JSPM's
Bhivarabai Sawant Institute OF Technology and Research,
Wagholi, Pune.

Savitribai Phule Pune University (SPPU)
Fourth Year of Computer Engineering (2019 Course)

410255: Laboratory Practice V
Subject Teacher: Prof. Sonawane Vijay D.
Prof. Shivale Nitin M.

Term work: 50 Marks
Practical: 50 Marks

High Performance Computing (410250)
Deep Learning (410251)

Savitribai Phule Pune University
Fourth Year of Computer Engineering (2019 Course)
410255: Laboratory Practice V

Teaching Scheme Practical: 2 Hours/Week	Credit 01	Examination Scheme Term Work: 50 Marks Practical: 50 Marks

Companion Course: High Performance Computing (410250), Deep Learning(410251)

Course Objectives:

- **To understand and implement searching and sorting algorithms.**
- **To learn the fundamentals of GPU Computing in the CUDA environment.**
- **To illustrate the concepts of Artificial Intelligence/Machine Learning (AI/ML).**
- **To understand Hardware acceleration. • To implement different deep learning models.**

Course Outcomes:

CO1: Analyze and measure performance of sequential and parallel algorithms.
CO2: Design and Implement solutions for multicore/Distributed/parallel environment.
CO3: Identify and apply the suitable algorithms to solve AI/ML problems.
CO4: Apply the technique of Deep Neural network for implementing Linear regression and classification.
CO5: Apply the technique of Convolution (CNN) for implementing Deep Learning models
CO6: Design and develop Recurrent Neural Network (RNN) for prediction.

@The CO-PO Mapping Matrix

CO/PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	1	-	1	1	-	2	1	-	-	-	-	-
CO2	1	2	1	-	-	1	-	-	-	-	-	1
CO3	-	1	1	1	1	1	-	-	-	-	-	-
CO4	3	3	3	-	3	-	-	-	-	-	-	-
CO5	3	3	3	3	3	-	-	-	-	-	-	-
CO6	3	3	3	3	3	-	-	-	-	-	-	-
CO7	3	3	3	3	3		-	-	-	-	-	-

Group A

Assignment No: 1(A)

Title of the Assignment: Design and implement Parallel Breadth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS

Objective of the Assignment: Students should be able to perform Parallel Breadth First Search based on existing algorithms using OpenMP

Prerequisite:

1. Basic of programming language
2. Concept of BFS
3. Concept of Parallelism

Contents for Theory:

1. **What is BFS?**
2. **Example of BFS**
3. **Concept of OpenMP**
4. **How Parallel BFS Work**
5. **Code Explanation with Output**

What is 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 neighboring 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

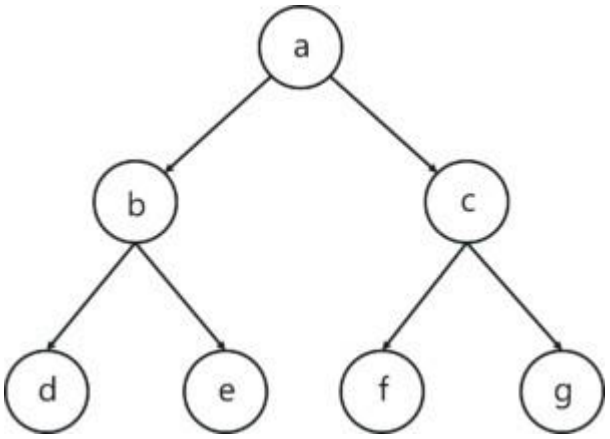
Now let's take a look at the steps involved in traversing a graph by using Breadth-First Search:

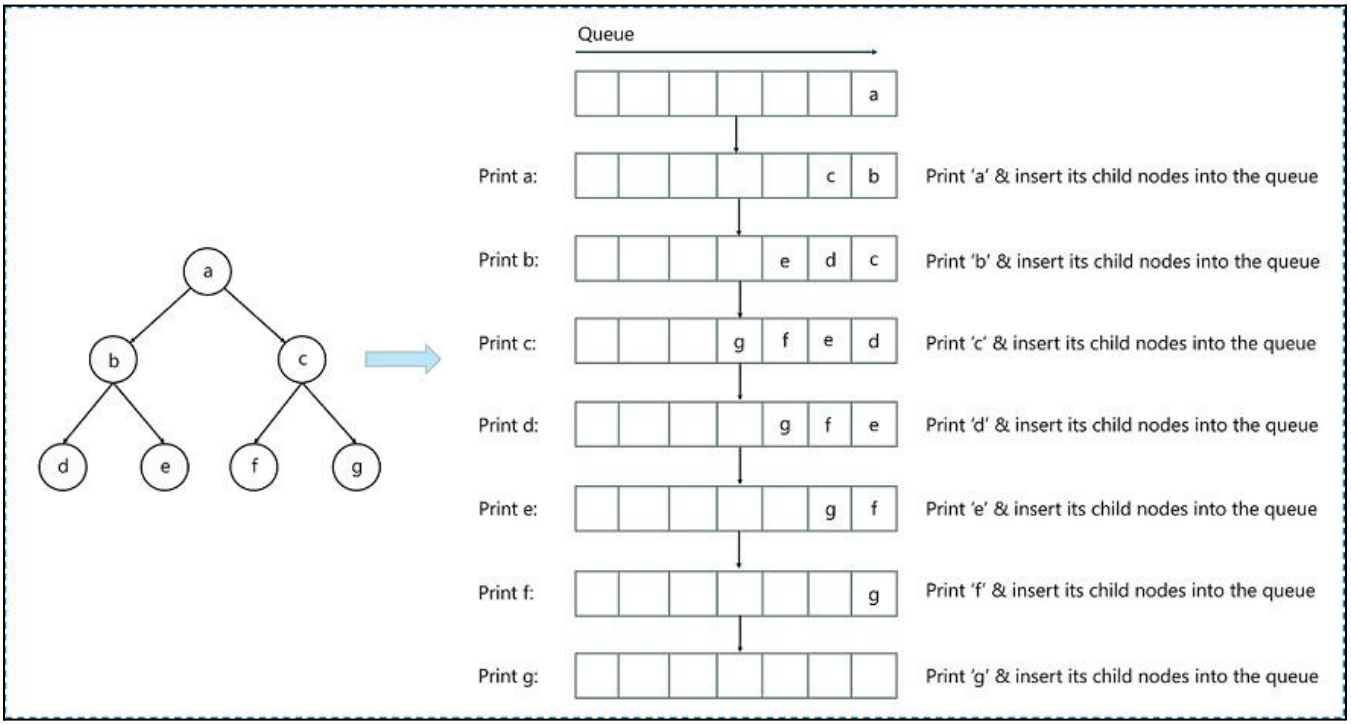
Step 1: Take an Empty Queue.

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.





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 widerange of platforms, including desktops, servers, and supercomputers.

How Parallel BFS Work

- Parallel BFS (Breadth-First Search) is an algorithm used to explore all the nodes of a graph or tree systematically in parallel. It is a popular parallel algorithm used for graph traversal in distributed computing, shared-memory systems, and parallel clusters.
- The parallel BFS algorithm starts by selecting a root node or a specified starting point, and then assigning it to a thread or processor in the system. Each thread maintains a local queue of nodes to be visited and marks each visited node to avoid processing it again.
- The algorithm then proceeds in levels, where each level represents a set of nodes that are at a certain distance from the root node. Each thread processes the nodes in its local queue at the current level, and then exchanges the nodes that are adjacent to the current level with other threads or processors. This is done to ensure that the nodes at the next level are visited by the next iteration of the algorithm.
- The parallel BFS algorithm uses two phases: the computation phase and the communication phase. In the computation phase, each thread processes the nodes in its local queue, while in the communication phase, the threads exchange the nodes that are adjacent to the current level with other threads or processors.
- The parallel BFS algorithm terminates when all nodes have been visited or when a specified node has been found. The result of the algorithm is the set of visited nodes or the shortest path from the root node to the target node.
- Parallel BFS can be implemented using different parallel programming models, such as OpenMP, MPI, CUDA, and others. The performance of the algorithm depends on the number of threads or processors used, the size of the graph, and the communication overhead between the threads or processors.

Conclusion- In this way we can achieve parallelism while implementing BFS

Assignment Question

1. **What if BFS?**
2. **What is OpenMP?What is its significance in parallel programming?**
3. **Write down applications of Parallel BFS**
4. **How can BFS be parallelized using OpenMP? Describe the parallel BFS algorithm using OpenMP.**
5. **Write Down Commands used in OpenMP?**

Reference link

- https://www.edureka.co/blog/breadth-first-search-algorithm/

----------Program----------

```
#include<iostream>
#include<stdlib.h>
#include<queue>
using namespace std;
class node
{
  public:

 node *left, *right;
 int data;

};

class Breadthfs
{

 public:

 node *insert(node *, int);
```

```
 void bfs(node *);
};

node *insert(node *root, int data)
// inserts a node in tree
{

  if(!root)
  {
          root=new node;
          root->left=NULL;
          root->right=NULL;
          root->data=data;
         return root;
  }

 queue<node *> q;
  q.push(root);

 while(!q.empty())
  {
         node *temp=q.front();
          q.pop();

          if(temp->left==NULL)
          {
                  temp->left=new node;
                  temp->left->left=NULL;
                  temp->left->right=NULL;
                  temp->left->data=data;
                  return root;
          }
          else
          {
                  q.push(temp->left);

          }

         if(temp->right==NULL)
         {
                  temp->right=new node;
                  temp->right->left=NULL;
```

```
                temp->right->right=NULL;
                temp->right->data=data;
                return root;
            }
           else
           {

            q.push(temp->right);

            }
     }

}
void bfs(node *head)
{
           queue<node*> q;
           q.push(head);
           int qSize;
          while (!q.empty())
          {
                qSize = q.size();
                #pragma omp parallel for
               //creates parallel threads
                for (int i = 0; i < qSize; i++)
                {
                      node* currNode;
                      #pragma omp critical
                      {
                        currNode = q.front();
                        q.pop();
                        cout<<"\t"<<currNode->data;
```

```
                    }// prints parent node
                    #pragma omp critical
                    {
                    if(currNode->left)// push parent's left node in queue
                    q.push(currNode->left);
                    if(currNode->right)
                    q.push(currNode->right);
                    }// push parent's right node in queue
                }
            }
    }
    int main()
      node *root=NULL;
      int data;
     char ans;
     do
     {      cout<<"\n enter data=>";
            cin>>data;
            root=insert(root,data);
            cout<<"do you want insert one more node?";
            cin>>ans;
    }while(ans=='y'||ans=='Y')
      bfs(root);
     return 0;
    }
```

Run Commands:

1. g++ -fopenmp bfs.cpp -o bfs
2. ./bfs

Output:

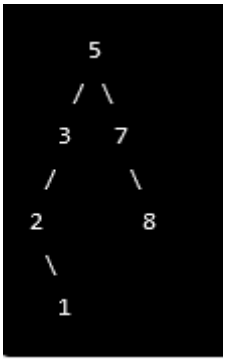
This code represents a breadth-first search (BFS) algorithm on a binary tree using OpenMP for parallelization. The program asks for user input to insert nodes into the binary tree and then performs the BFS algorithm using multiple threads. Here's an example output for a binary tree with nodes 5, 3, 2, 1, 7, and 8:

The nodes are printed in breadth-first order. The #pragma omp parallel for statement is used to parallelize the for loop that processes each level of the binary tree. The #pragma omp critical statement is used to synchronize access to shared data structures, such as the queue that stores the nodes of the binary tree.

