectives: Students should be able to perform CUDA Program for Addition of two large vectors.

Theory:

CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows developers to use the power of NVIDIA graphics processing units (GPUs) to accelerate computation tasks in various applications, including scientific computing, machine learning, and computer vision. CUDA provides a set of programming APIs, libraries, and tools that enable developers to write and execute parallel code on NVIDIA GPUs. It supports popular programming languages like C, C++, and Python, and provides a simple programming model that abstracts away much of the low-level details of GPU architecture. Using CUDA, developers can exploit the massive parallelism and high computational power of GPUs to accelerate computationally intensive tasks, such as matrix operations, image processing, and deep learning. CUDA has become an important tool for scientific research and is widely used in fields like physics, chemistry, biology, and engineering.

Steps for Addition of two large vectors using CUDA

1. Define the size of the vectors: In this step, you need to define the size of the vectors that you want to add. This will determine the number of threads and blocks you will need to use to parallelize the addition operation.

2. Allocate memory on the host: In this step, you need to allocate memory on the host for the two vectors that you want to add and for the result vector. You can use the C malloc function to allocate memory.

3. Initialize the vectors: In this step, you need to initialize the two vectors that you want to add on the host. You can use a loop to fill the vectors with data.

4. Allocate memory on the device: In this step, you need to allocate memory on the device for the two vectors that you want to add and for the result vector. You can use the CUDA function cudaMalloc to allocate memory.

5. Copy the input vectors from host to device: In this step, you need to copy the two input vectors from the host to the device memory. You can use the CUDA function cudaMemcpy to copy the vectors.

6. Launch the kernel: In this step, you need to launch the CUDA kernel that will perform the addition operation. The kernel will be executed by multiple threads in parallel. You can use the <<>> syntax to specify the number of blocks and threads to use.

7. Copy the result vector from device to host: In this step, you need to copy the result vector from the device memory to the host memory. You can use the CUDA function cudaMemcpy to copy the result vector.

8. Free memory on the device: In this step, you need to free the memory that was allocated on the device. You can use the CUDA function cudaFree to free the memory.

9. Free memory on the host: In this step, you need to free the memory that was allocated on the host. You can use the C free function to free the memory.

Execution of Program over CUDA Environment Here are the steps to run a CUDA program for adding two large vectors: 1. Install CUDA Toolkit: First, you need to install the CUDA Toolkit on your system. You can download the CUDA Toolkit from the NVIDIA website and follow the installation instructions provided.

2. Set up CUDA environment: Once the CUDA Toolkit is installed, you need to set up the CUDA environment on your system. This involves setting the PATH and LD_LIBRARY_PATH environment variables to the appropriate directories.

3. Write the CUDA program: You need to write a CUDA program that performs the addition of two large vectors. You can use a text editor to write the program and save it with a .cu extension.

4. Compile the CUDA program: You need to compile the CUDA program using the nvcc compiler that comes with the CUDA Toolkit.

The command to compile the program is:

```
nvcc -o program_name program_name.cu
```

5. This will generate an executable program named program_name. Run the CUDA program: Finally, you can run the CUDA program by executing the executable file generated in the previous step. The command to run the program is:

```
./program_name
```

This will execute the program and perform the addition of two large vectors.

Steps for Matrix Multiplication using CUDA Here are the steps for implementing matrix multiplication using CUDA C:

1. Matrix Initialization: The first step is to initialize the matrices that you want to multiply. You can use standard C or CUDA functions to allocate memory for the matrices and initialize their values. The matrices are usually represented as 2D arrays.

2. Memory Allocation: The next step is to allocate memory on the host and the device for the matrices. You can use the standard C malloc function to allocate memory on the host and the CUDA function cudaMalloc() to allocate memory on the device.

3. Data Transfer: The third step is to transfer data between the host and the device. You can use the CUDA function cudaMemcpy() to transfer data from the host to the device or vice versa.

4. Kernel Launch: The fourth step is to launch the CUDA kernel that will perform the matrix multiplication on the device. You can use the <<>> syntax to specify the number of blocks and threads to use. Each thread in the kernel will compute one element of the output matrix.

5. Device Synchronization: The fifth step is to synchronize the device to ensure that all kernel executions have completed before proceeding. You can use the CUDA function cudaDeviceSynchronize() to synchronize the device.

6. Data Retrieval: The sixth step is to retrieve the result of the computation from the device to the host. You can use the CUDA function cudaMemcpy() to transfer data from the device to the host.

