AISSMS
COLLEGE OF ENGINEERING
Approved by AICTE, New Delhi, Recognized by
Govt. of Maharashtra, Affiliated to Savitribai Phule Pune University
and recognized 2(f) and 12(B) by UGC (Id.No. PU / PN/ Engg. / 093 (1992)
Accredited by NAAC with 'A+' Grade

Institute Vision

Service to Society through Quality Education

Institute Mission

- Generation of national wealth through academics and research.
- Imparting quality technical education at the cost affordable to all strata of Society
- Enhancing the quality of life through sustainable development.
- Carrying out high quality intellectual work.
- Achieving distinction of the highest preferred engineering colleges in the eyes of stake holders

Department Vision

Contributing to the welfare of society through technical and quality education.

Department Mission

M1: To produce best quality computer science professionals by imparting quality training, hands on experience and value education.

M2: To strengthen links with Industry through partnerships and collaborative developmental works.

M3: To attain self-sustainability and overall development through research, consultancy and development activities.

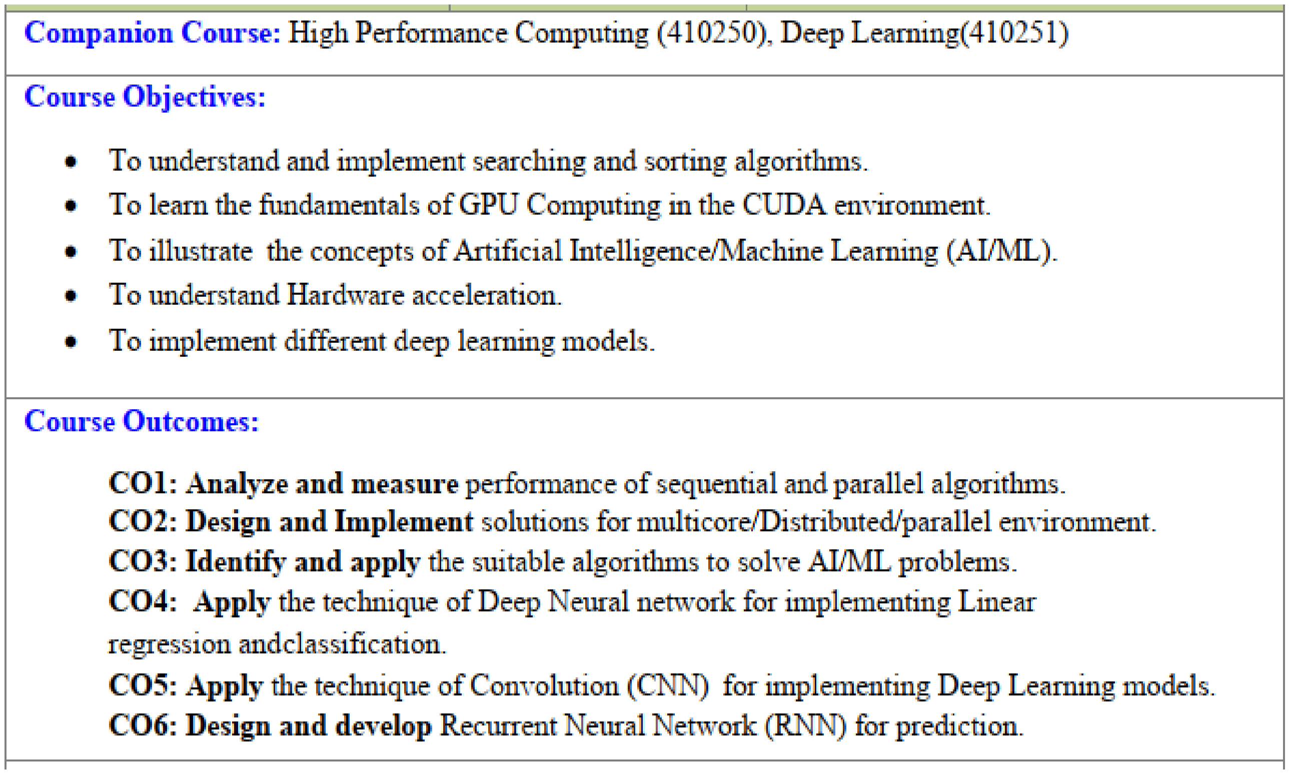
M4: To extend technical expertise to other technical institutions of the region and play a lead role to impart technical education.

Program Outcomes (POs)

PO1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSO)

PSO1	**Professional Skills-**The ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, big data analytics, and networking for efficient design of computer-based systems of varying complexities.
PSO2	**Problem-Solving Skills-** The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business success.
PSO3	**Successful Career and Entrepreneurship-** The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, and a zest for higher studies.



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 andclassification.
CO5: Apply the technique of Convolution (CNN) for implementing Deep Learning models.
CO6: Design and develop Recurrent Neural Network (RNN) for prediction.

AISSMS
COLLEGE OF ENGINEERING
Approved by AICTE, New Delhi, Recognized by
Govt. of Maharashtra, Affiliated to Savitribai Phule Pune University
and recognized 2(f) and 12(B) by UGC (Id.No. PU / PN/ Engg. / 093 (1992)
Accredited by NAAC with 'A+' Grade

LABORATORY MANUAL

LABORATORY PRACTICE – V 410255

BE-COMP

SEMESTER-VIII

TEACHING SCHEME	**EXAMINATION SCHEME**	
Practical: 2 Hrs/Week	**Practical:**	**50 Marks**
	TW:	**50 Marks**

Prepared by,

Mr. V. S. Gunjal

DEPARTMENT OF COMPUTER ENGINEERING
AISSMS College of Engineering, PUNE,

2022-2023

INDEX

410251 : Deep Learning

Any 3 Assignments and 1 Mini Project are Mandatory				
Group 1				
1.	Linear regression by using Deep Neural network: Implement Boston housing price prediction problem by Linear regression using Deep Neural network. Use Boston House price prediction dataset.			
2.	Classification using Deep neural network (Any One from the following) 1. Multiclass classification using Deep Neural Networks: Use the OCR letter recognition dataset Example: https://archive.ics.uci.edu/ml/datasets/letter+recognition 2. Binary classification using Deep Neural Networks Example: Classify movie reviews into positive" reviews and "negative" reviews, just based on the text content of the reviews. Use IMDB dataset			
3.	Convolutional neural network (CNN) (Any One from the following) • Use any dataset of plant disease and design a plant disease detection system using CNN. • Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories.			
4.	Recurrent neural network (RNN) Use the Google stock prices dataset and design a time series analysis and prediction system using RNN.			
Group 2				
5.	Mini Project: Human Face Recognition			
6.	Mini Project: Gender and Age Detection: predict if a person is a male or female and also their age			
7.	Mini Project: Colorizing Old B&W Images: color old black and white images to colorful images			

GROUP 1: ASSIGNMENTS

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 understand and implement searching algorithms like BFS & DFS.

Outcome:

Students will be able to Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP.

Pre-requisites:

64-bit Open source Linux or its derivative

Programming Languages:

C++/JAVA/PYTHON/R

Theory:

1. **Breadth-First Search:**

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

Graph traversals

Graph traversal means visiting every 1 vertex and edge exactly once in a well-defined order. While using certain graph 2 algorithms, you must ensure that each vertex of the graph is visited exactly once. The order 3 in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you 4 track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS)

BFS stands for Breadth-First Search.

It is a graph traversal algorithm used 1 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 2 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 3 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, 4 such as finding the shortest path between two nodes, solving puzzles, and searching through a tree or graph.

