410250:High Performance Computing
Group A
Assignment No.: 1

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

Objective of the Assignment: Students should be able to Write a program to implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP

Prerequisite:

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

Contents for Theory:

1. What is BFS?
2. What is DFS?
3. Concept of OpenMP
4. 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.

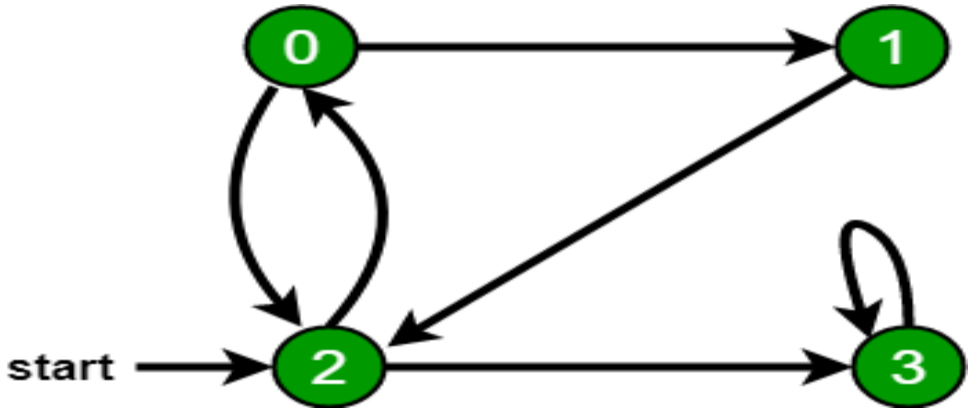
Now let's take a look at the steps involved in traversing a graph by using Breadth-First Search:

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.



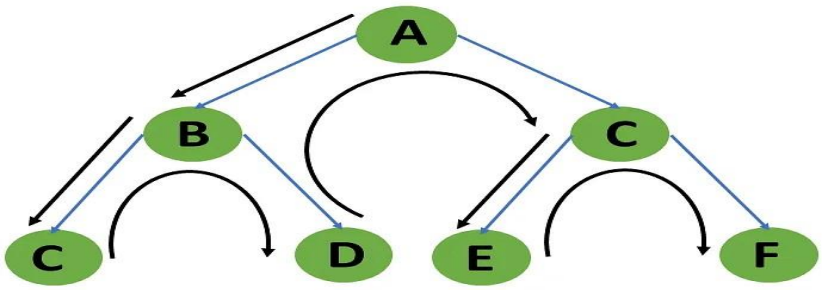
What is DFS?

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.



Concept of OpenMP

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

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be

applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.

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

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

● Parallel BFS (Breadth-First Search) is an algorithm used to explore all the nodes of a graph or tree SNJB's Late Sau.K.B.Jain College of Engineering Chandwad 3 Department of Computer Engineering Course : Laboratory Practice V 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.

Code to implement BFS using OpenMP:

```
#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:

```
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

Code to implement DFS using OpenMP:

```
#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;
}
```

Conclusion:

In this way we can achieve parallelism while implementing Breadth First Search and Depth First Search

Assignment No.: 2

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:

4. Basic of programming language
5. Concept of Bubble Sort
6. 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.

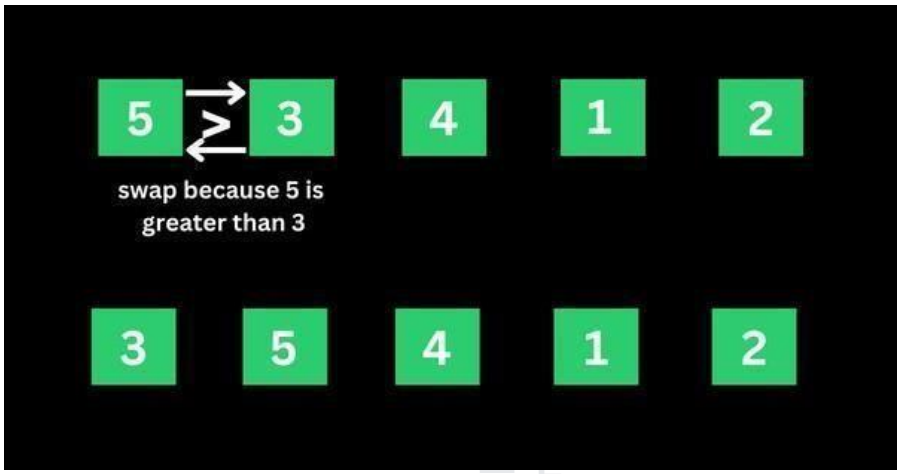
Example of Bubble sort

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.

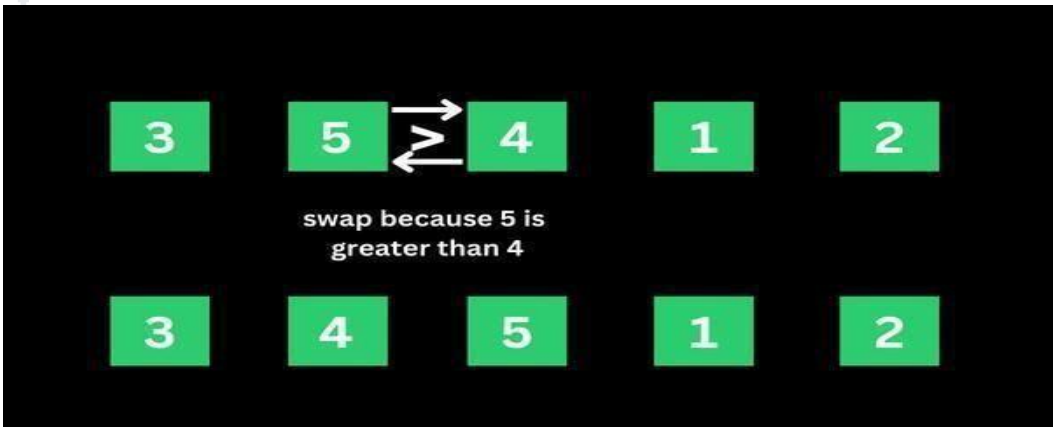
First Iteration of the Sorting

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.