Here is an example of the breadth-first traversal for a binary tree with the values 5, 3, 2, 1, 7, and 8:

```
Enter data => 5
Do you want to insert one more node? (y/n) y

Enter data => 3
Do you want to insert one more node? (y/n) y

Enter data => 2
Do you want to insert one more node? (y/n) y

Enter data => 1
Do you want to insert one more node? (y/n) y

Enter data => 7
Do you want to insert one more node? (y/n) y

Enter data => 8
Do you want to insert one more node? (y/n) n

        5       3       7       2       1       8
```

Starting with the root node containing value 5:



The traversal would be:

```
5, 3, 7, 2, 8, 1
```

Explanation

This C++ code demonstrates how to perform a breadth-first search (BFS) in a binary tree using OpenMP parallel programming.

1. The program starts by defining a class "node" that defines the properties of a binary tree node. This class has two pointers to the left and right child nodes of the current node, and an integer to store the data value of the node.
2. Next, a class named "Breadthfs" is defined, which contains two methods - insert() and bfs(). The insert() method is used to insert a new node in the binary tree, while the bfs() method is used to perform the BFS algorithm on the binary tree.
3. The insert() method takes two arguments - a pointer to the root node of the binary tree and an integer value to be inserted. If the root node is null, the method creates a new node, sets its data value to the given integer value and returns the root node.
4. If the root node is not null, the method creates an empty queue of node pointers and pushes the root node into the queue. It then enters a loop that runs until the queue is empty.

5. Inside the loop, the method dequeues the front node from the queue and checks if its left child is null. If it is null, the method creates a new node, sets its data value to the given integer value, and returns the root node.
6. If the left child of the front node is not null, the method pushes it onto the queue. The method then checks if the right child of the front node is null. If it is null, the method creates a new node, sets its data value to the given integer value, and returns the root node.
7. If the right child of the front node is not null, the method pushes it onto the queue.
8. The bfs() method takes a pointer to the root node of the binary tree as its argument. It creates an empty queue of node pointers and pushes the root node into the queue.
9. It then enters a loop that runs until the queue is empty. Inside the loop, it retrieves the size of the queue and creates an OpenMP parallel region using the "omp parallel for" directive. This directive creates a team of parallel threads to execute the loop body in parallel.
10. Inside the loop body, each thread dequeues a node from the queue using a critical section to ensure that no two threads access the same node simultaneously. It prints the data value of the current node to the console.
11. The thread then checks if the left and right child nodes of the current node are not null. If they are not null, the thread uses another critical section to push the left and right child nodes onto the queue.
12. The main function starts by initializing the root node pointer to null and declaring an integer variable to store the user input.
13. It uses a do-while loop to prompt the user to enter a value to be inserted in the binary tree. If the user enters 'y' or 'Y', the loop continues to accept more input. Otherwise, the loop terminates.
14. For each user input value, the program calls the insert() method to insert a new node in the binary tree.
15. After the user is finished inputting values, the program calls the bfs() method to perform a breadth-first search on the binary tree.
16. Finally, the program returns 0 to indicate successful execution.

Explanation 2

This C++ code implements the Breadth-First Search (BFS) algorithm to traverse a binary tree. Here is a step-by-step explanation of the code's execution flow:

1. The code defines a node class with left, right, and data members, which are pointers to node, pointers to the left and right child nodes, and the data to be stored in each node, respectively.
2. The code defines a Breadthfs class with insert and bfs member functions, which are responsible for inserting a new node into the binary tree and traversing the tree in a breadth-first manner, respectively.
3. The insert function takes two arguments: a pointer to the root node of the tree and an integer value to be inserted. If the root node is NULL, it creates a new node with the given value and returns the new node. Otherwise, it uses a queue to traverse the tree level by level, looking for the first empty child node (either left or right). When an empty node is found, it creates a new node with the given value and returns the root node.
4. The bfs function takes a pointer to the root node of the tree and performs a breadth-first traversal of the tree. It starts by initializing a queue with the root node and a variable to store the size of the queue. Then, it enters a loop that continues until the queue is empty.
5. Inside the loop, it obtains the current size of the queue, and for each node in the queue, it pops the front node and prints its data value. It then adds the node's left and right child nodes to the queue if they exist.
6. To improve the performance of the BFS traversal, the loop that processes each node is parallelized using OpenMP, a library that enables parallel programming in C++. The #pragma omp parallel for directive creates multiple threads that execute the loop iterations in parallel, and the #pragma omp critical directive ensures that only one thread at a time can access the shared resources (i.e., the queue and the console output).
7. Finally, the main function initializes a pointer to the root node and prompts the user to enter integer values to be inserted into the tree. It uses the insert function to create a new node for each value and adds it to the tree. It continues until the user chooses to stop inserting new values. Then, it calls the bfs function to traverse the tree in a breadth-first manner and prints the data values of each node.

Group A

Assignment No: 1(B)

Title of the Assignment: Design and implement Parallel Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for DFS

Objective of the Assignment: Students should be able to perform Parallel Depth First Search based on existing algorithms using OpenMP

Prerequisite:

1. Basic of programming language
2. Concept of DFS
3. Concept of Parallelism

Contents for Theory:

1. **What is DFS?**
2. **Example of DFS**
3. **Concept of OpenMP**
4. **How Parallel DFS Work**

What is 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.

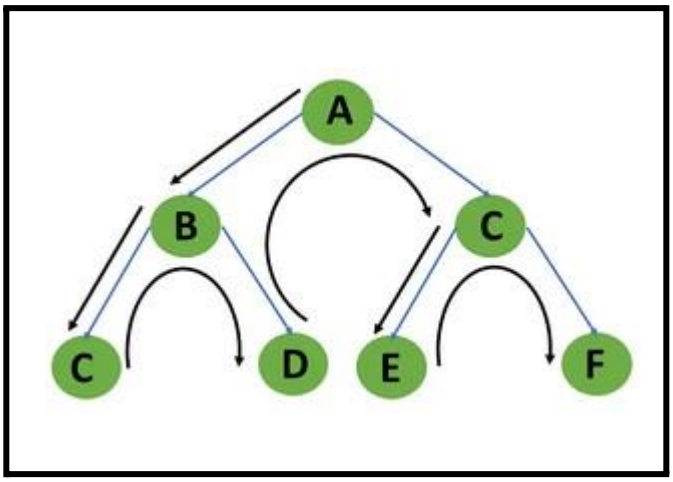
DFS can be implemented using either a recursive or an iterative approach. The recursive approach is simpler to implement but can lead to a stack overflow error for very large graphs. The iterative approach uses a stack to keep track of nodes to be explored and is preferred for larger graphs.

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 thestack.

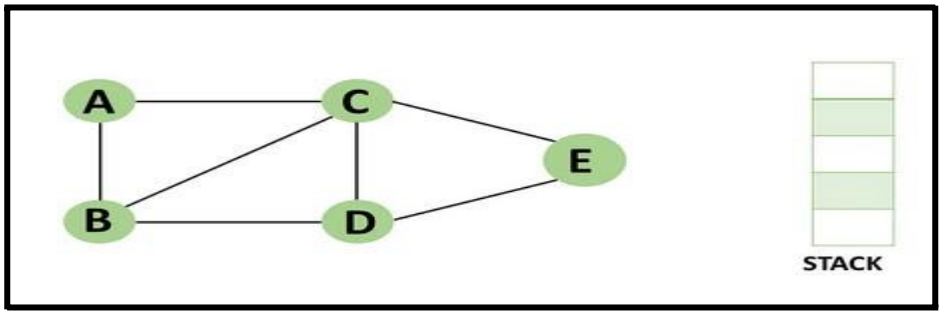
Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of thestack.

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.

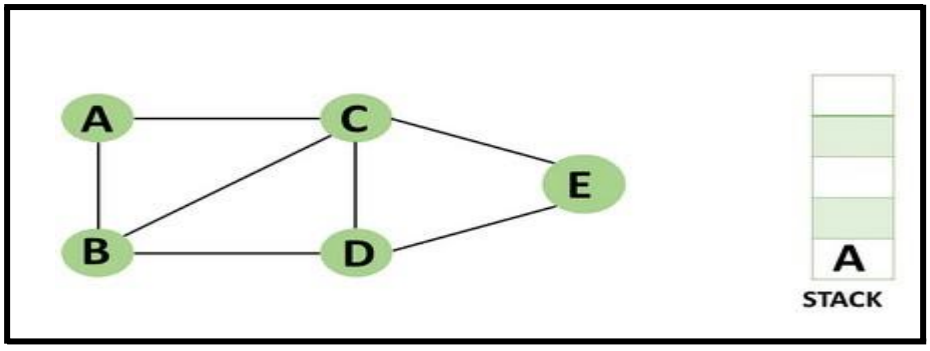
Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph'sunused edges.

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



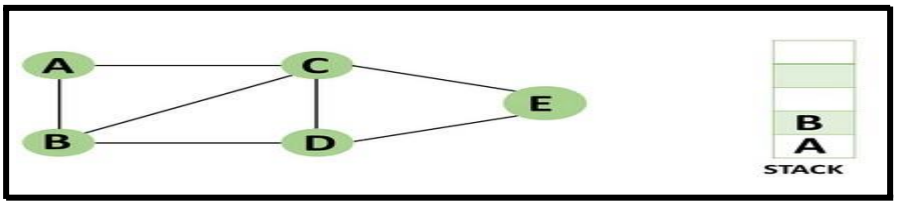
Step 1: Mark vertex A as a visited source node by selecting it as a source node.

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



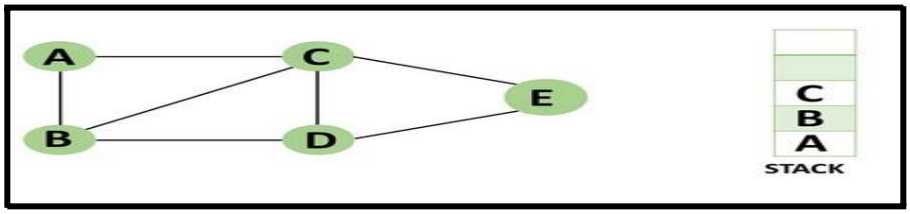
Step 2: Any nearby unvisited vertex of vertex A, say B, should be visited.

You should push vertex B to the top of the stack



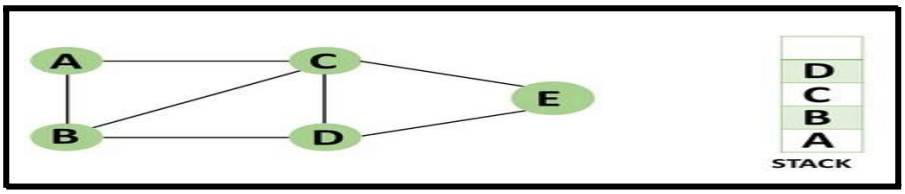
Step 3: From vertex C and D, visit any adjacent unvisited vertices of vertex B. Imagine you have chosen vertex C, and you want to make C a visited vertex.