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

Example of BFS

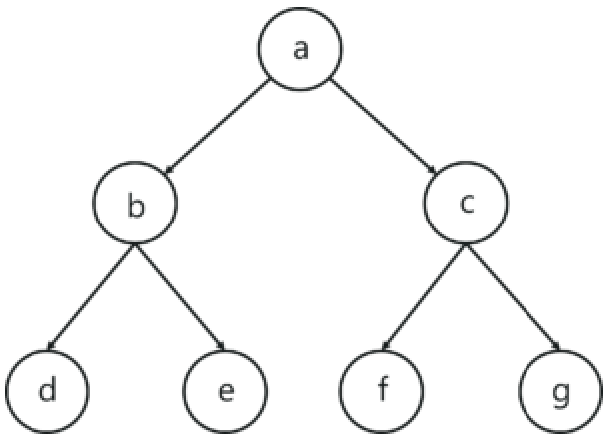
Now let's take a look at the steps involved in traversing a graph by using Breadth-First Search:

Step 1: Take an Empty Queue.

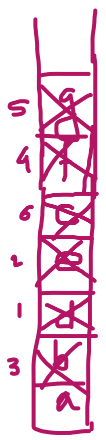
Step 2: Select a starting node (visiting a node) and insert it into the Queue.

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.

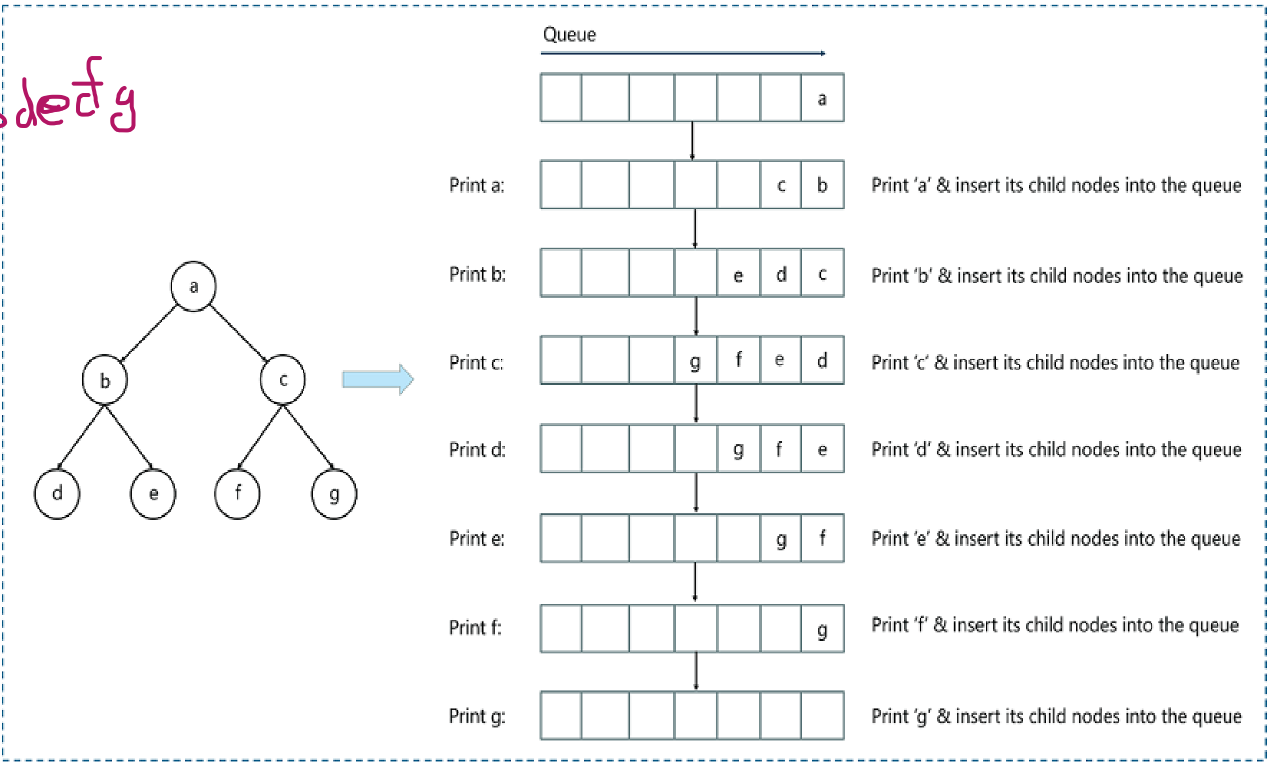
Step 4: Print the extracted node.



dFs



abdecfg



Print a: Print 'a' & insert its child nodes into the queue

Print b: Print 'b' & insert its child nodes into the queue

Print c: Print 'c' & insert its child nodes into the queue

Print d: Print 'd' & insert its child nodes into the queue

Print e: Print 'e' & insert its child nodes into the queue

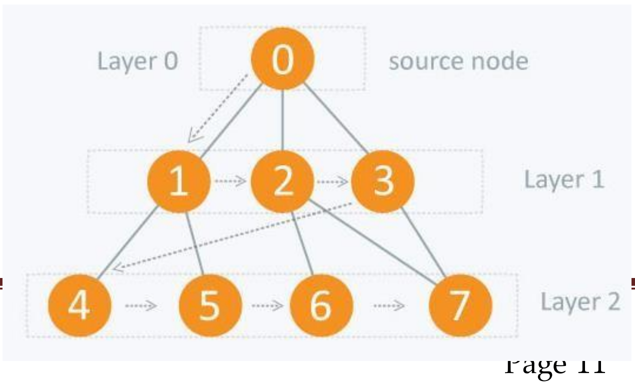
Print f: Print 'f' & insert its child nodes into the queue

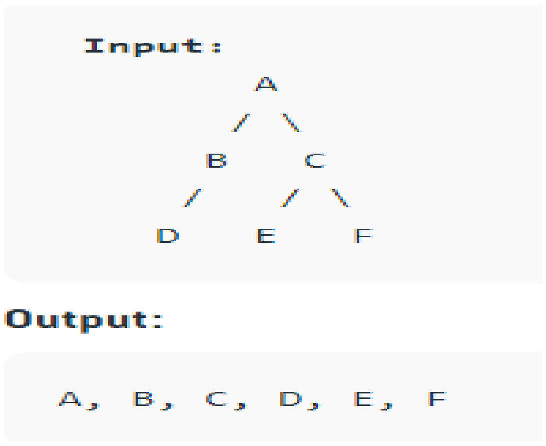
Print g: Print 'g' & insert its child nodes into the queue

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.





```
Input:
            A
           / \
          B   C
         /   / \
        D   E   F
```