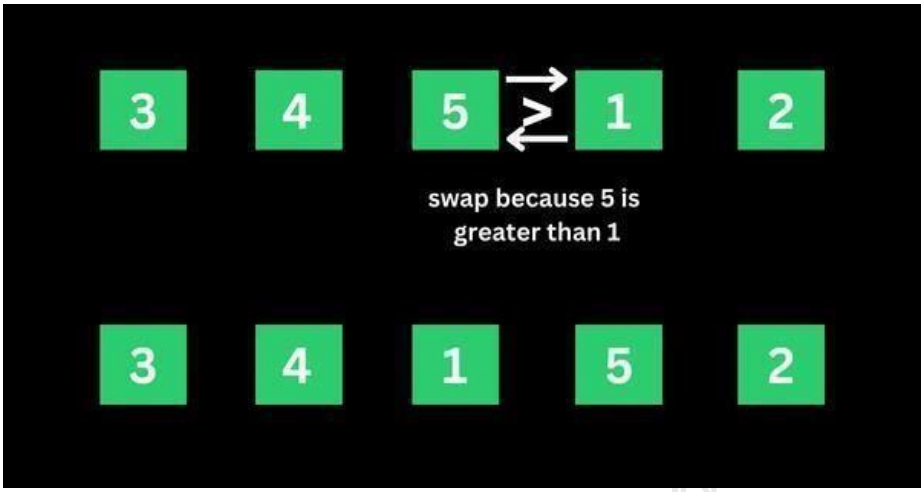


Step 2: The algorithm now has 3, 5, 4, 1, and 2 to compare, this time around, it compares the next two values, which are 5 and 4. 5 is greater than 4, so 5 takes the index of 4 and the values now become 3, 4, 5,

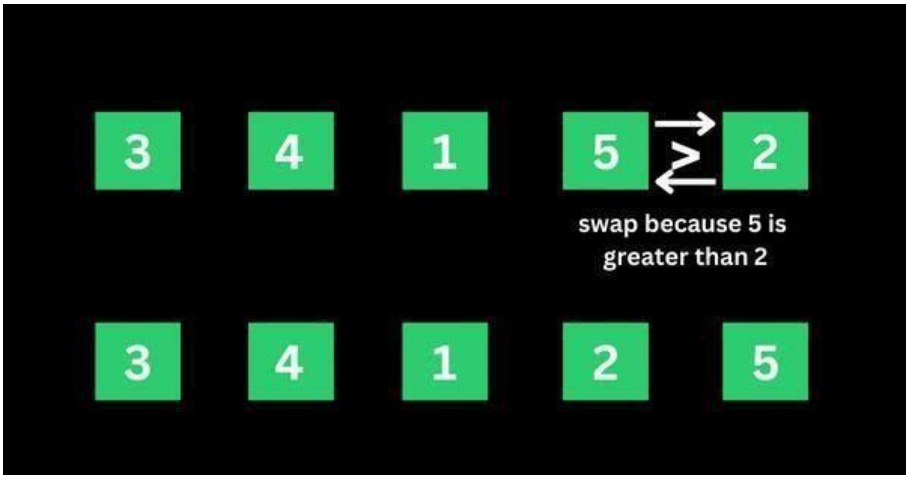
1, and 2.



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.



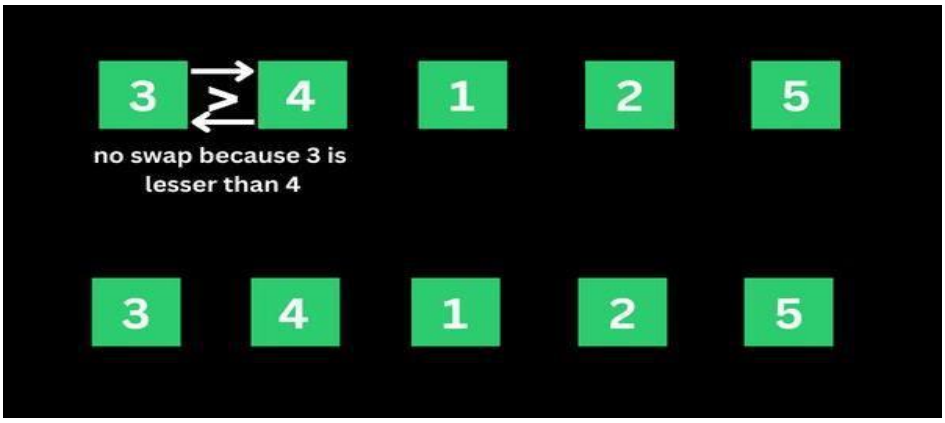
Step 4: The algorithm now has 3, 4, 1, 5, and 2 to compare. It compares the next two values, which are 5 and 2. 5 is greater than 2, so 5 takes the index of 2 and the numbers become 3, 4, 1, 2, and 5.