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

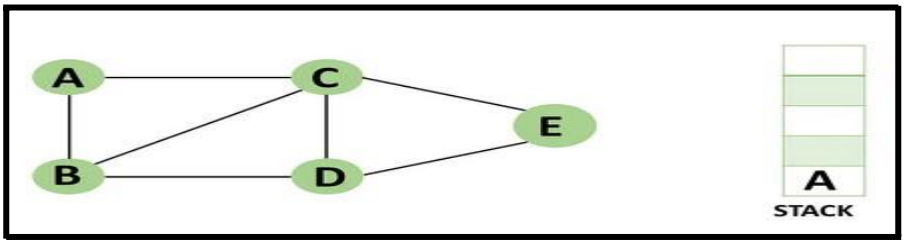


Step 4: You can visit any nearby unvisited vertices of vertex C, you need to select vertex D and designate it as a visited vertex.

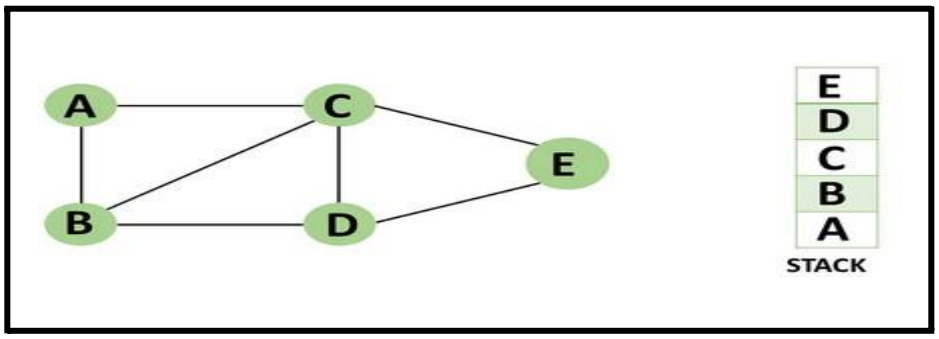
- Vertex D is pushed to the top of the stack.



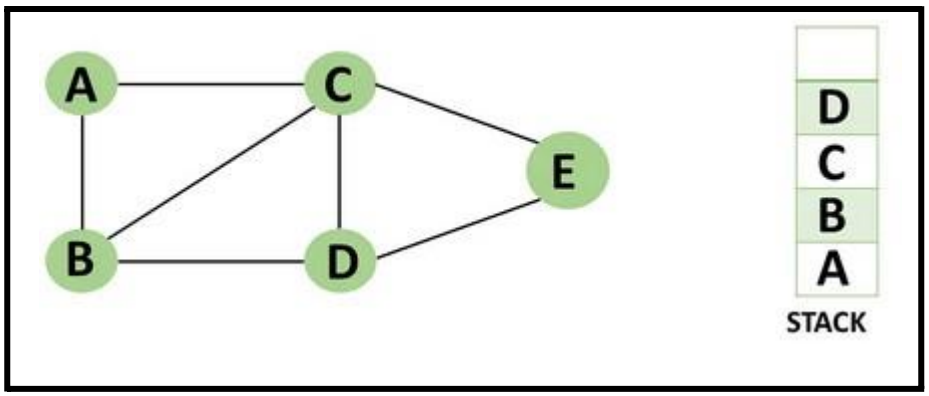
Step 5: Vertex E is the lone unvisited adjacent vertex of vertex D, thus marking it as visited.



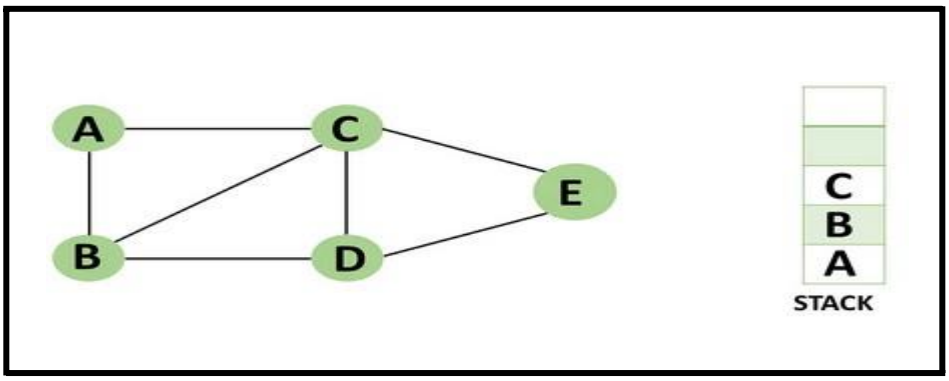
- Vertex E should be pushed to the top of the stack.



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

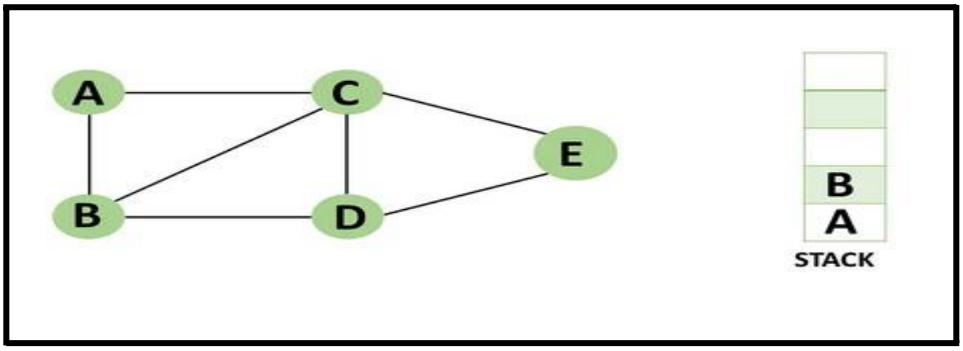


Step 7: Now that all of vertex D's nearby vertices, namely vertex B and C, have been visited, pop vertex D from the stack.

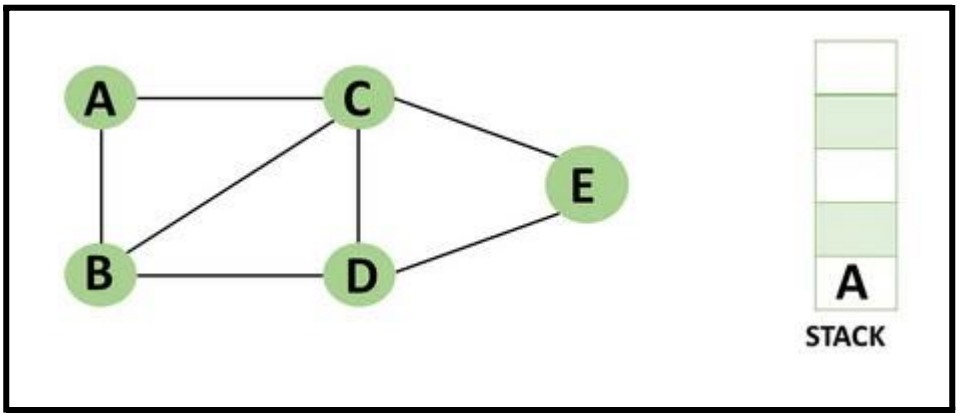


Step 8: Similarly, vertex C's adjacent vertices have already been visited; therefore, pop it from

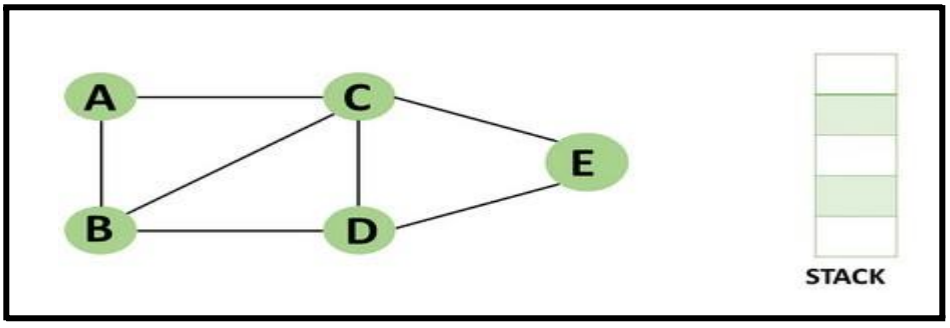
the stack.



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



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



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.

How Parallel DFS Work

- Parallel Depth-First Search (DFS) is an algorithm that explores the depth of a graph structure to search for nodes. In contrast to a serial DFS algorithm that explores nodes in a sequential manner, parallel DFS algorithms explore nodes in a parallel manner, providing a significant speedup in large graphs.
- Parallel DFS works by dividing the graph into smaller subgraphs that are explored simultaneously. Each processor or thread is assigned a subgraph to explore, and they work independently to explore the subgraph using the standard DFS algorithm. During the exploration process, the nodes are marked as visited to avoid revisiting them.
- To explore the subgraph, the processors maintain a stack data structure that stores the nodes in the order of exploration. The top node is picked and explored, and its adjacent nodes are pushed onto the stack for further exploration. The stack is updated concurrently by the processors as they explore their subgraphs.

- Parallel DFS can be implemented using several parallel programming models such as OpenMP, MPI, and CUDA. In OpenMP, the #pragma omp parallel for directive is used to distribute the work among multiple threads. By using this directive, each thread operates on a different part of the graph, which increases the performance of the DFS algorithm.

Conclusion- In this way we can achieve parallelism while implementing DFS

Assignment Question

1. **What if DFS?**
2. **Write a parallel Depth First Search (DFS) algorithm using OpenMP**
3. **What is the advantage of using parallel programming in DFS?**
4. **How can you parallelize a DFS algorithm using OpenMP?**
5. **What is a race condition in parallel programming, and how can it be avoided in OpenMP?**

Reference link

- https://www.programiz.com/dsa/graph-dfs
- https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm

PROGRAM

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

using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node) {
        stack<int> s;
        s.push(node);

        while (!s.empty()) {
        int curr_node = s.top();
        s.pop();
```

```
        if (!visited[curr_node]) {
        visited[curr_node] = true;

        if (visited[curr_node]) {
        cout << curr_node << " ";
        }

        #pragma omp parallel for
        for (int i = 0; i < graph[curr_node].size(); i++) {
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                s.push(adj_node);
                }
        }
        }
        }
}

int main() {
        int n, m, start_node;
        cout << "Enter No of Node,Edges,and start node:" ;
        cin >> n >> m >> start_node;
      //n: node,m:edges

cout << "Enter Pair of edges:" ;
        for (int i = 0; i < m; i++) {
        int u, v;

        cin >> u >> v;
//u and v: Pair of edges
        graph[u].push_back(v);
        graph[v].push_back(u);
        }

        #pragma omp parallel for
        for (int i = 0; i < n; i++) {
        visited[i] = false;
        }
        dfs(start_node);

/*      for (int i = 0; i < n; i++) {
        if (visited[i]) {
```

```
        cout << i << " ";
        }
        }*/

        return 0;
}
```

Explanation:
Let's go through the code step by step:

1. We start by including the necessary headers and declaring some global variables, such as the graph adjacency list, an array to keep track of visited nodes, and a maximum limit for the number of nodes in the graph.
2. Next, we define a function called dfs() which takes a starting node as input and performs the depth-first search algorithm. We use a stack to keep track of the nodes to be visited. The algorithm works as follows:
   - We push the starting node onto the stack.
   - While the stack is not empty, we pop the top node from the stack.
   - If the current node has not been visited, we mark it as visited and explore all its neighbors that have not been visited yet by adding them to the stack.
3. In the main() function, we read the input values: the number of nodes in the graph, the number of edges, and the starting node.
4. We then read the edge information and build the adjacency list of the graph.
5. We initialize the visited array to false for all nodes in the graph using OpenMP's parallel for construct to allow multiple threads to set the array values in parallel.
6. We call the dfs() function with the starting node, which performs the depth-first search algorithm in parallel.
7. Finally, we print out the list of visited nodes in the order they were visited.

The parallelization of the DFS algorithm is achieved by using the OpenMP parallel for construct inside the dfs() function to explore the neighbours of each node in parallel. This allows multiple threads to work on different parts of the graph simultaneously, which can lead to significant performance gains on multi-core CPUs

step-by-step explanation of how this program works:

dfs function

This function performs a parallel depth-first search starting from a given node. It takes the node number as a parameter and modifies the global visited array to keep track of which nodes have been visited.

1. It starts by creating an empty stack and pushing the starting node onto it.