Output:

```
A, B, C, D, E, F
```

The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

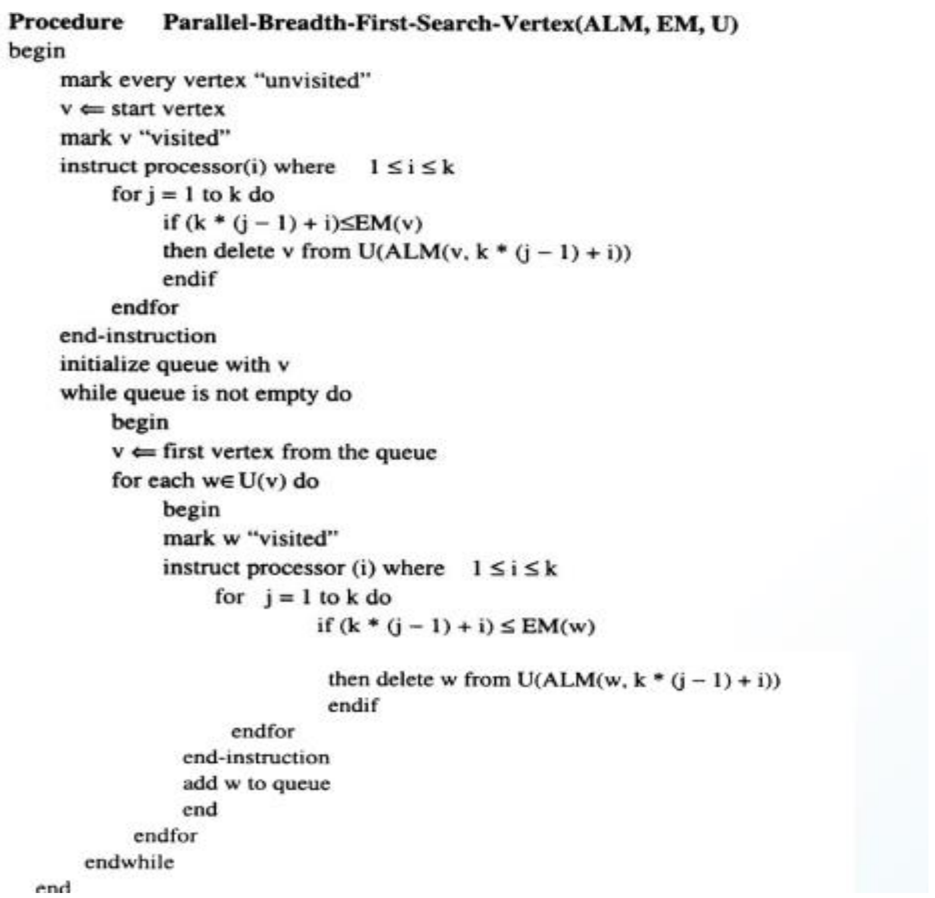
Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a Boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbors (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neigbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Pseudo code for BFS:



```
Procedure     Parallel-Breadth-First-Search-Vertex(ALM, EM, U)
begin
    mark every vertex "unvisited"
    v ⇐ start vertex
    mark v "visited"
    instruct processor(i) where    1 ≤ i ≤ k
        for j = 1 to k do
            if (k * (j – 1) + i)≤EM(v)
            then delete v from U(ALM(v, k * (j – 1) + i))
            endif
        endfor
    end-instruction
    initialize queue with v
    while queue is not empty do
        begin
        v ⇐ first vertex from the queue
        for each w∈ U(v) do
            begin
            mark w "visited"
            instruct processor (i) where    1 ≤ i ≤ k
                for   j = 1 to k do
                        if (k * (j – 1) + i) ≤ EM(w)
                        then delete w from U(ALM(w, k * (j – 1) + i))
                        endif
                endfor
            end-instruction
            add w to queue
            end
        endfor
    endwhile
end
```

Parallel Breadth First Search

1. To design and implement parallel breadth first search, you will need to divide the graph into smaller sub-graphs and assign each sub-graph to a different processor or thread.
2. Each processor or thread will then perform a breadth first search on its assigned sub-graph concurrently with the other processors or threads.
3. Two methods: Vertex by Vertex OR Level by Level.

2. Depth First Search

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 1 explores as far as possible along each branch before backtracking. This algorithm can be used to 2 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 3 backtracking is done to explore the next branch that has not been explored yet.

DFS can be implemented using either a 4 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.

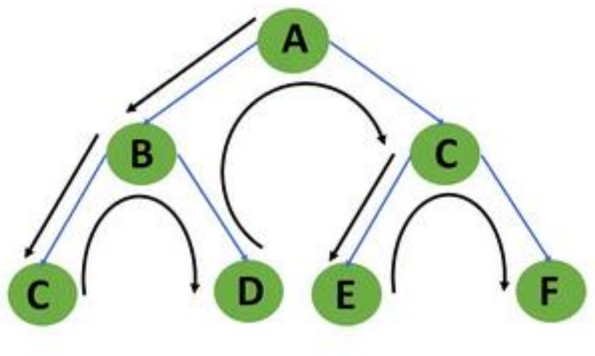
DFS can also be 5 used to detect cycles in a graph. If a cycle exists in a graph, the DFS algorithm will eventually reach a node that has already been visited, indicating that a cycle exists.

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

1. Visited

2. Not Visited

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



Example of DFS:

To implement DFS traversal, you need to take the following stages.
Step 1: Create a stack with the total number of vertices in the graph as the size.
Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to the stack.
Step 3 - Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.

Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.
Step 5 - If there are no new vertices to visit, go back and pop one from the stack using backtracking.
Step 6 - Continue using steps 3, 4, and 5 until the stack is empty.
Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

Parallel Depth First Search Steps:

1. In this implementation, the parallel_dfs function takes in a graph represented as an adjacency list, where each element in the list is a vector of neighboring vertices, and a starting vertex.
2. The dfs function uses a stack to keep track of the vertices to visit, and a boolean visited array to keep track of which vertices have been visited.
3. The #pragma omp parallel directive creates a parallel region and the #pragma omp single directive creates a single execution context within that region.
4. Inside the while loop, the #pragma omp task directive creates a new task for each unvisited neighbor of the current vertex.
5. This allows each task to be executed in parallel with other tasks. The firstprivate clause is used to ensure that each task has its own copy of the vertex variable.
6. This implementation is suitable for both tree and undirected graph, since both are represented as an adjacency list and the algorithm is using a stack to traverse the graph.
7. This is just one possible implementation, and there are many ways to improve it depending on the specific requirements of your application. For example, you can use omp atomic or omp critical to protect the shared resource stack.
8. The dfs function uses a stack to keep track of the vertices to visit, and a boolean visited array to keep track of which vertices have been visited. The #pragma omp parallel directive creates a parallel region and the #pragma omp single directive creates a single execution context within that region.

Conclusion: We have implemented Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP

Assignment Question

1.What is BFS and DFS?

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 a parallel Depth First Search (DFS) algorithm using OpenMP

6. What is the advantage of using parallel programming in DFS?

7.Write Down Commands used in OpenMP?

8. What is a race condition in parallel programming, and how can it be avoided in OpenMP?

Group A

Assignment No: 2

Title of the Assignment:

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

---

Objective of the Assignment:

To understand and implement Parallel Bubble Sort and Merge sort using OpenMP.

Outcome:

Students will be able to implement Parallel Bubble Sort and Merge sort using OpenMP.

Pre-requisites:

Student should know basic concepts of Bubble sort and Merge Sort

64-bit Open source Linux or its derivative

Programming Languages: C++/JAVA/PYTHON/R

Theory:

i) What is Sorting?

Sorting is a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, alphabetic order, etc.

Characteristics of Sorting are:

- Arrange elements of a list into certain order
- Make data become easier to access
- Speed up other operations such as searching and merging. Many sorting algorithms with different time and space complexities

ii) What is Parallel Sorting?

A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data. Therefore, parallel algorithms are used in sorting.

Design methodology:

- Based on an existing sequential sort algorithm
  - Try to utilize all resources available
  - Possible to turn a poor sequential algorithm into a reasonable parallel algorithm (bubble sort and parallel bubble sort)
- Completely new approach
  - New algorithm from scratch
  - Harder to develop
  - Sometimes yield better solution

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?**

A. Bubble Sort

The idea of bubble sort is 1 to compare two adjacent elements. If they are 2 not in the right order,switch them. 3 Do this comparing and switching (if necessary) until the end of the array is reached. 4 Repeat this process from the beginning of the array n times.