7. Memory Deallocation: The final step is to deallocate the memory that was allocated on the host and the device. You can use the C free function to deallocate memory on the host and the CUDA function cudaFree() to deallocate memory on the device.

Execution of Program over CUDA Environment

1. Install CUDA Toolkit: First, you need to install the CUDA Toolkit on your system. You can download the CUDA Toolkit from the NVIDIA website and follow the installation instructions provided.

2. Set up CUDA environment: Once the CUDA Toolkit is installed, you need to set up the CUDA environment on your system. This involves setting the PATH and LD_LIBRARY_PATH environment variables to the appropriate directories.

3. Write the CUDA program: You need to write a CUDA program that performs the addition of two large vectors. You can use a text editor to write the program and save it with a .cu extension.

4. Compile the CUDA program: You need to compile the CUDA program using the nvcc compiler that comes with the CUDA Toolkit. The command to compile the program is:

```
nvcc -o program_name program_name.cu
```

5. This will generate an executable program named program_name. Run the CUDA program: Finally, you can run the CUDA program by executing the executable file generated in the previous step. The command to run the program is:

```
./program_name
```

This will execute the program and perform the Matrix Multiplication using CUDA.

Conclusion: In this way we can implement the addition of two large vectors and the Matrix Multiplication using CUDA.

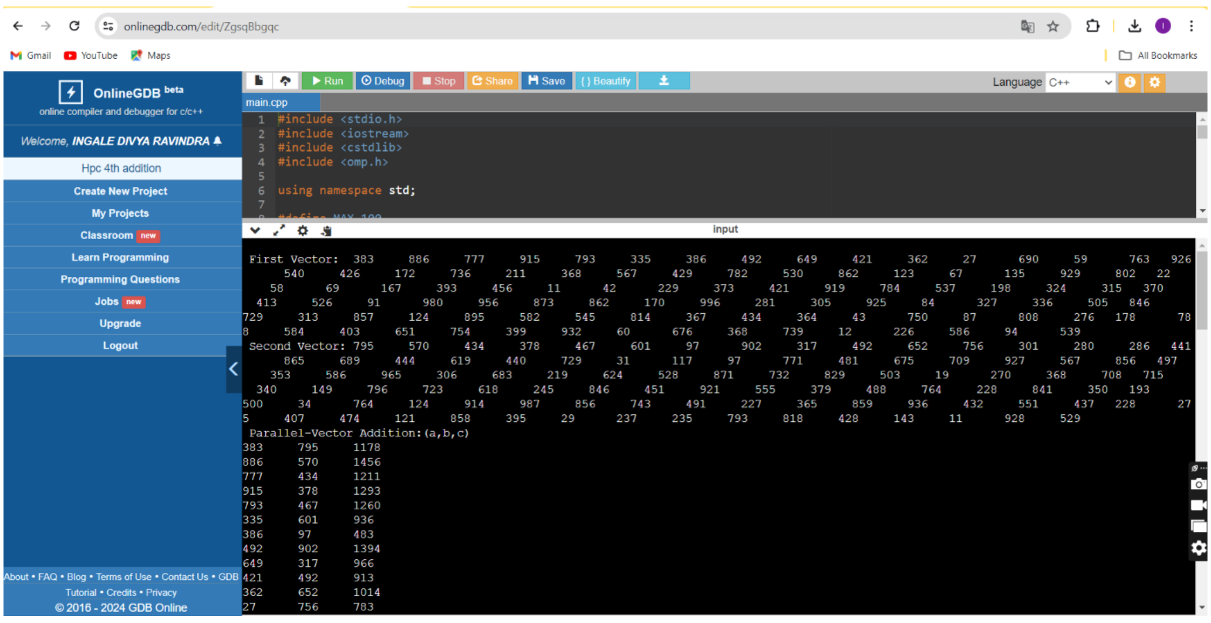
Program for Addition:

```
#include <stdio.h>

#include <iostream>

#include <cstdlib>
```

```
#include <omp.h>

using namespace std;

#define MAX 100

int main()
{
   int a[MAX], b[MAX], c[MAX], i;

   printf("\n First Vector:\t");

#pragma omp parallel for
   for (i = 0; i < MAX; i++)
   {
      a[i] = rand() % 1000;
   }

   for (i = 0; i < MAX; i++)
   {
      printf("%d\t", a[i]);
   }

   printf("\n Second Vector:\t");

#pragma omp parallel for
   for (i = 0; i < MAX; i++)
   {
      b[i] = rand() % 1000;
   }
```

```
    for (i = 0; i < MAX; i++)
    {
        printf("%d\t", b[i]);
    }

    printf("\n Parallel-Vector Addition:(a,b,c)\t");

#pragma omp parallel for
    for (i = 0; i < MAX; i++)
    {
        c[i] = a[i] + b[i];
    }

    for (i = 0; i < MAX; i++)
    {
        printf("\n%d\t%d\t%d", a[i], b[i], c[i]);
    }

    return 0;
}
```