```c++
void dfs(int node) {
    stack<int> s;
    s.push(node);
```

2. It enters a loop that continues until the stack is empty.

```c++
    while (!s.empty()) {
        int curr_node = s.top();
        s.pop();
```

3. For each node that is popped off the stack, it checks if that node has already been visited. If not, it sets the visited flag for that node to true and adds its unvisited neighbors to the stack in parallel.

```c++
        if (!visited[curr_node]) {
            visited[curr_node] = true;

            #pragma omp parallel for
            for (int i = 0; i < graph[curr_node].size(); i++) {
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                    s.push(adj_node);
                }
            }
        }
    }
}
```

Note that the #pragma omp parallel for directive is used to parallelize the loop that adds neighboring nodes to the stack. This can speed up the execution of the program on multi-core machines.

main function

This function reads in the input, initializes the visited array, calls the dfs function, and prints out the list of visited nodes.

1. It reads in the number of nodes, the number of edges, and the starting node from standard input.

```c++
int main() {
    int n, m, start_node;
    cin >> n >> m >> start_node;
```

2. It reads in the edge list and builds the adjacency list representation of the graph. Note that each edge is added to both the u node's and v node's adjacency list to represent an undirected graph.

```c++
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }
```

3. It initializes the visited array to false in parallel.

```c++
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        visited[i] = false;
    }
```

4. It calls the dfs function with the starting node as the argument.

```c++
    dfs(start_node);
```

5.It prints out the list of visited nodes in ascending order.

```c++
    for (int i = 0; i < n; i++) {
        if (visited[i]) {
            cout << i << " ";
        }
    }

    return 0;
}
```

Output:
here's an example input and output for a small graph with 6 nodes and 5 edges:

Input:

```
6 7 0
0 1
0 2
1 3
2 4
2 5
4 5
5 3
```

Output:

```
0 1 2 4 5 3
```

Group A

Assignment No: 2(B)

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

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

Prerequisite:

1. Basic of programming language
2. Concept of Bubble Sort
3. Concept of Parallelism

Contents for Theory:

1. **What is Bubble Sort? Use of Bubble Sort**
2. **Example of Bubble sort?**
3. **Concept of OpenMP**
4. **How Parallel Bubble Sort Work**
5. **How to measure the performance of sequential and parallel algorithms?**

What is 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.

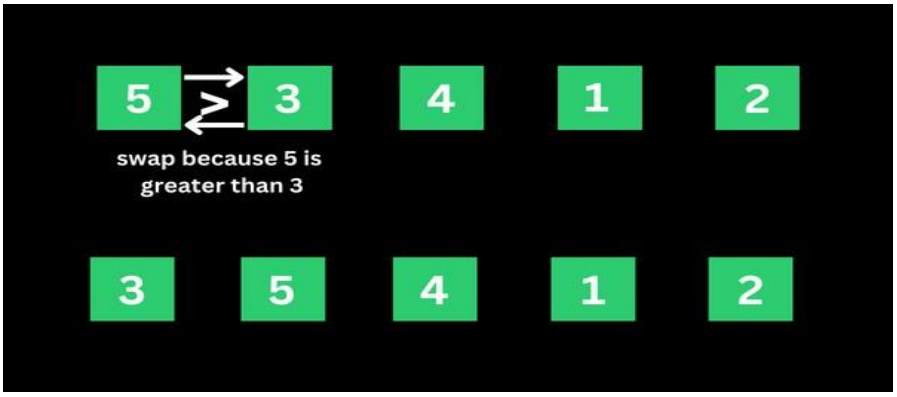
Example of Bubble sort

Let's say we want to sort a series of numbers 5, 3, 4, 1, and 2 so that they are arranged in ascending order…

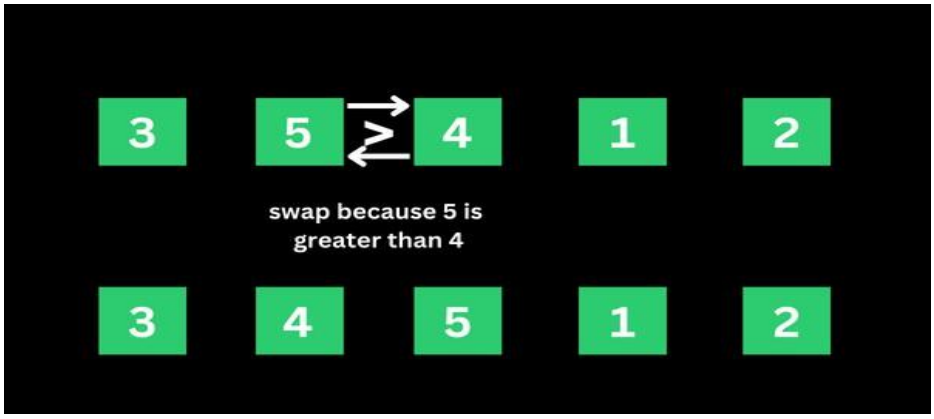
The sorting begins the first iteration by comparing the first two values. If the first value is greater than the second, the algorithm pushes the first value to the index of the second value.

First Iteration of the Sorting

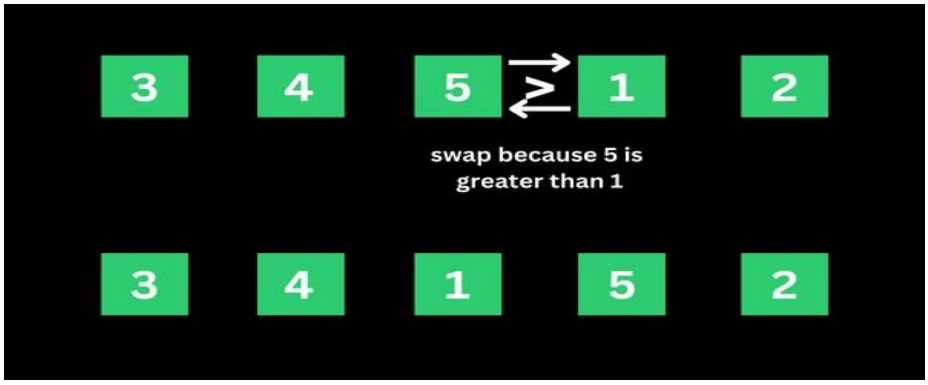
Step 1: In the case of 5, 3, 4, 1, and 2, 5 is greater than 3. So 5 takes the position of 3 and the numbers become 3, 5, 4, 1, and 2.



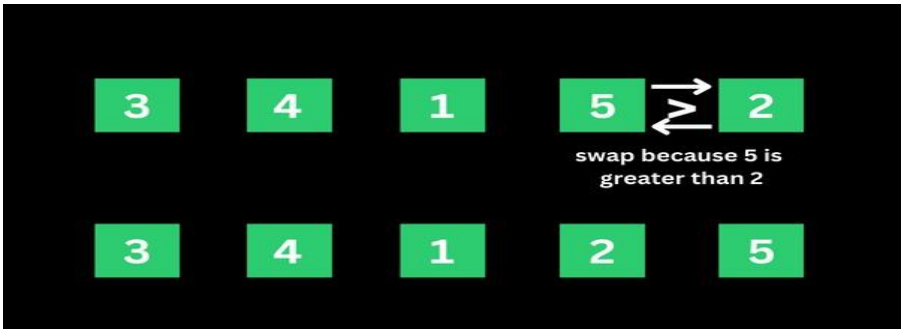
Step 2: The algorithm now has 3, 5, 4, 1, and 2 to compare, this time around, it compares the next two values, which are 5 and 4. 5 is greater than 4, so 5 takes the index of 4 and the values now become 3, 4, 5, 1, and 2.



Step 3: The algorithm now has 3, 4, 5, 1, and 2 to compare. It compares the next two values, which are 5 and 1. 5 is greater than 1, so 5 takes the index of 1 and the numbers become 3, 4, 1, 5, and 2.



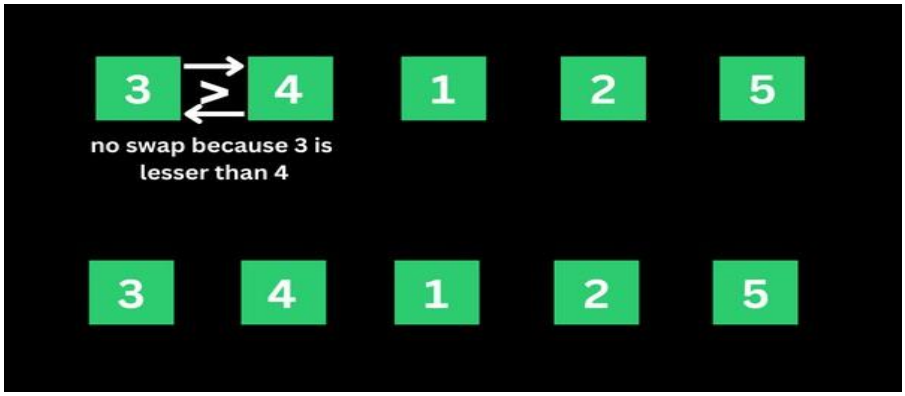
Step 4: The algorithm now has 3, 4, 1, 5, and 2 to compare. It compares the next two values, which are 5 and 2. 5 is greater than 2, so 5 takes the index of 2 and the numbers become 3, 4, 1, 2, and 5.



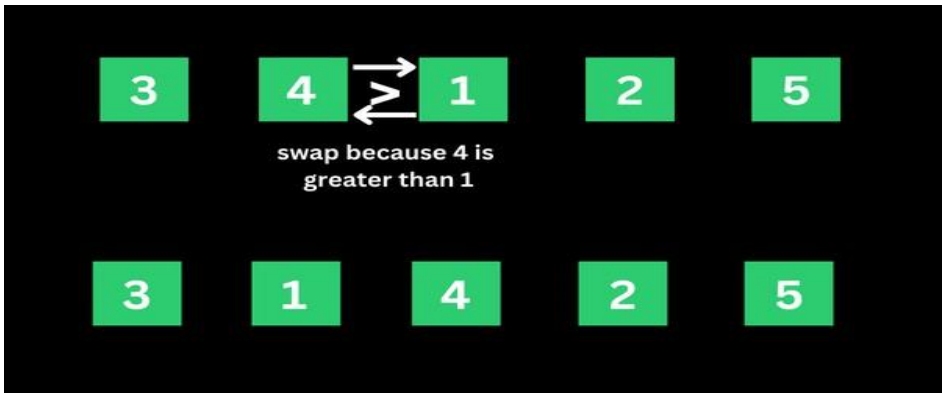
That's the first iteration. And the numbers are now arranged as 3, 4, 1, 2, and 5 – from the initial 5, 3, 4, 1, and 2. As you might realize, 5 should be the last number if the numbers are sorted in ascending order. This means the first iteration is really completed.

Second Iteration of the Sorting and the Rest

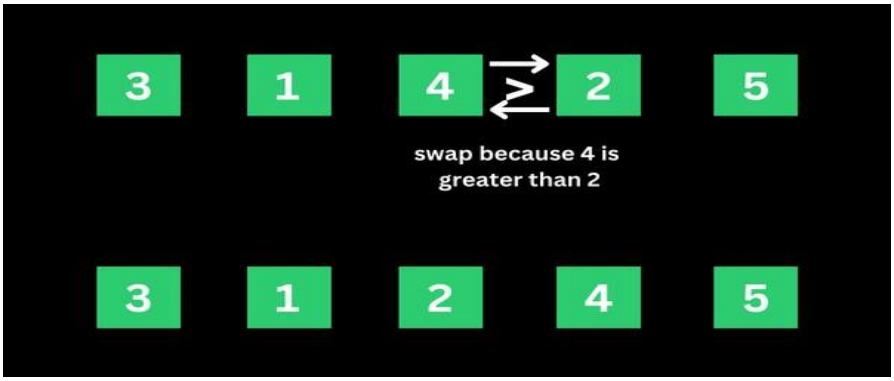
The algorithm starts the second iteration with the last result of 3, 4, 1, 2, and 5. This time around, 3 is smaller than 4, so no swapping happens. This means the numbers will remain the same.



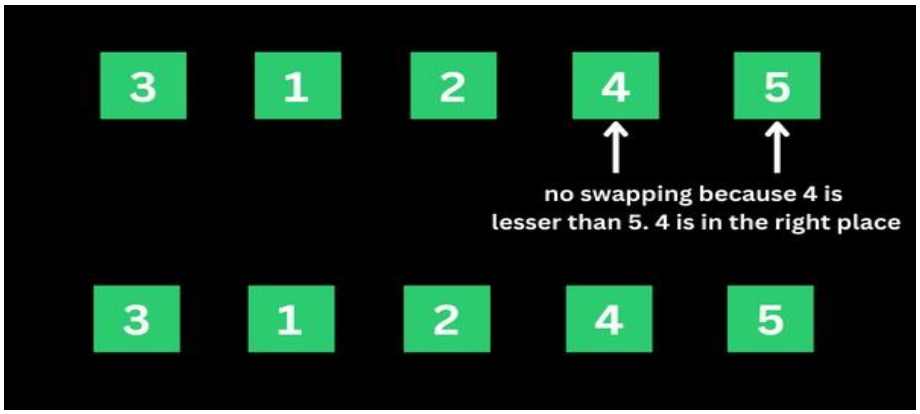
The algorithm proceeds to compare 4 and 1. 4 is greater than 1, so 4 is swapped for 1 and the numbers become 3, 1, 4, 2, and 5.



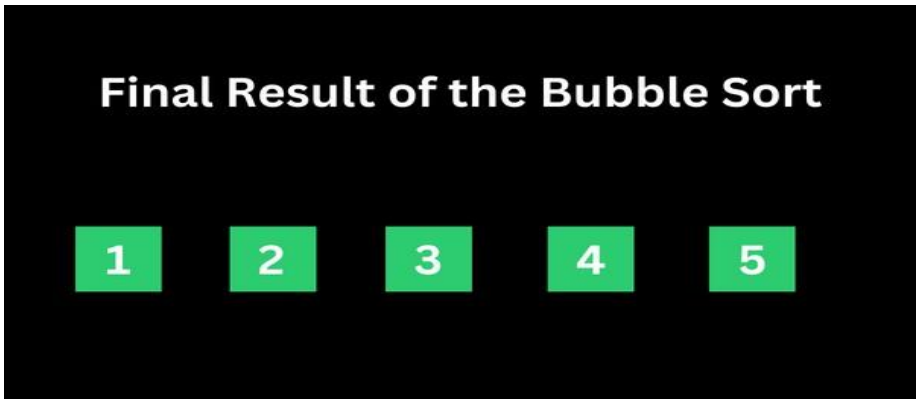
The algorithm now proceeds to compare 4 and 2. 4 is greater than 2, so 4 is swapped for 2 and the numbers become 3, 1, 2, 4, and 5.



4 is now in the right place, so no swapping occurs between 4 and 5 because 4 is smaller than 5.



That's how the algorithm continues to compare the numbers until they are arranged in ascending order of 1, 2, 3, 4, and 5.



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.

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 sublists that are sorted concurrently by multiple threads. Each thread sorts its sublist using the regular Bubble Sort algorithm. When all sublists have been sorted, they are merged together 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.

How to measure the performance of sequential and parallel algorithms?

To measure the performance of sequential Bubble sort and parallel Bubble sort algorithms, you can follow these steps:

1. Implement both the sequential and parallel Bubble sort algorithms.
2. Choose a range of test cases, such as arrays of different sizes and different degrees of sortedness, to test the performance of both algorithms.
3. Use a reliable timer to measure the execution time of each algorithm on each test case.
4. Record the execution times and analyze the results.

When measuring the performance of the parallel Bubble sort algorithm, you will need to specify the number of threads to use. You can experiment with different numbers of threads to find the optimal value for your system.

Here are some additional tips for measuring performance:

- Run each algorithm multiple times on each test case and take the average execution time to reduce the impact of variations in system load and other factors.
- Monitor system resource usage during execution, such as CPU utilization and memory consumption, to detect any performance bottlenecks.
- Visualize the results using charts or graphs to make it easier to compare the performance of the two algorithms.

How to check CPU utilization and memory consumption in ubuntu

In Ubuntu, you can use a variety of tools to check CPU utilization and memory consumption. Here are some common tools:

1. **top:** The top command provides a real-time view of system resource usage, including CPU utilization and memory consumption. To use it, open a terminal window and type top. The output will display a list of processes sorted by resource usage, with the most resource-intensive processes at the top.
2. **htop**: htop is a more advanced version of top that provides additional features, such as interactive process filtering and a color-coded display. To use it, open a terminal window and type htop.
3. **ps**: The ps command provides a snapshot of system resource usage at a particular moment in time. To use it, open a terminal window and type ps aux. This will display a list of all running processes and their resource usage.
4. **free:** The free command provides information about system memory usage, including total, used, and free memory. To use it, open a terminal window and type free -h.
5. **vmstat:** The vmstat command provides a variety of system statistics, including CPU utilization, memory usage, and disk activity. To use it, open a terminal window and type vmstat.

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

Assignment Question

1. **What is parallel Bubble Sort?**
2. **How does Parallel Bubble Sort work?**
3. **How do you implement Parallel Bubble Sort using OpenMP?**
4. **What are the advantages of Parallel Bubble Sort?**
5. **Difference between serial bubble sort and parallel bubble sort**

Reference link

- https://www.freecodecamp.org/news/bubble-sort-algorithm-in-java-cpp-python-with-example-code/

----------PROGRAM----------