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.

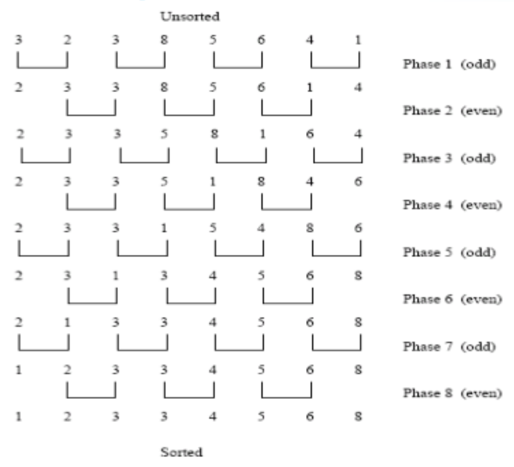
3. Move to the next pair of elements and repeat step 2.

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.

- One of the straight-forward sorting methods
  - Cycles through the list
  - Compares consecutive elements and swaps them if necessary
  - Stops when no more out of order pair
- Slow & inefficient
- Bubble sort is difficult to parallelize since the algorithm has no concurrency.
- The complexity of bubble sort is O(n^2)

Odd-Even Transposition



Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.

Bubble Sort Example

Here we want to sort an array containing [8, 5, 1]. The following figure shows how we can sortthis array using bubble sort. The elements in consideration are shown in **bold.**

8, 5, 1	Switch 8 and 5
5, **8, 1**	Switch 8 and 1
5, 1, 8	Reached end start again.
5, 1, 8	Switch 5 and 1
1, **5, 8**	No Switch for 5 and 8
1, 5, 8	Reached end start again.
1, 5, 8	No switch for 1, 5
1, **5, 8**	No switch for 5, 8
1, 5, 8	Reached end.

But do not start again since this is the nth iteration of same process

Parallel Bubble Sort

- Implemented as a pipeline.
- Let local_size = n / no_proc. We divide the array in no_proc parts, and each process executes the bubble sort on its part, including comparing the last element with the first one belonging to the next thread.
- Implement with the loop (instead of j<i)
  for (j=0; j<n-1; j++)

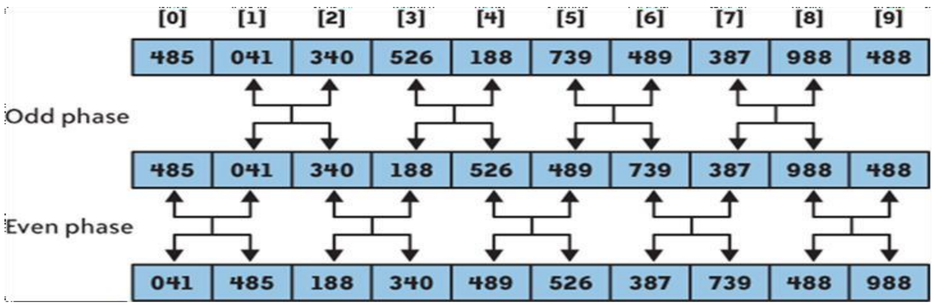
- For every iteration of i, each thread needs to wait until the previous thread has finished that iteration before starting.
- We'll coordinate using a barrier.

Algorithm for Parallel Bubble Sort

```
1.  For $k$ = 0 to $n$-2
2.  If $k$ is even then
3.      for $i$ = 0 to ($n$/2)-1 do in parallel
4.          If $A[2i] > A[2i+1]$ then
5.  Exchange $A[2i] \leftrightarrow A[2i+1]$
6.  Else
7.      for $i$ = 0 to ($n$/2)-2 do in parallel.
8.      If $A[2i+1] > A[2i+2]$ then
9.  Exchange $A[2i+1] \leftrightarrow A[2i+2]$
10. Next $k$
```

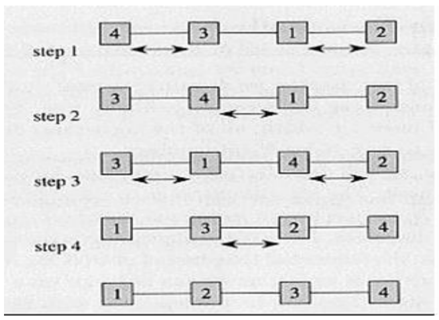
Parallel Bubble Sort Example 1

- Compare all pairs in the list in parallel
- Alternate between odd and even phases
- Shared flag, **sorted**, initialized to true at beginning of each iteration (2 phases), if any processor perform swap, **sorted** = false



Parallel Bubble Sort Example 2

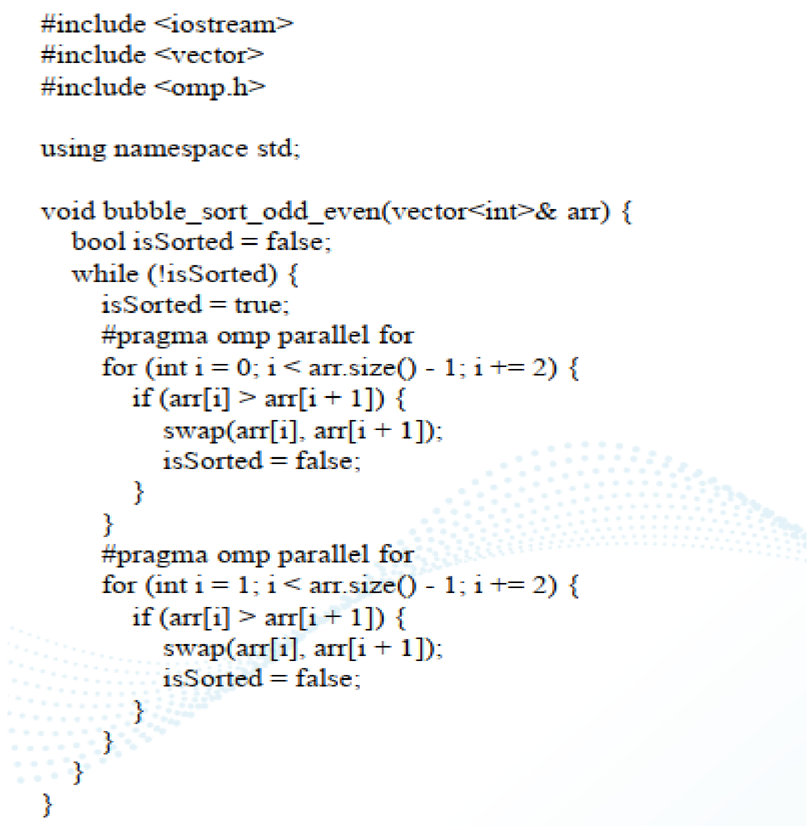
- How many steps does it take to sort the following sequence from least to greatest using the Parallel Bubble Sort? How does the sequence look like after 2 cycles?
- Ex: 4,3,1,2



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.