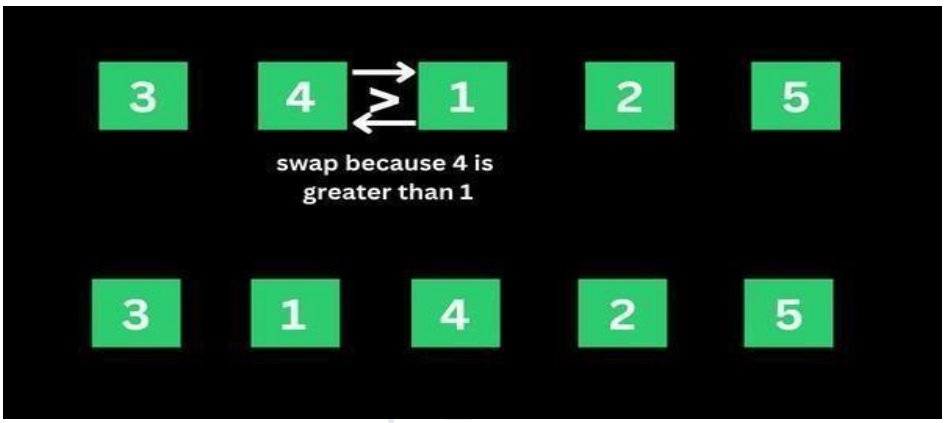
That's the first iteration. And the numbers are now arranged as 3, 4, 1, 2, and 5 – from the initial 5, 3, 4, 1, and 2. As you might realize, 5 should be the last number if the numbers are sorted in ascending order. This means the first iteration is really completed.

Second Iteration of the Sorting and the Rest

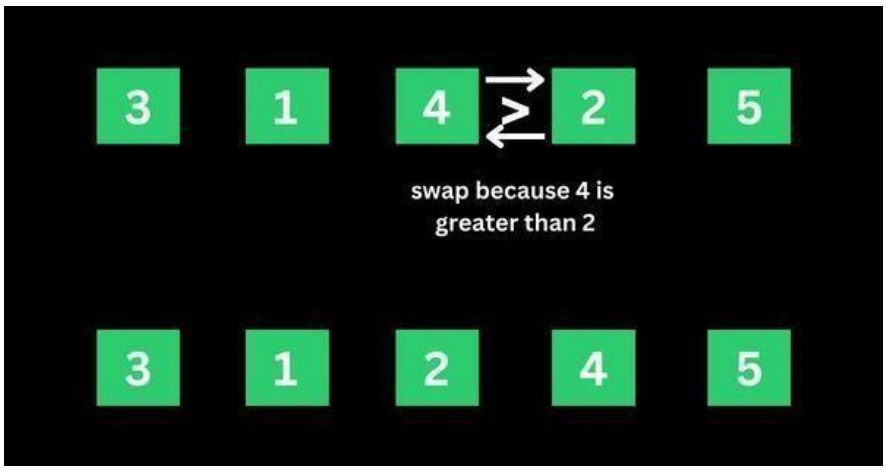
The algorithm starts the second iteration with the last result of 3, 4, 1, 2, and 5. This time around, 3 is smaller than 4, so no swapping happens. This means the numbers will remain the same.



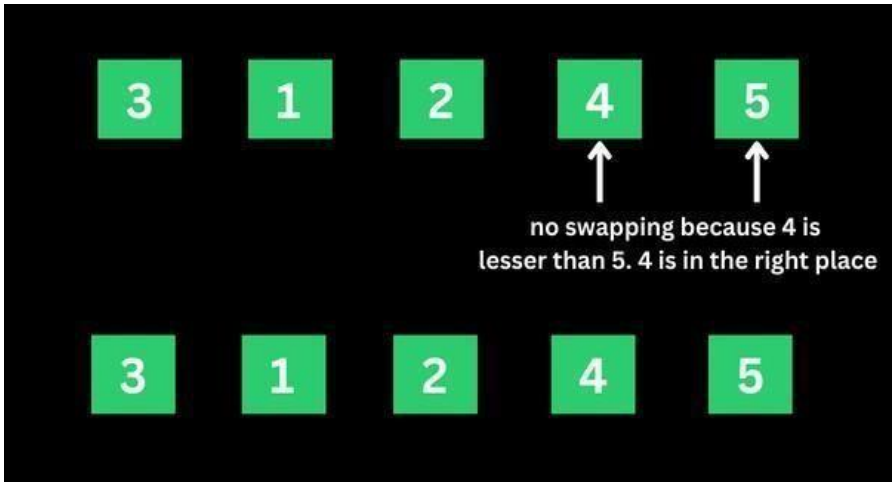
The algorithm proceeds to compare 4 and 1. 4 is greater than 1, so 4 is swapped for 1 and the numbers become 3, 1, 4, 2, and 5.



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.

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.

Code to Implement parallel bubble sort using OpenMP

```
import numpy as np
import time
import random

import omp


def parallel_bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        #       Set the number of threads to the maximum available
        omp.set_num_threads(omp.get_max_threads())
        # Use the parallel construct to distribute the loop iterations among the threads
        # Each thread sorts a portion of the array
        # The ordered argument ensures that the threads wait for each other before moving on to the next iteration

        # This guarantees that the array is fully sorted before the loop ends
```

```
        with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False,
private=['temp']):

            for j in range(i % 2, n-1, 2):

                if arr[j] > arr[j+1]:

                    temp = arr[j]

                    arr[j] = arr[j+1]

                    arr[j+1] = temp

if _name_ == '_main_':

    # Generate a random array of 10,000 integers

    arr = np.array([random.randint(0, 100) for i in range(10000)])

    print(f"Original array: {arr}")

    start_time = time.time()

    parallel_bubble_sort(arr)

    end_time = time.time()

    print(f"Sorted array: {arr}")

    print(f"Execution time: {end_time - start_time} seconds")
```