Output:



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

using namespace std;
```

```
 First Vector:  383     886     777     915     793     335     386     492     649     421     362     27      690     59      763   926
     540     426     172     736     211     368     567     429     782     530     862     123     67      135     929     802   22
   58      69      167     393     456     11      42      229     373     421     919     784     537     198     324     315   370
  413     526     91      980     956     873     862     170     996     281     305     925     84      327     336     505   846
729     313     857     124     895     582     545     814     367     434     364     43      750     87      808     276   178       78
8     584     403     651     754     399     932     60      676     368     739     12      226     586     94      539
 Second Vector: 795     570     434     378     467     601     97      902     317     492     652     756     301     280     286   441
     865     689     444     619     440     729     31      117     97      771     481     675     709     927     567     856   497
   353     586     965     306     683     219     624     528     871     732     829     503     19      270     368     708   715
  340     149     796     723     618     245     846     451     921     555     379     488     764     228     841     350   193
500     34      764     124     914     987     856     743     491     227     365     859     936     432     551     437   228       27
5     407     474     121     858     395     29      237     235     793     818     428     143     11      928     529
 Parallel-Vector Addition:(a,b,c)
383     795     1178
886     570     1456
777     434     1211
915     378     1293
793     467     1260
335     601     936
386     97      483
492     902     1394
649     317     966
421     492     913
362     652     1014
27      756     783
```

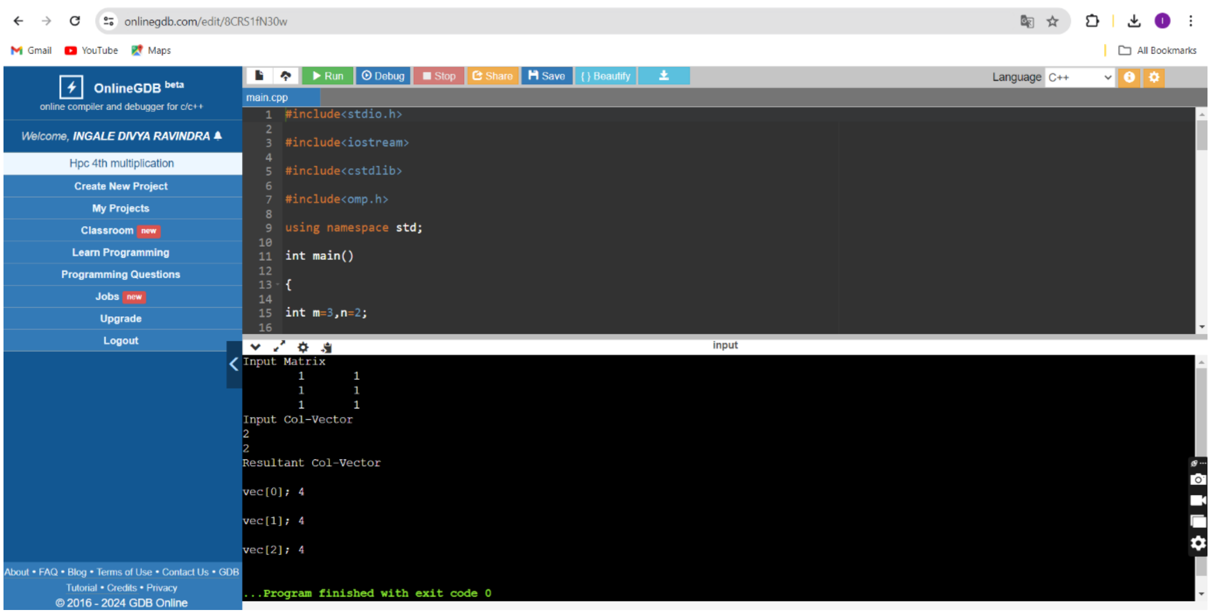
Program for Multiplication:

```
#include<stdio.h>

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

int main()

{

int m=3,n=2;

int mat[m][n], vec[n],out[m];

//matrix of size 3x2

for(int row=0;row<m;row++)