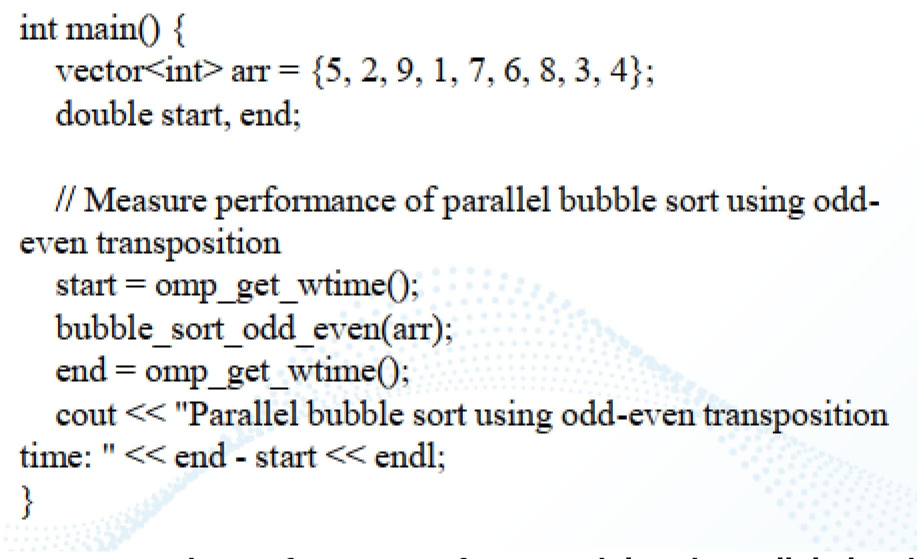
Parallel Bubble Sort



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

using namespace std;

void bubble_sort_odd_even(vector<int>& arr) {
    bool isSorted = false;
    while (!isSorted) {
        isSorted = true;
        #pragma omp parallel for
        for (int i = 0; i < arr.size() - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
        #pragma omp parallel for
        for (int i = 1; i < arr.size() - 1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
}
```



```
int main() {
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};
    double start, end;

    // Measure performance of parallel bubble sort using odd-
even transposition
    start = omp_get_wtime();
    bubble_sort_odd_even(arr);
    end = omp_get_wtime();
    cout << "Parallel bubble sort using odd-even transposition
time: " << end - start << endl;
}
```

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.

B. Merge Sort

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.

- Collects sorted list onto one processor
- Merges elements as they come together
- Simple tree structure
- Parallelism is limited when near the root

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.

Theory:

To sort *A*[*p* .. *r*]:

5. Divide Step

If a given array *A* has zero or one element, simply return; it is already sorted. Otherwise, split*A*[*p* .. *r*] into two subarrays*A*[*p* .. *q*] and *A*[*q* + 1 .. *r*], each containing about half of the elements of *A*[*p* .. *r*]. That is, *q* is the halfway point of *A*[*p* .. *r*].

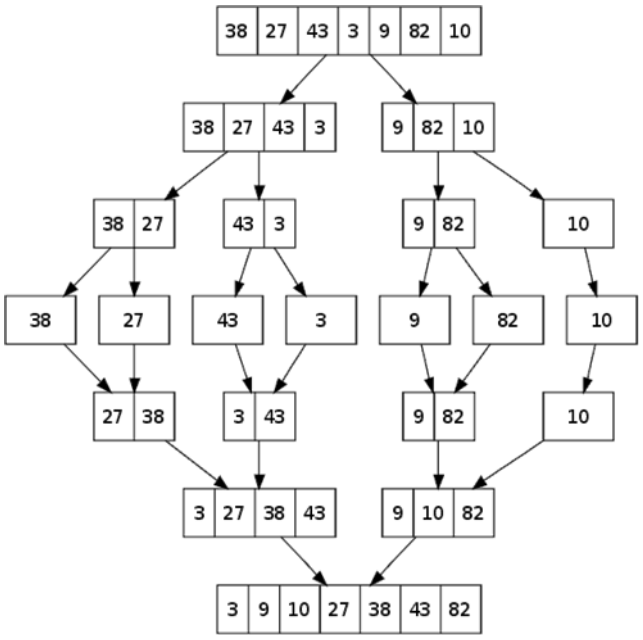
6. Conquer Step

Conquer by recursively sorting the two subarrays*A*[*p* .. *q*] and *A*[*q* + 1 .. *r*].

7. Combine Step

Combine the elements back in *A*[*p* .. *r*] by merging the two sorted subarrays*A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (*A*, *p*, *q*, *r*).

Example:



Parallel Merge Sort

- Parallelize processing of sub-problems
- Max parallelization achived with one processor per node (at each layer/height)

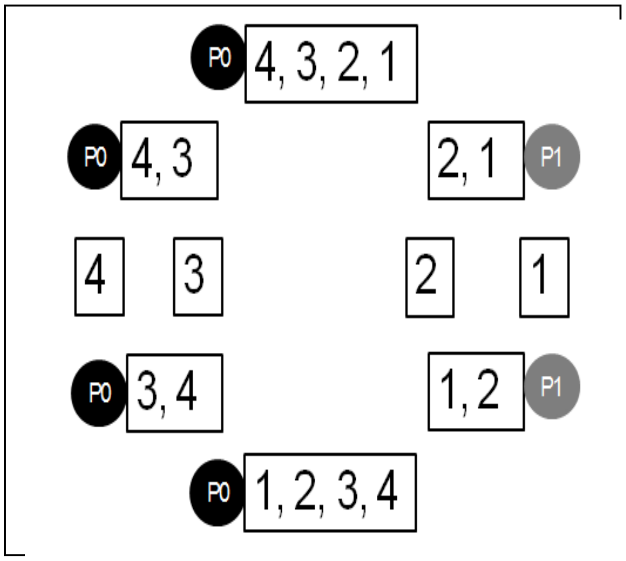
How Parallel Merge Sort Work

- 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.

Parallel Merge Sort Example

- Perform Merge Sort on the following list of elements. Given 2 processors, P0 & P1, which processor is reponsible for which comparison?
- 4,3,2,1



Algorithm for Parallel Merge Sort

```
1. Procedure parallelMergeSort
2. Begin
3. Create processors Pi where i = 1 to n
4. if i > 0 then recieve size and parent from the root
5. recieve the list, size and parent from the root
6. endif
7. midvalue= listsize/2
8. if both children is present in the tree then
9. send midvalue, first child
10. send listsize-mid,second child
11. send list, midvalue, first child
12. send list from midvalue, listsize-midvalue, second child
13. call mergelist(list,0,midvalue,list, midvalue+1,listsize,temp,0,listsize)
14. store temp in another array list2
15. else
16. call parallelMergeSort(list,0,listsize)
17. endif
18. if i >0 then
19. send list, listsize,parent
20. endif
```

21. end

INPUT:

1. Array of integer numbers.

OUTPUT:

1. Sorted array of numbers

ALGORITHM ANALYSIS

1. Time Complexity Of parallel Merge Sort and parallel Bubble sort in best case is(when all data is already in sorted form):O(n)
2. Time Complexity Of parallel Merge Sort and parallel Bubble sort in worst case is: O(n logn)
3. Time Complexity Of parallel Merge Sort and parallel Bubble sort in average case is: O(n logn)

APPLICATIONS

1. Representing Linear data structure & Sequential data organization : structure & files
2. For Sorting sequential data structure

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 performance 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 had implemented Bubble Sort and Merge Sort in parallel way using OpenMP also come to know how to measure performance of serial and parallel algorithm

FAQ

1. What is sorting?
2. What is parallel sort?
3. How to sort the element using Bubble Sort?
4. How to sort the element using Parallel Bubble Sort?
5. How to sort the element using Parallel Merge Sort?
6. How to sort the element using Merge Sort?
7. Difference between serial Mergesort and parallel Mergesort
8. Difference between serial Bubble sort and parallel Bubble sort.
9.

Group A

Assignment No: 3

Title of the Assignment:

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

Objective of the Assignment:

To study and implementation of directive based parallel programming model.

Outcome:

Students will be understand the implementation of sequential program augmented with compiler directives to specify parallelism.

Pre-requisites:

64-bit Open source Linux or its derivative

Programming Languages: C++/JAVA/PYTHON/R

Theory:

OpenMP:

OpenMP is a set of C/C++ pragmas (or FORTRAN equivalents) which provide the programmer a high-level front-end interface which get translated as calls to threads (or other similar entities). The key phrase here is "higher-level"; the goal is to better enable the programmer to "think parallel," alleviating him/her of the burden and distraction of dealing with setting up and coordinating threads. For example, the OpenMP directive.

OpenMP Core Syntax:

Most of the constructs in OpenMP are compiler directives:

#pragma omp construct [clause [clause]...]

Example

#pragma omp parallel num_threads(4)

Function prototypes and types in the file:

#include

<omp.h>

Most OpenMP constructs apply to a “structured block”

Structured block:

a block of one or more statements surrounded by “{ }”, with one point of entry at the top and one point of exit at the bottom.

Following is the sample code which illustrates max operator usage in OpenMP :