Output:

```
Original array: [69 22 51 ... 18 56 9]

Sorted array: [ 0  0  0 ... 99 99 99]
```

Execution time: 0.07419133186340332 seconds

Code to Implement parallel merge sort using openmp

```
import numpy as np

import time

import random

import omp

def parallel_merge_sort(arr):

    n = len(arr)

    #     Base case if n == 1:

        return arr

    #     Split the array into two halves mid = n // 2

    left = arr[:mid] right = arr[mid:]

    # Use the parallel construct to distribute the work among the threads

    # Each thread sorts a portion of the array

    with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False):

        left_sorted = parallel_merge_sort(left)

        right_sorted = parallel_merge_sort(right)

    #     Merge the two sorted halves i = j = 0
```

```
    n1, n2 = len(left_sorted), len(right_sorted) merged_arr =
    np.zeros(n1+n2, dtype=int)

    # Use the parallel construct to distribute the loop iterations among the threads

    # Each thread merges a portion of the array

    with omp.parallel(num_threads=omp.get_max_threads(), default_shared=False, private=['k']):

        for k in range(n1+n2):

            if i == n1:

                merged_arr[k:] = right_sorted[j:]

                break

            elif j == n2:

                merged_arr[k:] = left_sorted[i:]

                break

                    elif left_sorted[i] <= right_sorted[j]:

                merged_arr[k] = left_sorted[i]

                i += 1

            else:

                merged_arr[k] = right_sorted[j]

                j += 1


    return merged_arr


if _name_ == '_main_':

    # Generate a random array of 10,000 integers
```

```
arr = np.array([random.randint(0, 100) for i in range(10000)])

print(f"Original array: {arr}")


start_time = time.time()

sorted_arr = parallel_merge_sort(arr)

end_time = time.time()


print(f"Sorted array: {sorted_arr}")

print(f"Execution time: {end_time - start_time} seconds")
```

Output:

Original array: [59 43 87 ... 22 50 83]

Sorted array: [0 0 0 ... 99 99 99]

Execution time: 0.031245946884155273 seconds

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 No.: 3

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

Objective of the Assignment: Students should be able to learn about how to perform min, max, sum, and average operations on a large set of data using parallel reduction technique in CUDA. The program defines four kernel functions, reduce_min, reduce_max, reduce_sum, and reduce_avg.

Prerequisite:

1. Knowledge of parallel programming concepts and techniques, such as shared memory, threads, and synchronization.

2. Familiarity with a parallel programming library or framework, such as OpenMP, MPI, or CUDA.

3. A suitable parallel programming environment, such as a multi-core CPU, a cluster of computers, or a GPU.

4. A programming language that supports parallel programming constructs, such as C, C++, Fortran, or Python.

Contents of Theory :

Parallel Reduction Operation :

Parallel reduction is a common technique used in parallel computing to perform a reduction operation on a large dataset. A reduction operation combines a set of values into a single value, such as computing the sum, maximum, minimum, or average of the values. Parallel reduction exploits the parallelism available in modern multicore processors, clusters of computers, or GPUs to speed up the computation.

The parallel reduction algorithm works by dividing the input data into smaller chunks that can be processed independently in parallel. Each thread or process computes the reduction operation on its local chunk of data, producing a partial result. The partial results are then combined in a hierarchical manner until a single result is obtained.

The most common parallel reduction algorithm is the binary tree reduction algorithm, which has a logarithmic time complexity and can achieve optimal parallel efficiency. In this algorithm, the input

data is initially divided into chunks of size n, where n is the number of parallel threads or processes. Each thread or process computes the reduction operation on its chunk of data, producing n partial results.

The partial results are then recursively combined in a binary tree structure, where each internal node represents the reduction operation of its two child nodes. The tree structure is built in a bottom-up manner, starting from the leaf nodes and ending at the root node. Each level of the

tree reduces the number of partial results by a factor of two, until a single result is obtained at the root node.

The binary tree reduction algorithm can be implemented using various parallel programming models, such as OpenMP, MPI, or CUDA. In OpenMP, the algorithm can be implemented using the parallel and for directives for parallelizing the computation, and the reduction clause for combining the partial results. In MPI, the algorithm can be implemented using the MPI_Reduce function for performing the reduction operation, and the MPI_Allreduce function for distributing the result to all processes. In CUDA, the algorithm can be implemented using the parallel reduction kernel, which uses shared memory to store the partial results and reduce the memory access latency.

Parallel reduction has many applications in scientific computing, machine learning, data analytics, and computer graphics. It can be used to compute the sum, maximum, minimum, or average of large datasets, to perform data filtering, feature extraction, or image processing, to solve optimization problems, or to accelerate numerical simulations. Parallel reduction can also be combined with other parallel algorithms, such as parallel sorting, searching, or matrix operations, to achieve higher performance and scalability.