{

for(int col=0;col<n;col++)

{

mat[row][col]=1;
}
}

//display matrix

cout<<"Input Matrix"<<endl;

for(int row=0;row<m;row++)

{

for(int col=0;col<n;col++)

{

cout<<"\t"<<mat[row][col];

}

cout<<""""<<endl;

}

//column vector of size 2x1

for(int row=0;row<n;row++)

{

vec[row]=2;
```

```
}

//display vector
cout<<"Input Col-Vector"<<endl;

for(int row=0;row<n;row++)

{

cout<<vec[row]<<endl;

}
{

#pragma omp parallel for

for(int row=0;row<m;row++)

{

out[row]=0;

for(int col=0;col<n;col++)
{ out[row] +=mat[row][col]*vec[col];

}

}

}
```

```
//display resultant vector

cout<< "Resultant Col-Vector"<<endl;

for (int row=0; row<m;row++)

{

cout<<"\nvec["<<row<<"]; "<<out[row]<<endl;

}

return 0;

}
```

Output:



```
#include<stdio.h>

#include<iostream>

#include<cstdlib>

#include<omp.h>

using namespace std;

int main()

{

int m=3,n=2;
```

```
Input Matrix
        1       1
        1       1
        1       1
Input Col-Vector
2
2
Resultant Col-Vector

vec[0]; 4

vec[1]; 4

vec[2]; 4


...Program finished with exit code 0
```

Bharati Vidyapeeth's College of Engineering for Women, Pune-43

DEPARTMENT OF COMPUTER ENGINEERING

EXPERIMENT NO. 1

TITLE: Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .

NAME OF LABORATORY: LPV

DATE OF PERFORMANCE:

DATE OF SUBMISSION:

NAME OF STUDENT: Divya Ravindra Ingale

ROLL NO: 4225

SEMESTER: 8th Semester

YEAR: FOURTH YEAR

NAME OF FACULTY: Prof. Anjali Kadam

MARKS/ GRADE OBTAINED:

Signature of faculty

Bharati Vidyapeeth's College of Engineering for Women, Pune-43

DEPARTMENT OF COMPUTER ENGINEERING

EXPERIMENT NO. 2

TITLE: Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

NAME OF LABORATORY: LPV

DATE OF PERFORMANCE:

DATE OF SUBMISSION:

NAME OF STUDENT: Divya Ravindra Ingale

ROLL NO: 4225

SEMESTER: 8th Semester

YEAR: FOURTH YEAR

NAME OF FACULTY: Prof. Anjali Kadam

MARKS/ GRADE OBTAINED:

Signature of faculty

Bharati Vidyapeeth's College of Engineering for Women, Pune-43

DEPARTMENT OF COMPUTER ENGINEERING

EXPERIMENT NO. 3

TITLE: Implement Min, Max, Sum and Average operations using Parallel Reduction.

NAME OF LABORATORY: LPV

DATE OF PERFORMANCE:

DATE OF SUBMISSION:

NAME OF STUDENT: Divya Ravindra Ingale

ROLL NO: 4225

SEMESTER: 8th Semester

YEAR: FOURTH YEAR

NAME OF FACULTY: Prof. Anjali Kadam

MARKS/ GRADE OBTAINED:

Signature of faculty

Bharati Vidyapeeth's College of Engineering for Women, Pune-43

DEPARTMENT OF COMPUTER ENGINEERING

EXPERIMENT NO. 4

TITLE: Write a CUDA Program for : 1. Addition of two large vectors 2. Matrix Multiplication using CUDA C

NAME OF LABORATORY: LPV

DATE OF PERFORMANCE:

DATE OF SUBMISSION:

NAME OF STUDENT: Divya Ravindra Ingale

ROLL NO: 4225

SEMESTER: 8th Semester

YEAR: FOURTH YEAR

NAME OF FACULTY: Prof. Anjali Kadam

MARKS/ GRADE OBTAINED:

Signature of faculty

Bharati Vidyapeeth's College of Engineering for Women, Pune-43

DEPARTMENT OF COMPUTER ENGINEERING

EXPERIMENT NO. 5

TITLE: Project Report

NAME OF LABORATORY: LPV

DATE OF PERFORMANCE:

DATE OF SUBMISSION:

NAME OF STUDENT: Divya Ravindra Ingale

ROLL NO: 4225

SEMESTER: 8th Semester

YEAR: FOURTH YEAR

NAME OF FACULTY: Prof. Anjali Kadam

MARKS/ GRADE OBTAINED:

Signature of faculty