```
#include <stdio.h>
#include  <omp.h>
int main()
{
   double arr[10];
   omp_set_num_threads(4);
   double max_val=0.0;
   int i;
   for( i=0; i<10; i++)
      arr[i] = 2.0 + i;
   #pragma omp parallel for reduction(max : max_val)
   for( i=0;i<10; i++)
   {
      printf("thread id = %d and i = %d", omp_get_thread_num(), i);
      if(arr[i] > max_val)
      {
         max_val = arr[i];
      }
   }
   printf("\nmax_val = %f", max_val);
}
```

Following is the sample code which illustrates min operator usage in OpenMP :

```
#include <stdio.h>
#include  <omp.h>
int main()
{
   double arr[10];
   omp_set_num_threads(4);
   double min_val=0.0;
```

```
            int i;
            for( i=0; i<10; i++)
                arr[i] = 2.0 + i;
            #pragma omp parallel for reduction(min : min_val)
    for( i=0;i<10; i++)
    {
        printf("thread id = %d and i = %d", omp_get_thread_num(), i);
        if(arr[i] < min_val)
        {
            min_val = arr[i];
        }
    }
    printf("\nmin_val = %f", min_val);
}
```

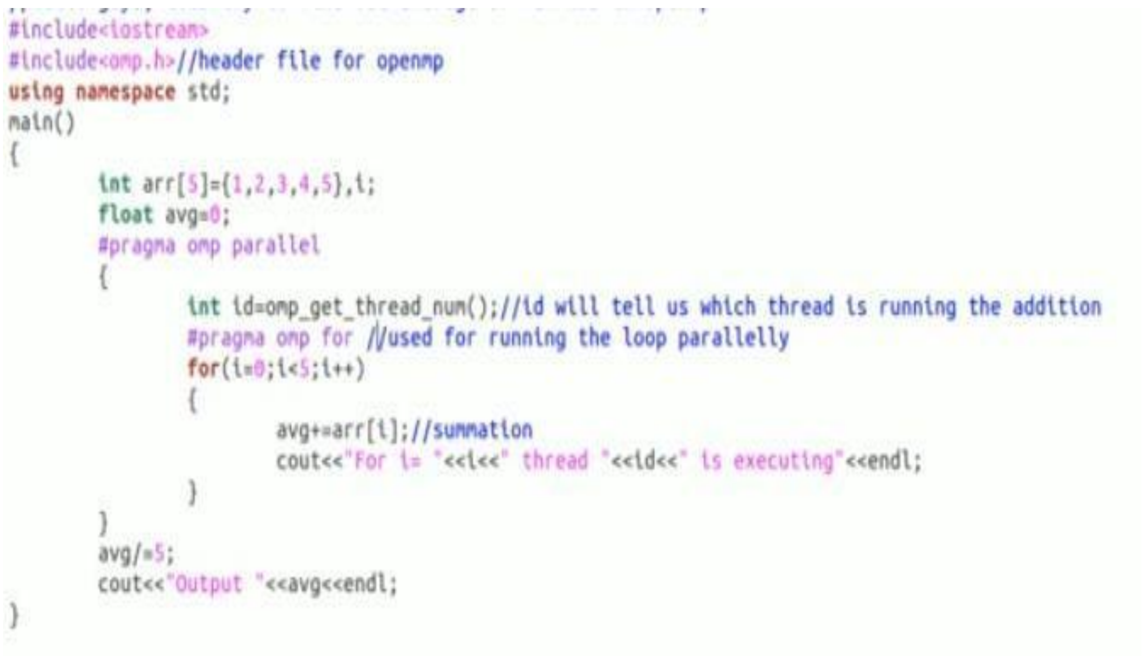
Following is the sample code which illustrates sum operation usage in OpenMP :

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

int main (int argc, char *argv[])
{
int  i, n;
float a[100], b[100], sum;
/* Some initializations */
n = 100;
for (i=0; i < n; i++)
a[i] = b[i] = i * 1.0;
sum = 0.0;


#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
printf("  Sum = %f\n",sum);
}
```

Following is the sample code which illustrates sum operation usage in OpenMP :



```
#include<iostream>
#include<omp.h>//header file for openmp
using namespace std;
main()
{
        int arr[5]={1,2,3,4,5},i;
        float avg=0;
        #pragma omp parallel
        {
                int id=omp_get_thread_num();//id will tell us which thread is running the addition
                #pragma omp for //used for running the loop parallelly
                for(i=0;i<5;i++)
                {
                        avg+=arr[i];//summation
                        cout<<"For i= "<<i<<" thread "<<id<<" is executing"<<endl;
                }
        }
        avg/=5;
        cout<<"Output "<<avg<<endl;
}
```

Implementation of Min, Max, Sum and Average operations using Parallel Reduction

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

using namespace std;
void min_reduction(vector<int>& arr) {
        int min_value = INT_MAX;
        #pragma omp parallel for reduction(min: min_value)
        for (int i = 0; i < arr.size(); i++) {
                if (arr[i] < min_value) {
                        min_value = arr[i];
                }
        }
        cout << "Minimum value: " << min_value << endl;
}


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

 void sum_reduction(vector<int>& arr) {
        int sum = 0;
        #pragma omp parallel for reduction(+: sum)
        for (int i = 0; i < arr.size(); i++) {
                sum += arr[i];
        }
        cout << "Sum: " << sum << endl;
}


void average_reduction(vector<int>& arr) {
        int sum = 0;
        #pragma omp parallel for reduction(+: sum)
        for (int i = 0; i < arr.size(); i++) {
                sum += arr[i];
        }
        cout << "Average: " << (double)sum / arr.size() << endl;
}

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

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

- The min_reduction function finds the minimum value in the input array using the #pragma omp parallel for reduction(min: min_value) directive, which creates a parallel region and divides the loop iterations among the available threads. Each thread performs the comparison operation in parallel and updates the min_value variable if a smaller value is found.
- Similarly, the max_reduction function finds the maximum value in the array, sum_reduction function finds the sum of the elements of array and average_reduction function finds the average of the elements of array by dividing the sum by the size of the array.
- The reduction clause is used to combine the results of multiple threads into a single value, which is then returned by the function. The min and max operators are used for the min_reduction and max_reduction functions, respectively, and the + operator is used for the sum_reduction and average_reduction functions. In the main function, it creates a vector and calls the functions min_reduction, max_reduction, sum_reduction, and average_reduction to compute the values of min, max, sum and average respectively.

Conclusion: We have implemented parallel reduction using Min, Max, Sum andAverage Operations.