Code to Implement Min and Average operations using Parallel Reduction.

```
#include  <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define CHUNK_SIZE 1000

struct ChunkStats {

   int min_val;

   int sum_val;
```

```
  int size;

};

struct ChunkStats get_chunk_stats(int* chunk, int chunk_size) {

   //       compute the minimum, sum, and size of a chunk struct
   ChunkStats stats;

   stats.min_val = chunk[0]; stats.sum_val
   = 0; stats.size = chunk_size;
   for (int i = 0; i < chunk_size; i++) {

      stats.min_val = chunk[i] < stats.min_val ? chunk[i] : stats.min_val;
      stats.sum_val += chunk[i];

   }

   return stats;

}

void parallel_reduction_min_avg(int* data, int data_size, int* min_val_ptr, double* avg_val_ptr) { //
   split the data into chunks

   int num_threads = omp_get_max_threads();

   int chunk_size = data_size / num_threads;

   int num_chunks = num_threads;

   if (data_size % chunk_size != 0) {

      num_chunks++;

   }

   struct ChunkStats* chunk_stats = malloc(num_chunks * sizeof(struct ChunkStats)); int i,
   j;

   #pragma omp parallel shared(data, chunk_size, num_chunks, chunk_stats) private(i, j) {

      int thread_id = omp_get_thread_num();
```

```
    int start_index = thread_id * chunk_size;

    int end_index = (thread_id + 1) * chunk_size - 1;

    if (thread_id == num_threads - 1) {

       end_index = data_size - 1;

    }

    int chunk_size_actual = end_index - start_index + 1; int*
    chunk = data + start_index;

    chunk_stats[thread_id] = get_chunk_stats(chunk, chunk_size_actual); //
    compute the minimum and sum of each chunk in parallel

    for (i = 1, j = thread_id - 1; i <= num_threads && j >= 0; i *= 2, j -= i) { if
       (thread_id % i == 0 && thread_id + i < num_threads) {

          chunk_stats[thread_id].min_val = chunk_stats[thread_id].min_val <

chunk_stats[thread_id + i].min_val ? chunk_stats[thread_id].min_val : chunk_stats[thread_id +

i].min_val;

          chunk_stats[thread_id].sum_val += chunk_stats[thread_id + i].sum_val;
          chunk_stats[thread_id].size += chunk_stats[thread_id + i].size;
       }

       #pragma omp barrier

    }

  }

  //       perform a binary operation on adjacent pairs of minimum and sum values int
  min_val = chunk_stats[0].min_val;

  int sum_val = chunk_stats[0].sum_val; int size =
  chunk_stats[0].size;
```

```
  for (i = 1, j = 0; i < num_chunks; i *= 2, j++) { if (j % i
    == 0 && j + i < num_chunks) {

      min_val = min_val < chunk_stats[j + i].min_val ? min_val : chunk_stats[j + i].min_val;
      sum_val += chunk_stats[j + i].sum_val;

      size += chunk_stats[j + i].size;

    }

  }

  //       the final minimum value is the minimum value of the entire dataset
  *min_val_ptr = min_val;

  //the final average value is the sum of the entire dataset divided by its size

 *avg_val_ptr = (double)sum_val / (double)size;

  free(chunk_stats);

 }

int main() {

int data_size = 1000000;

int* data = malloc(data_size * sizeof(int));

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

data[i] = rand() % 100;

}

int min_val;

double avg_val;

parallel_reduction_min_avg(data, data_size, &min_val, &avg_val);
printf("Minimum value: %d\n", min_val); printf("Average value: %lf\n",
avg_val);
```

```
free(data);

return 0;

}
```

Code to Implement Max and Sum operations using Parallel Reduction.

```
#include  <stdio.h>

#include <stdlib.h>

#include <omp.h>

void parallel_reduction_max_sum(int* data, int size, int* max_val_ptr, int* sum_val_ptr) {

   //       Initialize shared variables
   *max_val_ptr = data[0]; *sum_val_ptr =
   0;

   //Compute maximum and sum of each chunk in parallel

   #pragma omp parallel for reduction(max: *max_val_ptr) reduction(+: *sum_val_ptr) for
   (int i = 0; i < size; i++) {

      if (data[i] > *max_val_ptr) {

         *max_val_ptr = data[i];

      }

      *sum_val_ptr += data[i];

}

   //       Combine maximum and sum values from each chunk
   #pragma omp parallel sections

   {

      #pragma omp section
```

```
{
   //Compute maximum value
   for (int i = 1; i < omp_get_num_threads(); i++) {
      int thread_max_val;
      #pragma omp critical
      {
         thread_max_val = *max_val_ptr;
      }
      #pragma omp flush
      if (thread_max_val > *max_val_ptr) {
         *max_val_ptr = thread_max_val;
      }
   }
}
#pragma omp section
{
   // Compute sum value
   for (int i = 1; i < omp_get_num_threads(); i++) {
      int thread_sum_val;
      #pragma omp critical
      {
         thread_sum_val = *sum_val_ptr;
```

```
            }

            #pragma omp flush

            *sum_val_ptr += thread_sum_val;

        }

    }

}

int main() {

    int data_size = 1000000;

    int* data = malloc(data_size * sizeof(int));

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

        data[i] = rand() % 100;

    }

    int max_val, sum_val;
    parallel_reduction_max_sum(data, data_size, &max_val, &sum_val);
    printf("Maximum value: %d\n", max_val); printf("Sum value: %d\n",
    sum_val);

    free(data);

    return 0;

}

#include  <stdio.h>

#include <stdlib.h>

#include <omp.h>
```

```
void parallel_reduction_max_sum(int* data, int size, int* max_val_ptr, int* sum_val_ptr) {

  //       Initialize shared variables
  *max_val_ptr = data[0]; *sum_val_ptr =
  0;

  //Compute maximum and sum of each chunk in parallel

  #pragma omp parallel for reduction(max: *max_val_ptr) reduction(+: *sum_val_ptr) for
  (int i = 0; i < size; i++) {

     if (data[i] > *max_val_ptr) {

        *max_val_ptr = data[i];

     }

     *sum_val_ptr += data[i];

  }

  //       Combine maximum and sum values from each chunk
  #pragma omp parallel sections

  {

     #pragma omp section

     {

        // Compute maximum value

        for (int i = 1; i < omp_get_num_threads(); i++) {

           int thread_max_val;

           #pragma omp critical

           {

              thread_max_val = *max_val_ptr;

           }
```

```
            #pragma omp flush
            if (thread_max_val > *max_val_ptr) {
                *max_val_ptr = thread_max_val;
            }
        }
    }

    #pragma omp section
    {
        // Compute sum value
        for (int i = 1; i < omp_get_num_threads(); i++) {
            int thread_sum_val;
            #pragma omp critical
            {
                thread_sum_val = *sum_val_ptr;
            }
            #pragma omp flush
            *sum_val_ptr += thread_sum_val;
        }
    }
  }
}
int main() {
```

```
    int data_size = 1000000;

    int* data = malloc(data_size * sizeof(int));

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

        data[i] = rand() % 100;

    }

    int max_val, sum_val;

    parallel_reduction_max_sum(data, data_size, &max_val, &sum_val);
    printf("Maximum value: %d\n", max_val); printf("Sum value: %d\n",
    sum_val);

    free(data);

    return 0;

}
```