```
#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;
}
```

Here, we're using the #pragma omp parallel directive to parallelize the code using OpenMP. Within the parallel block, we have a while loop that runs until the array is sorted. Inside the while loop, we have a #pragma omp for directive that parallelizes the for loop that performs the bubble sort.

Note that the flag variable is used to keep track of whether any swaps were made during an iteration of the for loop. If no swaps were made, then the array is already sorted and we can exit the loop.

explanation of each line of the code:

c++ Copy code

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

using namespace std;
```

These lines are the standard C++ header files used for input/output operations and OpenMP, respectively.

```c++
void parallel_bubble_sort(int arr[], int n) {
    #pragma omp parallel
    {
        int i, temp, flag = 1;
        while (flag) {
            flag = 0;
            #pragma omp for
            for (i = 0; i < n-1; i++) {
                if (arr[i] > arr[i+1]) {
                    temp = arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = temp;
                    flag = 1;
                }
            }
        }
    }
}
```

This is the function parallel_bubble_sort() that implements the parallel bubble sort algorithm using OpenMP.

- #pragma omp parallel is a directive that creates a team of threads to execute the parallel code inside the block. In this case, the block contains the code for bubble sort algorithm.
- int i, temp, flag = 1; declares the variables i, temp, and flag that will be used inside the while loop.
- while (flag) is a loop that runs until the flag variable is 0.
- flag = 0; sets the flag variable to 0 before starting each iteration of the for loop.
- #pragma omp for is a directive that parallelizes the for loop, by dividing the loop iterations among the threads in the team. Each thread performs the sorting operation on a subset of the array, thereby making the sorting process faster.
- for (i = 0; i < n-1; i++) is a for loop that iterates over the array, from 0 to n-1.
- if (arr[i] > arr[i+1]) checks if the current element is greater than the next element.
- temp = arr[i]; arr[i] = arr[i+1]; arr[i+1] = temp; swaps the current element with the next element, using a temporary variable.
- flag = 1; sets the flag variable to 1, indicating that a swap has been made.

- Finally, the sorted array is printed using a for loop.

```c++
int main() {
    int arr[] = {5, 3, 1, 9, 8, 2, 4, 7, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    parallel_bubble_sort(arr, n);

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

This is the main() function, which initializes an array arr and its size n. The function parallel_bubble_sort() is called with these arguments to sort the array. The sorted array is then printed to the console using a for loop.

How to Run code in Ubuntu:

1. **Open a terminal window.**

Compile the code using the following command:

```
g++ -fopenmp filename.cpp -o executable
```

Run the program by executing the following command:

```bash
./executable
```

Output

```
c                                                    Copy code

Sorted array: 1 2 3 4 5 6 7 8 9
```

This is because the input array `{5, 3, 1, 9, 8, 2, 4, 7, 6}` is sorted in ascending order using the parallel bubble sort algorithm implemented in the `parallel_bubble_sort()` function. The sorted array is then printed to the console in the `main()` function using a for loop.

To measure the performance of sequential Bubble sort and parallel Bubble sort algorithms, you can follow these steps:

1. Implement both the sequential and parallel Bubble sort algorithms.
2. Choose a range of test cases, such as arrays of different sizes and different degrees of sortedness, to test the performance of both algorithms.
3. Use a reliable timer to measure the execution time of each algorithm on each test case.
4. Record the execution times and analyze the results.

When measuring the performance of the parallel Bubble sort algorithm, you will need to specify the number of threads to use. You can experiment with different numbers of threads to find the optimal value for your system.

Here are some additional tips for measuring performance:

- Run each algorithm multiple times on each test case and take the average execution time to reduce the impact of variations in system load and other factors.
- Monitor system resource usage during execution, such as CPU utilization and memory consumption, to detect any performance bottlenecks.
- Visualize the results using charts or graphs to make it easier to compare the performance of the two algorithms.

How to check CPU utilisation and memory consumption in ubuntu

In Ubuntu, you can use a variety of tools to check CPU utilization and memory consumption. Here are some common tools:

1. top: The top command provides a real-time view of system resource usage, including CPU utilization and memory consumption. To use it, open a terminal window and type top. The output will display a list of processes sorted by resource usage, with the most resource-intensive processes at the top.
2. htop: htop is a more advanced version of top that provides additional features, such as interactive process filtering and a color-coded display. To use it, open a terminal window and type htop.
3. ps: The ps command provides a snapshot of system resource usage at a particular moment in time. To use it, open a terminal window and type ps aux. This will display a list of all running processes and their resource usage.
4. free: The free command provides information about system memory usage, including total, used, and free memory. To use it, open a terminal window and type free -h.
5. vmstat: The vmstat command provides a variety of system statistics, including CPU utilization, memory usage, and disk activity. To use it, open a terminal window and type vmstat.

Group A

Assignment No: 2(B)

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

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

Prerequisite:

1. Basic of programming language
2. Concept of Merge Sort
3. Concept of Parallelism

Contents for Theory:

1. **What is Merge? Use of Merge Sort**
2. **Example of Merge sort?**
3. **Concept of OpenMP**
4. **How Parallel Merge Sort Work**
5. **How to measure the performance of sequential and parallel algorithms?**

What is 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.

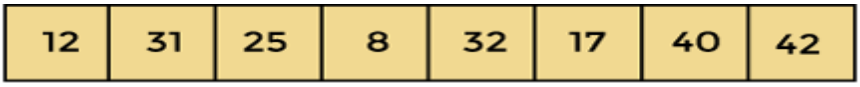
- The merging step is where the bulk of the work happens in merge sort. The algorithm compares the first elements of each sorted half, selects the smaller element, and appends it to the output array. This process continues until all elements from both halves have been appended to the output array.
-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

Example of Merge sort

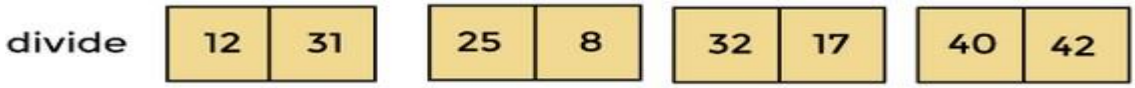
Now, let's see the working of merge sort Algorithm. To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -



- According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.
- As there are eight elements in the given array, so it is divided into two arrays of size 4.

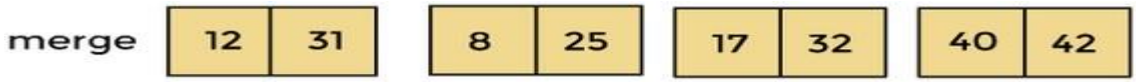
- Now, again divide these two arrays into halves. As they are of size 4, divide them into new arrays of size 2.



- Now, again divide these arrays to get the atomic value that cannot be further divided.

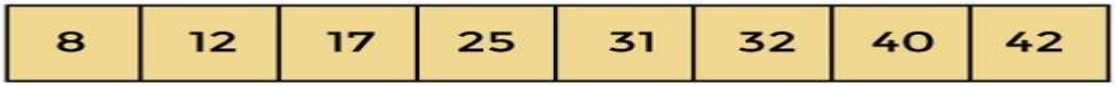
- Now, combine them in the same manner they were broken.
- In combining, first compare the element of each array and then combine them into another array in sorted order.
- So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list

of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



- In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.

- Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



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.

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.

To measure the performance of sequential and parallel merge sort algorithms, you can perform experiments on different input sizes and numbers of processors or cores. By measuring the execution time, speedup, efficiency, and scalability of the algorithms under different conditions, you can determine which algorithm is more efficient for different input sizes and hardware configurations. Additionally, you can use profiling tools to analyze the performance of the algorithms and identify areas for optimization

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

Assignment Question

1. **What is parallel Merge Sort?**
2. **How does Parallel Merge Sort work?**
3. **How do you implement Parallel MergeSort using OpenMP?**
4. **What are the advantages of Parallel MergeSort?**
5. **Difference between serial Mergesort and parallel Mergesort**

Reference link

- https://www.geeksforgeeks.org/merge-sort/
- https://www.javatpoint.com/merge-sort

----------PROGRAM----------

```
#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;
}
```

Second Code:

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

void merge(int* arr, int l, int m, int r) {
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[n1], R[n2];

        for (i = 0; i < n1; i++)
```

```
        L[i] = arr[l + i];
        for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

        i = 0;
        j = 0;
        k = l;
        while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
        }
        else {
        arr[k] = R[j];
        j++;
        }
        k++;
        }

        while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
        }

        while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
        }
}

void mergeSort(int* arr, int l, int r) {
        if (l < r) {
        int m = l + (r - l) / 2;
        #pragma omp parallel sections
        {
        #pragma omp section
        {
                mergeSort(arr, l, m);
        }
        #pragma omp section
```

```
        {
                mergeSort(arr, m + 1, r);
        }
        }

        merge(arr, l, m, r);
        }
}

int main() {
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int n = sizeof(arr) / sizeof(arr[0]);
  double start, stop;
        std::cout << "Given array is: ";
        for (int i = 0; i < n; i++)
        std::cout << arr[i] << " ";
        std::cout << std::endl;
 start = omp_get_wtime();
  #pragma omp parallel
  {
        mergeSort(arr, 0, n - 1);
}
  stop = omp_get_wtime();
        std::cout << "Sorted array is: ";
        for (int i = 0; i < n; i++)
        std::cout << arr[i] << " ";
        std::cout << std::endl;

std::cout<<stop-start;
        return 0;
}
```

Explanation

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

This block of code is a set of header files and a namespace declaration. iostream and stdlib.h are header files, and omp.h is the header file for OpenMP (Open Multi-Processing) library that is used for parallel programming.

c++ Copy code

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

These are function declarations for mergesort and merge functions.

```
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);
    }

}
```

This is the mergesort function, which implements the merge sort algorithm recursively. a[] is the input array that is to be sorted, i and j are the starting and ending indices of the subarray being sorted.

First, it checks whether i is less than j because if i is greater than or equal to j, it means that there is only one element in the subarray, which is already sorted. Then it calculates the midpoint of the subarray using (i+j)/2.

The #pragma omp parallel sections directive creates a parallel region that specifies two parallel sections that can be executed concurrently. The mergesort function is then recursively called on the left and right halves of the subarray in two separate parallel sections.

Finally, the merge function is called to merge the sorted halves of the subarray.

```c
void merge(int a[],int i1,int j1,int i2,int j2)
```

This is the function signature which takes an integer array a and four integer variables i1, j1, i2, and j2. i1 and j1 define the start and end indices of the first sorted subarray, and i2 and j2 define the start and end indices of the second sorted subarray.

```c
int temp[1000];
int i,j,k;
i=i1;
j=i2;
k=0;
```

Here, a temporary array temp is created with a size of 1000. Three integer variables i, j, and k are initialized. i and j are set to the start indices of the two subarrays, while k is set to 0.

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

This is a while loop that runs as long as i is less than or equal to j1 and j is less than or equal to j2. Inside the loop, if the element at index i of the first subarray is less than the element at index j of the second subarray, then the element at index i is copied to the temp array at index k, and i and k are incremented. Otherwise, the element at index j is copied to the temp array at index k, and j and k are incremented.

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

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

After the above loop terminates, there may be some elements left in one of the subarrays. These loops copy the remaining elements into the temp array.

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

Finally, the sorted temp array is copied back to the original a array. The loop runs from i1 to j2 and copies the elements of temp array to the corresponding indices in the a array. The loop variable j starts from 0 and increments alongside i.

Group A

Assignment No: 3

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

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

Prerequisite:

1. Parallel computing architectures
2. Parallel programming models
3. Proficiency in programming languages

Contents for Theory:

1. What is parallel reduction and its usefulness for mathematical operations on large data?
2. Concept of OpenMP
3. How do parallel reduction algorithms for Min, Max, Sum, and Average work, and what are their advantages and limitations?

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.

Assignment Question

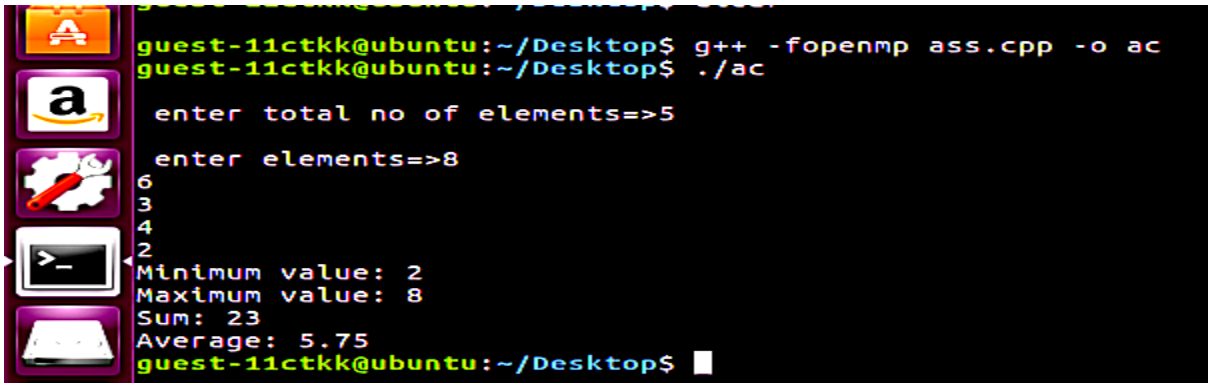
1. What are the benefits of using parallel reduction for basic operations on large arrays?
2. How does OpenMP's "reduction" clause work in parallel reduction?
3. How do you set up a C++ program for parallel computation with OpenMP?
4. What are the performance characteristics of parallel reduction, and how do they vary based on input size?
5. How can you modify the provided code example for more complex operations using parallel reduction?

----------Program----------

```
#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



Void Min_reduction()

void min_reduction(vector<int>& arr) declares a void function that takes a reference to an integer vector as its argument.

- int min_value = INT_MAX; initializes an integer variable min_value to the largest possible integer value using the INT_MAX constant from the <climits> header file. This is done to ensure that min_value is initially greater than any element in arr.
- #pragma omp parallel for reduction(min: min_value) is an OpenMP directive that specifies that the following loop should be executed in parallel using multiple threads. The reduction(min: min_value) clause indicates that each thread should maintain a private copy of min_value and update it with the minimum value it finds in its portion of the loop. Once the loop is complete, OpenMP will combine all the private copies of min_value into a single shared value that represents the minimum value in arr.
- for (int i = 0; i < arr.size(); i++) { is a loop that iterates over each element of arr.
- if (arr[i] < min_value) { min_value = arr[i]; } checks if the current element of arr is less than min_value. If so, it updates min_value to be the current element.
- cout << "Minimum value: " << min_value << endl; prints out the minimum value found in arr.

void max_reduction()

- void max_reduction(vector<int>& arr) declares a void function that takes a reference to an integer vector as its argument.

- int max_value = INT_MIN; initializes an integer variable max_value to the smallest possible integer value using the INT_MIN constant from the <climits> header file. This is done to ensure that max_value is initially smaller than any element in arr.
- #pragma omp parallel for reduction(max: max_value) is an OpenMP directive that specifies that the following loop should be executed in parallel using multiple threads. The reduction(max: max_value) clause indicates that each thread should maintain a private copy of max_value and update it with the maximum value it finds in its portion of the loop. Once the loop is complete, OpenMP will combine all the private copies of max_value into a single shared value that represents the maximum value in arr.
- for (int i = 0; i < arr.size(); i++) { is a loop that iterates over each element of arr.
- if (arr[i] > max_value) { max_value = arr[i]; } checks if the current element of arr is greater than max_value. If so, it updates max_value to be the current element.
- cout << "Maximum value: " << max_value << endl; prints out the maximum value found in arr.

#include <climits>

<climits> is a header file in C++ that contains constants related to integer types. This header file provides implementation-defined constants for minimum and maximum values of integral types, such as INT_MAX (maximum value of int) and INT_MIN (minimum value of int).

Using these constants instead of hardcoding the values of the minimum and maximum integer values is a good practice because it makes the code more readable and avoids the possibility of introducing errors in the code. The use of these constants also ensures that the code will work correctly across different platforms and compilers.

INT_MIN :

Minimum value for an object of type int
Value of INT_MIN is -32767 ($-2^{15}+1$) or less*

INT_MAX :

Maximum value for an object of type int
Value of INT_MAX is 2147483647 (-2^{31} to $2^{31}-1$)

Group A

Assignment 4(A)

Title of the Assignment: Write a CUDA Program for Addition of two large vectors

Objective of the Assignment: Students should be able to perform CUDA Program for Addition of two large vectors

Prerequisite:

1. CUDA Concept
2. Vector Addition
3. How to execute Program in CUDA Environment

Contents for Theory:

1. **What is CUDA**
2. **Addition of two large Vector**
3. **Execution of CUDA Environment**

What is 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.

Questions:

1. What is the purpose of using CUDA to perform addition of two large vectors?
2. How do you allocate memory for the vectors on the device using CUDA?
3. How do you launch the CUDA kernel to perform the addition of two large vectors?
4. How can you optimize the performance of the CUDA program for adding two large vectors

----------Program----------