Group A

Assignment No: 4

Title of the Assignment:

Write a CUDA Program for :

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

Objective of the Assignment:

Student should be able to learn parallel programming, CUDA architecture and CUDA processing flow

----

Outcome:

Students will be understanding and Implement nxn matrix parallel addition, multiplication using CUDA, use shared memory.

Pre-requisites:

64-bit Open source Linux or its derivative

Programming Languages: C++/JAVA/PYTHON/R

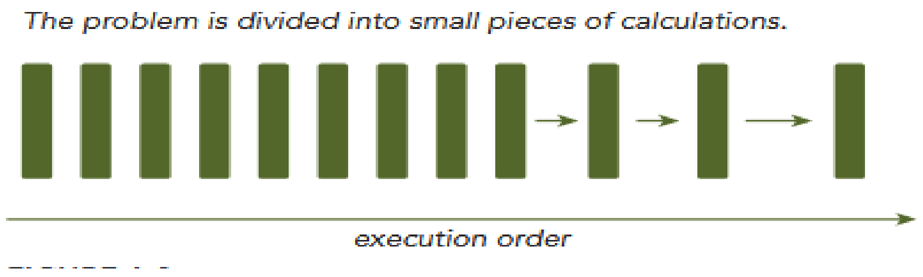
Concept of matrix addition, multiplication.
Basics of CUDA programming

Theory:

Sequential Programming:

When solving a problem with a computer program, it is natural to divide the problem into a discrete series of calculations; each calculation performs a specified task, as shown in following Figure. Such a pro-gram is called a sequential program.

.

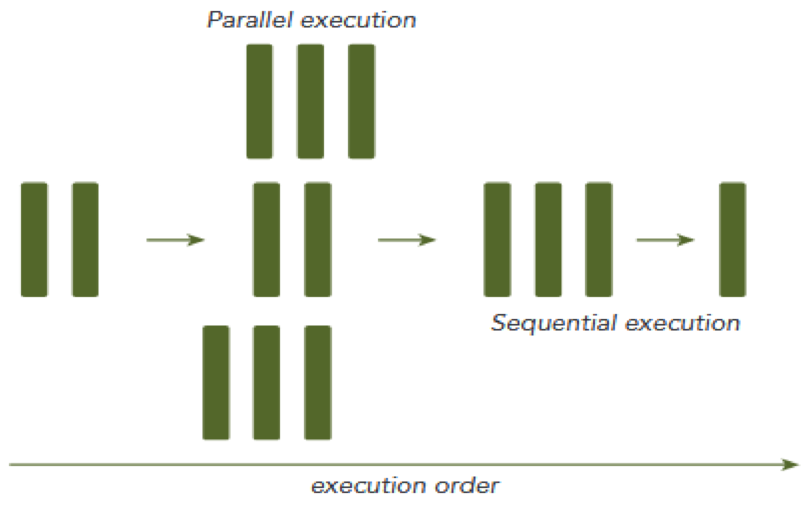


Parallel Programming:

There are two fundamental types of parallelism in applications:

➢Task parallelism
➢Data parallelism

Task parallelism arises when there are many tasks or functions that can be operated independently and largely in parallel. Task parallelism focuses on distributing functions across multiple cores.
Data parallelism arises when there are many data items that can be operated on at the same time.
Data parallelism focuses on distributing the data across multiple cores.



CUDA :

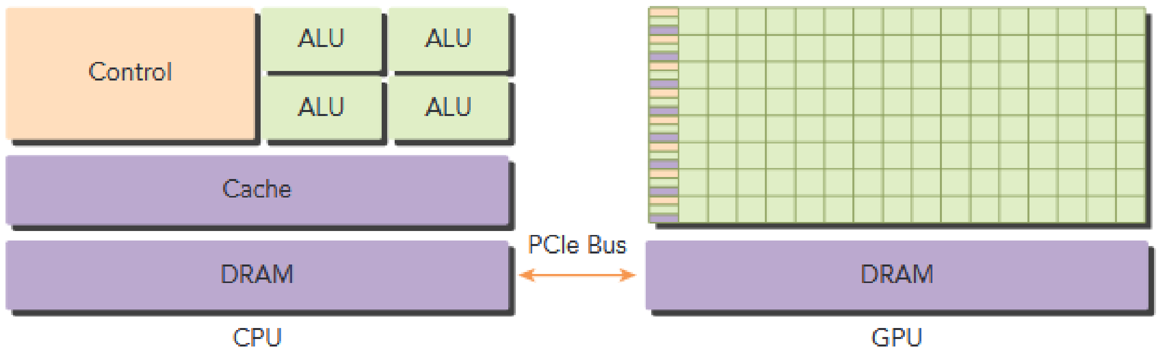
CUDA programming is especially well-suited to address problems that can be expressed as data parallel computations. Any applications that process large data sets can use a data-parallel model to speed up the computations. Data-parallel processing maps data elements to parallelthreads.
The first step in designing a data parallel program is to partition data across threads, with each thread working on a portion of the data.

CUDA Architecture:

A heterogeneous application consists of two parts:

- ➢Host code
- ➢Device code

Host code runs on CPUs and device code runs on GPUs. An application executing on a heterogeneous platform is typically initialized by the CPU. The CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks on the device. With computational intensive applications, program sections often exhibit a rich amount of data parallelism. GPUs are used to accelerate the execution of this portion of data parallelism. When a hardware component that is physically separate from the CPU is used to accelerate computationally intensive sections of an application, it is referred to as a hardware accelerator. GPUs are arguably the most common example of a hardware accelerator. GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus, as shown in Figure.



NVIDIA’s CUDA nvcc compiler separates the device code from the host code during the compilation process. The device code is written using CUDA C extended with keywords for labeling data-parallel functions, called kernels . The device code is further compiled by

Nvcc . During the link stage, CUDA runtime libraries are added for kernel procedure calls and explicit GPU device manipulation. Further kernel function, named helloFromGPU, to print the string of “Hello World from GPU!” as follows:

```
__global__void helloFromGPU(void)
{
printf(“Hello World from GPU!\n”);
}
```

The qualifier __global_tells the compiler that the function will be called from the CPU and exe-cuted on the GPU. Launch the kernel function with the following code:

helloFromGPU <<<1,10>>>();

Triple angle brackets mark a call from the host thread to the code on the device side. A kernel is executed by an array of threads and all threads run the same code. The parameters within the triple angle brackets are the execution configuration, which specifies how many threads will execute the kernel. In this example, you will run 10 GPU threads.