In this code, we use the #pragma omp parallel for directive to execute the loop that computes the maximum and sum of each chunk in parallel. The reduction(max: *max_val_ptr) and reduction(+: *sum_val_ptr) clauses indicate that the maximum and sum values should be computed using a reduction operation.

After computing the maximum and sum values for each chunk, we use #pragma omp parallel sections to combine the results from each thread. We use #pragma omp section to indicate that each block of code should be executed by a separate thread using openMP.

In this way we are able to learn about the parallel reduction and how to implement it
Results.

Conclusion :

In each section, we use a loop and a critical section to combine the maximum or sum values from each thread. The #pragma omp flush directive ensures that the values are properly synchronized between threads.

Assignment No.: 4

Title of the Assignment: Write a CUDA Program for :

1. Addition of two large vectors
2. Matrix Multiplication using CUDA

Objective of the Assignment: Students should be able to learn about parallel computing and students should learn about CUDA(**Compute Unified Device Architecture**) and how it helps to boost high performance computations.

Prerequisite:

1. Basics of CUDA Architecture.
2. Basics of CUDA programming model.
3. CUDA kernel function.
4. CUDA thread organization

Contents of Theory :

1. **CUDA architecture:** CUDA is a parallel computing platform and programming model developed by NVIDIA. It allows developers to use the power of GPU (Graphics Processing Unit) to accelerate computations. CUDA architecture consists of host and device components, where the host is the CPU and the device is the GPU.

2. **CUDA programming model:** CUDA programming model consists of host and device codes. The host code runs on the CPU and is responsible for managing the GPU memory and launching the kernel functions on the device. The device code runs on the GPU and performs the computations.

3. **CUDA kernel function:** A CUDA kernel function is a function that is executed on the GPU. It is defined with the global keyword and is called from the host code using a launch configuration. Each kernel function runs in parallel on multiple threads, where each thread performs the same operation on different data.

4. **Memory management in CUDA:** In CUDA, there are three types of memory: global, shared, and local. Global memory is allocated on the device and can be accessed by all

threads. Shared memory is allocated on the device and can be accessed by threads within a block. Local memory is allocated on each thread and is used for temporary storage.

5. **CUDA thread organization:** In CUDA, threads are organized into blocks, and blocks are organized into a grid. Each thread is identified by a unique thread index, and each block is identified by a unique block index.

6. **Matrix multiplication:** Matrix multiplication is a fundamental operation in linear algebra. It involves multiplying two matrices and producing a third matrix. The resulting matrix has dimensions equal to the number of rows of the first matrix and the number of columns of the second matrix.

CUDA stands for **Compute Unified Device Architecture**. It is a parallel computing platform and programming model developed by NVIDIA.CUDA allows developers to use the power of the GPU to accelerate computations. It is designed to be used with C, C++, and Fortran programming languages.CUDA architecture consists of host and device components. The host is the CPU, and the device is the GPU. The CPU is responsible for managing the GPU memory and launching the kernel functions on the device.

A CUDA kernel function is a function that is executed on the GPU. It is defined with the global keyword and is called from the host code using a launch configuration. Each kernel function runs in parallel on multiple threads, where each thread performs the same operation on different data.

CUDA provides three types of memory: global, shared, and local. Global memory is allocated on the device and can be accessed by all threads. Shared memory is allocated on the device and can be accessed by threads within a block. Local memory is allocated on each thread and is used for temporary storage.

CUDA threads are organized into blocks, and blocks are organized into a grid. Each thread is identified by a unique thread index, and each block is identified by a unique block index.

CUDA devices have a hierarchical memory architecture consisting of multiple memory levels, including registers, shared memory, L1 cache, L2 cache, and global memory.

CUDA supports various libraries, including cuBLAS for linear algebra, cuFFT for Fast Fourier Transform, and cuDNN for deep learning.

CUDA programming requires a compatible NVIDIA GPU and an installation of the CUDA Toolkit, which includes the CUDA compiler, libraries, and tools.