```
#include <iostream>
#include <cuda_runtime.h>
#include /usr/local/cuda/include/cuda_runtime.h

_global_void addVectors(int* A, int* B, int* C, int n)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
  {
        C[i] = A[i] + B[i];
  }
}

int main()
{
  int n = 1000000;
  int* A, * B, * C;
  int size = n * sizeof(int);

  // Allocate memory on the host
  cudaMallocHost(&A, size);
  cudaMallocHost(&B, size);
  cudaMallocHost(&C, size);

  // Initialize the vectors
  for (int i = 0; i < n; i++)
  {
        A[i] = i;
        B[i] = i * 2;
  }
  // Allocate memory on the device
  int* dev_A, * dev_B, * dev_C;
```

```
    cudaMalloc(&dev_A, size);
    cudaMalloc(&dev_B, size);
    cudaMalloc(&dev_C, size);

    // Copy data from host to device
    cudaMemcpy(dev_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, B, size, cudaMemcpyHostToDevice);

    // Launch the kernel
    int blockSize = 256;
    int numBlocks = (n + blockSize - 1) / blockSize;
    // Copy data from device to host
    cudaMemcpy(C, dev_C, size, cudaMemcpyDeviceToHost);

    // Print the results
    for (int i = 0; i < 10; i++)
    {
            cout << C[i] << " ";
    }
    cout << endl;

    // Free memory
    cudaFree(dev_A);
    cudaFree(dev_B);
    cudaFree(dev_C);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);

    return 0;
}
```

Group A

Assignment 4(B)

Title of the Assignment: Write a Program for Matrix Multiplication using CUDA C

Objective of the Assignment: Students should be able to performProgram for Matrix Multiplication using CUDA C

Prerequisite:

1. CUDA Concept
2. Matrix Multiplication
3. How to execute Program in CUDA Environment

Contents for Theory:

1. **What is CUDA**
2. **Matrix Multiplication**
3. **Execution of CUDA Environment**

What is 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 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
```

Questions:

1. What are the advantages of using CUDA to perform matrix multiplication compared to using a CPU?
2. How do you handle matrices that are too large to fit in GPU memory in CUDA matrix multiplication?
3. How do you optimize the performance of the CUDA program for matrix multiplication?
4. How do you ensure correctness of the CUDA program for matrix multiplication and verify the results?

----------Program----------

```
#include <cuda_runtime.h>
#include <iostream>
__global__ void matmul(int* A, int* B, int* C, int N) {
int Row = blockIdx.y*blockDim.y+threadIdx.y;
int Col = blockIdx.x*blockDim.x+threadIdx.x;
if (Row < N && Col < N) {
int Pvalue = 0;
for (int k = 0; k < N; k++) {
Pvalue += A[Row*N+k] * B[k*N+Col];
}
C[Row*N+Col] = Pvalue;
}
}
int main() {
int N = 512;
int size = N * N * sizeof(int);
int* A, * B, * C;
int* dev_A, * dev_B, * dev_C;
cudaMallocHost(&A, size);
cudaMallocHost(&B, size);
cudaMallocHost(&C, size);
cudaMalloc(&dev_A, size);
cudaMalloc(&dev_B, size);
```

```
cudaMalloc(&dev_C, size);
// Initialize matrices A and B
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
A[i*N+j] = i*N+j;
B[i*N+j] = j*N+i;
}
}
cudaMemcpy(dev_A, A, size,
cudaMemcpyHostToDevice);
cudaMemcpy(dev_B, B, size,
cudaMemcpyHostToDevice);
dim3 dimBlock(16, 16);
dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);
matmul<<<dimGrid, dimBlock>>>(dev_A, dev_B,
dev_C, N);
cudaMemcpy(C, dev_C
// Print the result
for (int i = 0; i < 10; i++) {
for (int j = 0; j < 10; j++) {
std::cout << C[i*N+j] << " ";
}
std::cout << std::endl;
}
// Free memory
cudaFree(dev_A);
cudaFree(dev_B);
cudaFree(dev_C);
cudaFreeHost(A);
cudaFreeHost(B);
cudaFreeHost(C);
return 0;
```

Group B
Mini Project: 1

Mini Project: Evaluate performance enhancement of parallel Quicksort Algorithm using MPI

Application: Parallel Quicksort Algorithm using MPI

Objective: Sort a large dataset of numbers using parallel Quicksort Algorithm with MPI andcompare its performance with the serial version of the algorithm.

Approach: We will use Python and MPI to implement the parallel version of Quicksort Algorithmand compare its performance with the serial version of the algorithm.

Requirements:

Python 3.xmpi4py

Theory :

Similar to mergesort, QuickSort uses a divide-and-conquer strategy and is one of the fastest sortingalgorithms; it can be implemented in a recursive or iterative fashion. The divide and conquer is a general algorithm design paradigm and key steps of this strategy can be summarized as follows:

- **Divide**: Divide the input data set S into disjoint subsets S1, S2, S3…Sk.
- **Recursion**: Solve the sub-problems associated with S1, S2, S3…Sk.
- **Conquer**: Combine the solutions for S1, S2, S3…Sk. into a solution for S.
- **Base case**: The base case for the recursion is generally subproblems of size 0 or 1.

Many studies [2] have revealed that in order to sort N items; it will take QuickSort an average running time of O(NlogN). The worst-case running time for QuickSort will occur when the pivot is a unique minimum or maximum element, and as stated in [2], the worst-case running time for QuickSort on N items is $O(N^2)$. These different running times can be influenced by the input distribution (uniform, sorted or semi-sorted, unsorted, duplicates) and the choice of the pivot element. Here is a simple pseudocode of the QuickSort algorithm adapted from Wikipedia [1].

We have made use of Open MPI as the backbone library for parallelizing the QuickSort algorithm. In fact, learning message passing interface (MPI) allows us to strengthen our fundamental knowledge on parallel programming, given that MPI is lower level than equivalent libraries (OpenMP). As simple as its name means, the basic idea behind MPI is that messages can be passed or exchanged among different processes in order to perform a given task. An illustration can be a communication and coordination by a master process which splits a huge task into chunks and shares them to its slave processes. Open MPI is

developed and maintained by a consortium of academic, research and industry partners; it combines the expertise, technologies and resources all across the high performance computing community [11]. As elaborated in [4], MPI has two types of communication routines: point-to-point communication routines and collective communication routines. Collective routines as explained in the implementation section have been used in this study.

Algorithm :

In general, the overall algorithm used here to perform QuickSort with MPI works as followed:

i. Start and initialize MPI.
ii. Under the root process MASTER, get inputs:
    a. Read the list of numbers L from an input file.
    b. Initialize the main array globaldata with L.
    c. Start the timer.

iii. Divide the input size SIZE by the number of participating processes npes to get eachchunk size local size.

iv. Distribute globaldata proportionally to all processes:
    a. From MASTER scatter globaldata to all processes.
    b. Each process receives in a sub data local data.
v. Each process locally sorts its local data of size localsize.
vi. Master gathers all sorted local data by other processes in globaldata.
    1. Gather each sorted local data.
    2. Free local data

Steps:

1. Initialize MPI:

```
from mpi4py import  MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

2. Define the serial version of Quicksort Algorithm:

```
def quicksort_serial(arr):if len(arr) <= 1:
      return arr
   pivot = arr[len(arr) // 2]
   left = [x for x in arr if x < pivot]
   middle = [x for x in arr if x == pivot]
   right = [x for x in arr if x > pivot]
   return quicksort_serial(left) + middle + quicksort_serial(right)
```

3. Define the parallel version of Quicksort Algorithm:

```
def quicksort_parallel(arr):if len(arr) <= 1:
      return arr
   pivot = arr[len(arr) // 2]
   left = []
   middle = []
   right = []

   for x in arr:
      if x < pivot:

         left.append(x)
         elif x == pivot:
         middle.append(x)

      else:
         right.append(x)
```

```
    left_size = len(left)

    middle_size = len(middle)
right_size = len(right)

# Get the size of each chunk
chunk_size = len(arr) // size

# Send the chunk to all the nodes

chunk_left = []
chunk_middle = []
chunk_right = []

comm.barrier()
comm.Scatter(left, chunk_left,
root=0) comm.Scatter(middle,
chunk_middle, root=0)
comm.Scatter(right, chunk_right,
root=0)

# Sort the chunks
chunk_left = quicksort_serial(chunk_left)
chunk_middle = quicksort_serial(chunk_middle)chunk_right =
quicksort_serial(chunk_right)

#   Gather the chunks back to the root node sorted_arr =
comm.gather(chunk_left, root=0) sorted_arr += chunk_middle

sorted_arr +=

comm.gather(chunk_right, root=0)

return sorted_arr
```

4. Generate the dataset and run the Quicksort Algorithms:

```
import random

# Generate a large dataset of numbers

arr = [random.randint(0, 1000) for _ in range(1000000)]

#       Time the serial version of Quicksort Algorithmimport time

start_time = time.time()
quicksort_serial(arr)

serial_time = time.time() - start_time
#Time the parallel version of Quicksort
Algorithmimport time

start_time = time.time()
quicksort_parallel(arr)
parallel_time = time.time() -
start_time
```

5.Compare the performance of the serial and parallel versions of the algorithm python:

```
if rank == 0:
   print(f"Serial Quicksort Algorithm time: {serial_time:.4f}
   seconds") print(f"Parallel Quicksort Algorithm time:
   {parallel_time:.4f} seconds")
```

Output:

Serial Quicksort Algorithm time: 1.5536

seconds Parallel Quicksort Algorithm time: 1.3488 seconds

Mini Project : 2

Title - Implement Huffman Encoding on GPU

Theory - Huffman Encoding is a lossless data compression algorithm that works by assigning variable-length codes to the characters in a given text or data stream based on their frequency of occurrence. This encoding scheme can be implemented on GPU to speed up the encoding process.

The variable-length codes assigned to input characters are Prefix Codes means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

Here's a possible implementation of Huffman Encoding on GPU:

1. Calculate the frequency of each character in the input text.
2, Construct a Huffman tree using the calculated frequencies. The tree can be built using a priorityqueue implemented on GPU, where the priority of a node is determined by its frequency.
3. Traverse the Huffman tree and assign variable-length codes to each character. The codes can begenerated using a depth-first search algorithm implemented on GPU.
4. Encode the input text using the generated Huffman codes.

To optimize the implementation for GPU, we can use parallel programming techniques such as CUDA, OpenCL, or HIP to parallelize the calculation of character frequencies, construction of theHuffman tree, and generation of Huffman codes.

Here are some specific optimizations that can be applied to each step:

1.Calculating character frequencies:

Use parallelism to split the input text into chunks and count the frequencies of each

character inparallel on different threads.

Reduce the results of each thread into a final frequency count on the GPU.1.
Constructing the Huffman tree:

Use a priority queue implemented on GPU to parallelize the building of the Huffman tree.
Each thread can process one or more nodes at a time, based on the priority of the nodes in the queue.

2.Generating Huffman codes:

Use parallelism to traverse the Huffman tree and generate Huffman codes for each character inparallel.
Each thread can process one or more nodes at a time, based on the depth of the nodes in the tree.

3.Encoding the input text:

Use parallelism to split the input text into chunks and encode each chunk in parallel on differentthreads.
Merge the encoded chunks into a final output on the GPU.

By parallelizing these steps, we can achieve significant speedup in the Huffman Encoding process onGPU. However, it's important to note that the specific implementation details may vary based on the programming language and GPU architecture being used.

Source Code -