A typical processing flow of a CUDA program follows this pattern:

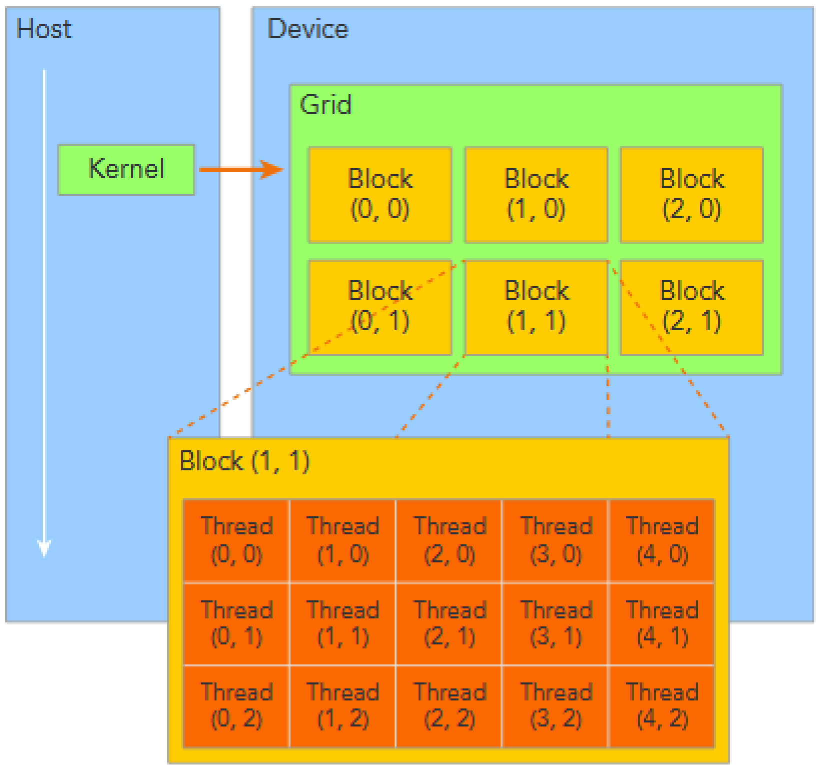
1. Copy data from CPU memory to GPU memory.
2. Invoke kernels to operate on the data stored in GPU memory.
3. Copy data back from GPU memory to CPU memory

Table lists the standard C functions and their corresponding CUDA C functions for memory operations. Host and Device Memory Functions are follows.

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Organizing Threads:

When a kernel function is launched from the host side, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function. The two-level thread hierarchy decomposed into blocks of threads and gridsof blocks, as shown in following figure:



All threads spawned by a single kernel launch are collectively called a grid . All threads in a grid
share the same global memory space. A grid is made up of many thread blocks. A thread block is a group of threads that can cooperate with each other using:

➢Block-local synchronization

➢Block-local shared memory

Threads from different blocks cannot cooperate.
Threads rely on the following two unique coordinates to distinguish themselves from each other:

➢blockIdx (block index within a grid)

➢threadIdx (thread index within a block)

These variables appear as built-in, pre-initialized variables that can be accessed within kernel functions. When a kernel function is executed, the coordinate variables blockIdx and

threadIdx are assigned to each thread by the CUDA runtime. Based on the coordinates, you can assign portions of data to different threads. It is a structure containing three unsigned integers, and the 1st, 2nd, and 3rd components are accessible through the fields x, y, and z respectively.

blockIdx.x
blockIdx.y
blockIdx.z

threadIdx.x
threadIdx.y
threadIdx.z
CUDA organizes grids and blocks in three dimensions. The dimensions of a grid and a block are specified by the following two built-in variables:

➢blockDim (block dimension, measured in threads)

➢gridDim (grid dimension, measured in blocks)

These variables are of type dim3, that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1.Each component in a variable of type dim3 is accessible through its x,y,, and z fields, respectively, as shown in the following example:

blockDim.x
blockDim.y
blockDim.

Matrix Multiplication using CUDA C

A straightforward matrix multiplication example that illustrates the basic features of memoryand thread management in CUDA programs

- Leave shared memory usage until later
- Local, register usage
- Thread ID usage

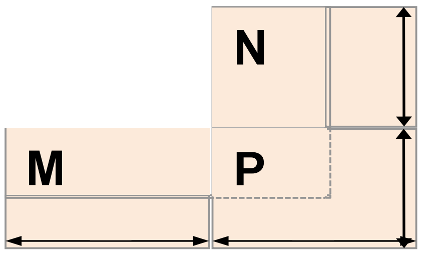
Memory data transfer API between host and device

- *P = M * N of size WIDTH x WIDTH*

Without tiling:

- One thread handles one element of P

M and N are loaded WIDTH times from global memory



Matrix Multiplication steps

1. **Matrix Data Transfers**
2. **Simple Host Code in C**
3. **Host-side Main Program Code**
4. **Device-side Kernel Function**
5. **Some Loose Ends**

Step 1: Matrix Data Transfers

```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  =  WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);
// Copy M from the host to the device
```

```
        cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
        // Read M from the device to the host into P
        cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);
        ...
        // Free device memory
        cudaFree(Md.elements);
```

Step 2: Simple Host Code in C

```
        // Matrix multiplication on the (CPU) host in double precision
        // for simplicity, we will assume that all dimensions are equal

    void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
   {
       for (int i = 0; i < M.height; ++i)
           for (int j = 0; j < N.width; ++j) {
               double sum = 0;
               for (int k = 0; k < M.width; ++k) {
                   double a = M.elements[i * M.width + k];
                   double b = N.elements[k * N.width + j];
                   sum += a * b;
               }
               P.elements[i * N.width + j] = sum;
           }
```

Multiply Using One Thread Block

- One Block of threads compute matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)

- Size of matrix limited by the number of threads allowed in a thread block

Step 3: Host-side Main Program Code

```
int main(void) {
// Allocate and initialize the matrices
    Matrix M = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix N = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);
// M * N on the device
    MatrixMulOnDevice(M, N, P);
// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```

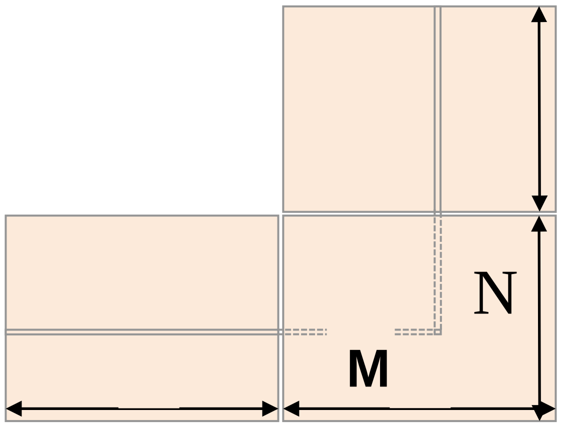
Host-side code

```
  // Matrix multiplication on the device
  void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
  {
      // Load M and N to the device
      Matrix Md = AllocateDeviceMatrix(M);
      CopyToDeviceMatrix(Md, M);
      Matrix Nd = AllocateDeviceMatrix(N);
      CopyToDeviceMatrix(Nd, N);
      // Allocate P on the device
      Matrix Pd = AllocateDeviceMatrix(P);
      CopyToDeviceMatrix(Pd, P); // Clear memory
      // Setup the execution configuration
      dim3 dimBlock(WIDTH, WIDTH);
      dim3 dimGrid(1, 1);
      // Launch the device computation threads!
      MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);
      // Read P from the device
      CopyFromDeviceMatrix(P, Pd);
      // Free device matrices

    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Step 4: Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global___void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
for (int k = 0; k < M.width; ++k)
    {
         float Melement = M.elements[ty * M.pitch + k];
         float Nelement = Nd.elements[k * N.pitch + tx];
         Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P.pitch + tx] = Pvalue;
}
```



Step 5: Some Loose Ends

- Free allocated CUDA memory

Vector Addition

```
                    #include <iostream>
                    #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;
}
```

Facilities:

Latest version of 64 Bit Operating Systems, CUDA enabled NVIDIA Graphics card

Input:

Two matrices

Output:

Multiplication of two matrix

Software Engg.:

Mathematical Model:

Conclusion:

We learned parallel programming with the help of CUDA architecture.

Questions:

1. What is CUDA?
2. Explain Processing flow of CUDA programming.
3. Explain advantages and limitations of CUDA.
4. Make the comparison between GPU and CPU.
5. Explain various alternatives to CUDA.
6. Explain CUDA hardware architecture in detail.

GROUP B: ASSIGNMENTS