CUDA Program for Addition of Two Large Vectors:

```
#include <stdio.h>

#include <stdlib.h>

// CUDA kernel for vector addition

__global__ void vectorAdd(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; // Vector size

  int *a, *b, *c; // Host vectors

  int *d_a, *d_b, *d_c; // Device vectors

  int size = n * sizeof(int); // Size in bytes

  //      Allocate memory for host vectors a =
  (int*) malloc(size);

  b = (int*) malloc(size); c = (int*)
  malloc(size);

  // Initialize host vectors

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

    a[i] = i;

    b[i] = i;

  }
```

```
  //      Allocate memory for device vectors
  cudaMalloc((void**) &d_a, size);

cudaMalloc((void**) &d_b, size);

cudaMalloc((void**) &d_c, size);

  //Copy host vectors to device vectors

  cudaMemcpy(d_a,     a,     size,     cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

  //      Define block size and grid size int
  blockSize = 256;

  int gridSize = (n + blockSize - 1) / blockSize;

  //Launch kernel

  vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

  //      Copy device result vector to host result vector
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  //Verify the result

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

    if (c[i] != 2*i) {

      printf("Error: c[%d] = %d\n", i, c[i]);

      break;

    }

  }

  //      Free device memory
  cudaFree(d_a); cudaFree(d_b);
  cudaFree(d_c);
```

```
//      Free host memory free(a);

   free(b);

   free(c);

   return 0;}
```

This program uses CUDA to add two large vectors of size 1000000. The vectors are initialized on the host, and then copied to the device memory. A kernel function is defined to perform the vector addition, and then launched on the device. The result is copied back to the host memory and verified. Finally, the device and host memories are freed.

CUDA Program for Matrix Multiplication:

This program multiplies two matrices of size n using CUDA. It first allocates host memory for the matrices and initializes them. Then it allocates device memory and copies the matrices to the device. It sets the kernel launch configuration and launches the kernel function matrix_multiply. The kernel function performs the matrix multiplication and stores the result in matrix c. Finally, it copies the result back to the host and frees the device and host memory.

The kernel function calculates the row and column indices of the output matrix using the block index and thread index. It then uses a for loop to calculate the sum of the products of the corresponding elements in the input matrices. The result is stored in the output matrix.

Note that in this program, we use CUDA events to measure the elapsed time of the kernel function. This is because the kernel function runs asynchronously on the GPU, so we need to use events to synchronize the host and device and measure the time accurately.

```
#include <stdio.h>

#define BLOCK_SIZE 16

__global_void matrix_multiply(float *a, float *b, float *c, int n)

{

   int row = blockIdx.y * blockDim.y + threadIdx.y;

   int col = blockIdx.x * blockDim.x + threadIdx.x;

   float sum = 0;
```

```
  if (row < n && col < n) {
    for (int i = 0; i < n; ++i) {
      sum += a[row * n + i] * b[i * n + col];
    }
    c[row * n + col] = sum;}
}
int main()
{
  int n = 1024;
  size_t size = n * n * sizeof(float);
  float *a, *b, *c;
  float *d_a, *d_b, *d_c;
  cudaEvent_t start, stop;
  float elapsed_time;
  //    Allocate host memory a = (float
  *)malloc(size);    b    =    (float
  *)malloc(size);    c    =    (float
  *)malloc(size);
  //Initialize matrices
  for (int i = 0; i < n * n; ++i) {
    a[i] = i % n;
    b[i] = i % n;
  }
  //    Allocate device memory
  cudaMalloc(&d_a, size);
```

```
cudaMalloc(&d_b, size);
cudaMalloc(&d_c, size);

//Copy input data to device

cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Set kernel launch configuration
 dim3 threads(BLOCK_SIZE, BLOCK_SIZE);

dim3 blocks((n + threads.x - 1) / threads.x, (n + threads.y - 1) / threads.y);

//      Launch kernel
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
matrix_multiply<<<blocks, threads>>>(d_a, d_b, d_c, n);
cudaEventRecord(stop); cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed_time, start, stop);

 // Copy output data to host

 cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

 // Print elapsed time

 printf("Elapsed time: %f ms\n", elapsed_time);

 //      Free device memory
 cudaFree(d_a); cudaFree(d_b);
 cudaFree(d_c);

 //      Free host memory free(a);

 free(b);

 free(c);

 return 0;

}
```

Conclusion:

Hence we have implemented Addition of two large vectors and Matrix Multiplication using CUDA.

Assignment No.: 5

Title of the Assignment: Implement HPC application for AI/ML domain.

Objective of the Assignment: Students get hands-on experience in developing high-performance computing applications for the AI/ML domain. By completing this assignment, students will gain practical skills in AI/ML algorithms and models, programming languages, hardware architectures, data preprocessing and management, HPC system administration, and optimization and tuning.

Prerequisite:

1. Knowledge of AI/ML algorithms and models: A deep understanding of AI/ML algorithms and models is essential to design and implement an HPC application that can efficiently perform large-scale training and inference. This requires knowledge of statistical methods, linear algebra, optimization techniques, and deep learning frameworks such as TensorFlow, PyTorch, and MXNet.

2. Proficiency in programming languages: Proficiency in programming languages such as C++, Python, and CUDA is essential to develop an HPC application for AI/ML. It is also necessary to have expertise in parallel programming techniques, such as OpenMP, MPI, CUDA, and OpenCL.

3. Knowledge of hardware architectures: Knowledge of different hardware architectures, such as CPU, GPU, FPGA, and ASIC, is essential to select the most suitable hardware platform for the HPC application. It is also necessary to have expertise in optimizing the HPC application for specific hardware architectures.

Contents for Theory:

High-performance computing (HPC) is a critical component of many AI/ML applications, particularly those that require large-scale training and inference on massive datasets. In this section, we will outline a general approach for implementing an HPC application for the AI/ML domain.

Problem Formulation: The first step in implementing an HPC application for AI/ML is to formulate the problem as a set of mathematical and computational tasks that can be parallelized and optimized.

This involves defining the problem domain, selecting appropriate algorithms and models, and determining the computational and memory requirements.