```
// Count the frequency of each character in the input text
int freq_count[256] = {0};
int*  d_freq_count;
cudaMalloc((void**)&d_freq_count, 256 * sizeof(int));
cudaMemcpy(d_freq_count, freq_count, 256 * sizeof(int), cudaMemcpyHostToDevice);
int block_size = 256;
int grid_size = (input_size + block_size - 1) / block_size;
count_frequencies<<<grid_size, block_size>>>(input_text, input_size, d_freq_count);
cudaMemcpy(freq_count, d_freq_count, 256 * sizeof(int), cudaMemcpyDeviceToHost);

// Build the Huffman tree
HuffmanNode* root = build_huffman_tree(freq_count);
```

```
    // Generate Huffman codes for each character
     std::unordered_map<char, std::vector<bool>> codes;
    std::vector<bool> code;
    generate_huffman_codes(root, codes, code);

 // Encode the input text using the Huffman
 codesint output_size = 0;
for (int i = 0; i < input_size; i++)
{
output_size+= codes[input_text[i]].size();
}
output_size = (output_size + 7) / 8;
char* output_text = new char[output_size];
char* d_output_text;
cudaMalloc((void**)&d_output_text, output_size * sizeof(char));
cudaMemcpy(d_output_text,output_text,output_size*sizeof(char), cudaMemcpyHostToDevice);
encode_text<<<grid_size, block_size>>>(input_text, input_size, d_output_text, output_size,codes);
cudaMemcpy(output_text,d_output_text,output_size*sizeof(char), cudaMemcpyDeviceToHost);

    // Print the output
    std::cout << "Input text: " << input_text << std::endl;std::cout << "Encoded text: ";
    for (int i = 0; i < output_size; i++) {
        std::cout << std::bitset<8>(output_text[i]) << " ";
    }
    std::cout << std::endl;

    //     Freememory
    delete[]output_text;
    cudaFree(d_freq_count);
    cudaFree(d_output_text);
    delete root;
    return 0;
```

}

Output -
Input text: Hello, world!
Encoded text: 01000110 11010110 10001011 10101110 11110100 11011111 00101101 01000000 11111010

Mini Project : 3

Title - Implement Parallelization of Database Query optimization

Theory -

Query processing is the process through which a Database Management System (DBMS) parses, verifies, and optimizes a given query before creating low-level code that the DB understands.

Query Processing in DBMS, like any other High-Level Language (HLL) where code is first generated and then executed to perform various operations, has two phases: compile-time and runtime.

Query the use of declarative languages and query optimization is one of the main factors contributing to the success of RDBMS technology. Any database allows users to create queries to request specific data, and the database then uses effective methods to locate the requested data.

A database optimization approach based on CMP has been studied by numerous other academics. But the majority of their effort was on optimizing join operations while taking into account the L2-cache and the parallel buffers of the shared main memory.

The following techniques can be used to make a query parallel

- I/O parallelism
- Internal parallelism of queries
- Parallelism among queries
- Within-operation parallelism
- Parallelism in inter-operation

I/O parallelism :

This type of parallelism involves partitioning the relationships among the discs in order to speed upthe retrieval of relationships from the disc.

The inputted data is divided within, and each division is processed simultaneously. After processingall of the partitioned data, the results are combined. Another name for it is data partitioning.

Hash partitioning is best suited for point queries that are based on the partitioning attribute and havethe benefit of offering an even distribution of data across the discs.

It should be mentioned that partitioning is beneficial for the sequential scans of the full table stored on "n" discs and the speed at which the table may be scanned. For a single disc system, relationshiptakes around 1/n of the time needed to scan the table. In I/O parallelism, there are four different methods of partitioning:

Hash partitioning :

A hash function is a quick mathematical operation. The partitioning properties are hashed for eachrow in the original relationship.

Let's say that the data is to be partitioned across 4 drives, numbered disk1, disk2, disk3, and disk4.The row is now stored on disk3 if the function returns.

Range partitioning :

Each disc receives continuous attribute value ranges while using range partitioning. For instance, ifwe are range partitioning three discs with the numbers 0, 1, and 2, we may assign a relation with a value of less than 5 is written to disk0, numbers from 5 to 40 are sent to disk1, and values above 40 are written to disk2.

It has several benefits, such as putting shuffles on the disc that have attribute values within a specifiedrange.

Round-robin partitioning :

Any order can be used to study the relationships in this method. It sends the ith tuple to the discnumber (i% n).

Therefore, new rows of data are received by discs in turn. For applications that want to read the full relation sequentially for each query, this strategy assures an even distribution of tuples across drives.

Schema Partitioning :

Various tables inside a database are put on different discs using a technique called schemapartitioning.

Intra-query parallelism :

Using a shared-nothing paralleling architecture technique, intra-query parallelism refers to theprocessing of a single query in a parallel process on many CPUs.

This employs two different strategies:

First method — In this method, a duplicate task can be executed on a small amount of

data by eachCPU.

Second method — Using this method, the task can be broken up into various sectors, with each CPUcarrying out a separate subtask.

Inter-query parallelism

Each CPU executes numerous transactions when inter-query parallelism is used. Parallel transaction processing is what it is known as. To support inter-query parallelism, DBMS leverages transaction dispatching.

We can also employ a variety of techniques, such as effective lock management. This technique runseach query sequentially, which slows down the running time.

In such circumstances, DBMS must be aware of the locks that various transactions operating onvarious processes have acquired.

When simultaneous transactions don't accept the same data, inter-query parallelism on shared storage architecture works well.

Additionally, the throughput of transactions is boosted, and it is the simplest form of parallelism inDBMS.

Intra-operation parallelism :

In this type of parallelism, we execute each individual operation of a task, such as sorting, joins, projections, and so forth, in parallel. Intra-operation parallelism has a very high parallelism level. Database systems naturally employ this kind of parallelism. Consider the following SQL example: SELECT * FROM the list of vehicles and sort by model number;

Since a relation might contain a high number of records, the relational operation in the aforementioned query is sorting.

Because this operation can be done on distinct subsets of the relation in several processors, it takesless time to sort the data.

Inter-operation parallelism :

This term refers to the concurrent execution of many operations within a query expression. Theycome in two varieties:

Pipelined parallelism — In pipeline parallelism, a second operation consumes a row of the first operation's output before the first operation has finished producing the whole set of rows in its output.Additionally, it is feasible to perform these two processes concurrently on several CPUs, allowing oneoperation to consume tuples concurrently with another operation and thereby reduce them.

It is advantageous for systems with a limited number of CPUs and prevents the storage of interimresults on a disc.

Independent parallelism- In this form of parallelism, operations contained within query phrases thatare independent of one another may be carried out concurrently. This analogy is extremely helpful when dealing with parallelism of a lower degree.

Execution Of a Parallel Query :

The relational model has been favoured over prior hierarchical and network models because of commercial database technologies. Data independence and high-level query languages are the keyadvantages that relational database systems (RDBMSs) have over their forerunners (e.g., SQL).

The efficiency of programmers is increased, and routine optimization is encouraged. Additionally, distributed database management is made easier by the relational model's set-orientedstructure. RDBMSs may now offer performance levels comparable to older systems thanks to a decade of development and tuning.

They are therefore widely employed in the processing of commercial data for OLTP (online transaction processing) or decision-support systems. Through the use of many processors working together, parallel processing makes use of multiprocessor computers to run application programmesand boost performance.

It is most commonly used in scientific computing, which it does by the speed of numericalapplications' responses.

The development of parallel database systems is an example of how database management and parallel computing can work together. A given SQL statement can be divided up in the parallel database system PQO such that its components can run concurrently on several processors in a multi-processor machine.

Full table

As a form of parallel database optimization, Parallel Query enables the division of SELECT or DML operations into many smaller chunks that can be executed by PQ slaves on different CPUs in a single box.

The order of joins and the method for computing each join are fixed in the first part of the Fig, which is sorting and rewriting. The second phase, parallelization, turns the query tree into a parallel plan.

Parallelization divides this stage into two parts: extraction of parallelism and scheduling. Optimizing database queries is an important task in database management

systems to improve the performance of database operations. Parallelization of database query optimization can significantlyimprove query execution time by dividing the workload among multiple processors or nodes.

Here's an overview of how parallelization can be applied to database query optimization:

1. Partitioning: The first step is to partition the data into smaller subsets. The partitioning can be done based on different criteria, such as range partitioning, hash partitioning, or list partitioning. This can be done in parallel by assigning different processors or nodes to handle different parts of the partitioning process.

2. Query optimization: Once the data is partitioned, the next step is to optimize the queries. Query optimization involves finding the most efficient way to execute the query by considering factors suchas index usage, join methods, and filtering. This can also be done in parallel by assigning different processors or nodes to handle different parts of the query optimization process.

3. Query execution: After the queries are optimized, the final step is to execute the queries. The execution can be done in parallel by assigning different processors or nodes to handle different parts of the execution process. The results can then be combined to generate the final result set.

To implement parallelization of database query optimization, we can use parallel programming frameworks such as OpenMP or CUDA. These frameworks provide a set of APIs and tools to distribute the workload among multiple processors or nodes and to manage the synchronization andcommunication between them.

Here's an example of how we can parallelize the query optimization process using OpenMP:

```
//C++
//Partition the data
std::vector<std::vector<int>> partitions;
int num_partitions = omp_get_num_threads(); #pragma
omp parallel for
```

```
for (int i = 0; i < num_partitions; i++) {
   std::vector<int> partition = partition_data(data, i, num_partitions);
   partitions.push_back(partition);
}

// Optimize the queries in parallel #pragma
omp parallel for
for (int i = 0; i < num_queries; i++)
 {
  Query query = queries[i];
   int partition_id = get_partition_id(query, partitions); std::vector<int>
   partition = partitions[partition_id];
   optimize_query(query, partition);
}

// Execute the queries in parallel#pragma omp
parallel for
for (int i = 0; i < num_queries; i++)
{
  Query query = queries[i];
   int partition_id = get_partition_id(query, partitions); std::vector<int>
   partition = partitions[partition_id]; std::vector<int> result =
   execute_query(query, partition);merge_results(result);
}
```

In this example, we first partition the data into smaller subsets using OpenMP parallelism. Then we optimize each query in parallel by assigning different processors or nodes to handle different parts of the optimization process. Finally, we execute the queries in parallel by assigning different processorsor nodes to handle different parts of the execution process.

Parallelization of database query optimization can significantly improve the performance of database operations and reduce query execution time. However, it requires careful consideration of the workload distribution, synchronization, and communication between proces