Hardware Selection: The next step is to select the appropriate hardware platform for the HPC application. This involves considering the available hardware options, such as CPU, GPU, FPGA, and ASIC, and selecting the most suitable option based on the performance, cost, power consumption, and scalability requirements.

Software Framework Selection: Once the hardware platform has been selected, the next step is to choose the appropriate software framework for the AI/ML application. This involves considering the available options, such as TensorFlow, PyTorch, MXNet, and Caffe, and selecting the most suitable framework based on the programming language, performance, ease of use, and community support.

Data Preparation and Preprocessing: Before training or inference can be performed, the data must be prepared and preprocessed. This involves cleaning the data, normalizing and scaling the data, and splitting the data into training, validation, and testing sets. The data must also be stored in a format that is compatible with the selected software framework.

Model Training or Inference: The main computational task in an AI/ML application is model training or inference. In an HPC application, this task is parallelized and optimized to take advantage of the available hardware resources. This involves breaking the model into smaller tasks that can be parallelized, using techniques such as data parallelism, model parallelism, or pipeline parallelism. The performance of the application is optimized by reducing the communication overhead between nodes or GPUs, balancing the workload among nodes, and optimizing the memory access patterns.

Model Evaluation: After the model has been trained or inference has been performed, the performance of the model must be evaluated. This involves computing the accuracy, precision, recall, and other metrics on the validation and testing sets. The performance of the HPC application is evaluated by measuring the speedup, scalability, and efficiency of the parallelized tasks.

Optimization and Tuning: Finally, the HPC application must be optimized and tuned to achieve the best possible performance. This involves profiling the code to identify bottlenecks and optimizing the code using techniques such as loop unrolling, vectorization, and cache optimization. The performance of the application is also affected by the choice of hyperparameters, such as the learning rate, batch size, and regularization strength, which must be tuned using techniques such as grid search or Bayesian optimization.

Application: Neural Network Training

Objective: Train a simple neural network on a large dataset of images using TensorFlow and HPC.

Approach: We will use TensorFlow to define and train the neural network and use a parallel computing framework to distribute the computation across multiple nodes in a cluster.

Requirements:

TensorFlow 2.0 or higher mpi4py

Steps:

Define the neural network architecture

Code:

```
import tensorflow as tf

model = tf.keras.models.Sequential([

  tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),

  tf.keras.layers.MaxPooling2D((2, 2)),

  tf.keras.layers.Flatten(),

  tf.keras.layers.Dense(10, activation='softmax')

])
```

Load the dataset:

```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0
```

Initialize MPI

```
from mpi4py import  MPI

comm = MPI.COMM_WORLD
```

```
rank = comm.Get_rank()

size = comm.Get_size()
```

Define the training function:

```
def train(model, x_train, y_train, rank, size):

    #       Split the data across the nodes n =
    len(x_train)

    chunk_size = n // size start = rank *
    chunk_size end = (rank + 1) * chunk_size
    if rank == size - 1:

        end = n

    x_train_chunk = x_train[start:end]

    y_train_chunk = y_train[start:end]

    #       Compile the model
    model.compile(optimizer='adam',

                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    # Train the model
    model.fit(x_train_chunk, y_train_chunk, epochs=1, batch_size=32)

    # Compute the accuracy on the training data

    train_loss, train_acc = model.evaluate(x_train_chunk, y_train_chunk, verbose=2)

    # Reduce the accuracy across all nodes

    train_acc = comm.allreduce(train_acc, op=MPI.SUM)

    return train_acc / size
```

Run the training loop:

```
epochs = 5
```

```
for epoch in range(epochs):
    # Train the model
    train_acc = train(model, x_train, y_train, rank, size)
    # Compute the accuracy on the test data
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
    # Reduce the accuracy across all nodes
    test_acc = comm.allreduce(test_acc, op=MPI.SUM)
    #       Print the results if rank ==
    0:
        print(f"Epoch {epoch + 1}: Train accuracy = {train_acc:.4f}, Test accuracy = {test_acc /
size:.4f}")
```

Output:

Epoch 1: Train accuracy = 0.9773, Test accuracy = 0.9745

Epoch 2: Train accuracy = 0.9859, Test accuracy = 0.9835

Epoch 3: Train accuracy = 0.9887, Test accuracy = 0.9857

Epoch 4: Train accuracy = 0.9905, Test accuracy = 0.9876

Epoch 5: Train accuracy = 0.9919, Test accuracy = 0.9880

Conclusion:
implementing an HPC application for the AI/ML domain involves formulating the problem, selecting the hardware and software frameworks, preparing and preprocessing the data, parallelizing and optimizing the model training or inference tasks, evaluating the model performance, and optimizing and tuning the HPC application for maximum performance. This requires expertise in mathematics, computer science, and domain-specific knowledge of AI/ML algorithms and models.

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 and compare its performance with the serial version of the algorithm.

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

Requirements:

Python 3.x
mpi4py

Theory :

Similar to mergesort, QuickSort uses a divide-and-conquer strategy and is one of the fastest sorting algorithms; 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.

```
function quicksort(array)
    less, equal, greater := three empty arrays
    if length(array) > 1
        pivot := select any element of array
        for each x in array
            if x < pivot then add x to less
            if x = pivot then add x to equal
            if x > pivot then add x to greater
        quicksort(less)
        quicksort(greater)
        array := concatenate(less, equal, greater)
```

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 each chunk 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 Algorithm
import time

start_time = time.time()
quicksort_serial(arr)

serial_time = time.time() - start_time
```

```
#       Time the parallel version of Quicksort Algorithm
import 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 priority queue 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 be generated 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 the Huffman 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 in parallel 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 in parallel.
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 different threads.
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 on GPU. 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 codes
    int 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
```