Practical No - 01

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

Objectives: Students should be able to understand and implement searching algorithms like BFS and DFS.

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

Pre-requisites: 64 bit Open source Linux
Programming Languages: C++/ JAVA / Python/ R.

Theory: Graph Traversals →

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each

vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

Breadth First Search (BFS) →

It is a graph traversal algorithm used to explore all nodes of 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.

BFS is a traversing algorithm where you should start traversing from a selected node and traverse graph layerwise thus exploring the neighbor nodes, you must then move towards next-level neighbour nodes.

Example of BFS :

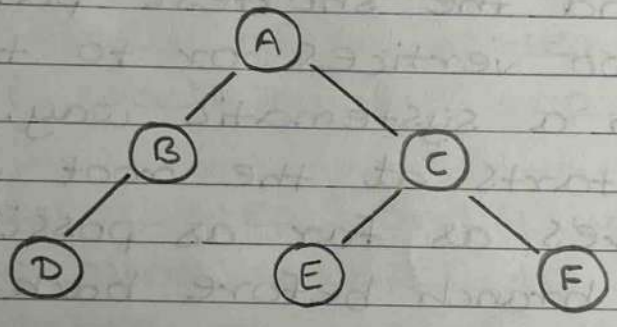
Now let's take a look at steps involved in traversing a graph by using Breadth-First Search.

Step 1: Take an empty queue.

Step 2: Select a starting node and insert it into the queue.

Step 3: Provided that the queue is not empty, extract the node from queue and insert its child nodes into queue.

Step 4: Print the extracted node.



Output : A, B, C, D, E, F.

Parallel Breadth First Search :

(i) 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.

(ii) Each processor or thread will then perform a breadth first search on its assigned sub-graph concurrently with

(iii) Two methods: vertex by vertex OR
Level by Level.

Depth First Search (DFS) →

DFS 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.

A standard DFS implementation puts each vertex of graph into one of two categories:
(i) Visited
(ii) Not visited.

Example of DFS:

Step1: Create a stack with the total number of vertices in the graph as the size.

Step2: Choose any vertex as traversal's beginning point. Push a visit to that vertex and add it to the stack.

Step3: Push any non-visited adjacent vertices of a vertex at the top of stack to the top of stack.

Step4: Repeat step 3 and 4 until there are no more vertices to visit from the vertex at top of stack.

Step5: If there are no new vertices to visit, go back and pop one from stack using backtracking.

Step6: Continue using step 3, 4, 5 until the stack is empty.

Step7: When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

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

Assignment Questions →

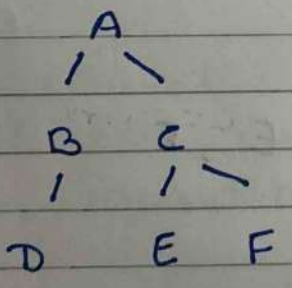
1. What is BFS and DFS ?

→ Breadth First Search -

It is a verten based technique for finding the shortest path in the graph. It uses a queue data structure that follows first in first out. In BFS, one verten is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

Example:

Input :



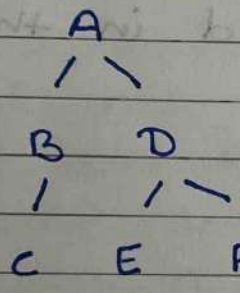
Output : A , B , C , D , E , F.

Depth First Search -

DFS is an edge - based technique. It uses the stack data structure and performs two stages, first visited vertices are pushed into stack, and second if there are no vertices then visited vertices are popped.

Example:

Input:



Output : A, B, C, D, E, F

2. What is Open MP ? What is its significance in parallel programming ?

→ - Open mp is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared memory environments.

- Open MP identifies parallel regions as blocks of code that may run in parallel.

- Application developers insert compiler directives into their code at parallel regions, and these directives instruct the Open MP run time library to execute the region in parallel.

- Open MP provides several additional directives for running code regions in parallel, including parallelizing loops.

- In addition to providing directives for parallelization, Open MP allows developers to choose among several levels of parallelism.

- Eg. they can set number of threads manually.

3. Write down applications of Parallel BFS.

→ - In peer-to-peer network like bit-torrent, BFS is used to find all neighbour nodes.

- Search engine crawlers are used BFS to build index. Starting from source page, it finds all links in it to get new pages.

- Using GPS navigation system BFS is used to find neighboring places.

- In networking, when we want to broadcast some packets, we use the BFS algorithm.

- BFS is used in Fork-Fulkerson algorithm to find maximum flow in a network.

4. How can BFS be parallelized using OpenMP ? Describe the parallel BFS algorithm using OpenMP.

→ - 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 different processor or thread.

- Each processor or thread will then perform a breadth first search or its assigned sub-graph concurrently with the other processor or threds.

- Two methods : Vertex by vertex
Level by level.

5. Write a parallel DFS algorithm using OpenMP.

→ - 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 neighbor vertices, and a starting vertex.

- 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 content within that region.

- Inside the while loop, the #pragma omp task directive creates a new task for each unvisited neighbor of current vertex.

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

→ Efficiency -

A computer that uses parallel programming can make better use of its resource to process and solve problems. When computers use all their resources to solve a problem or process information, they are more efficient at performing tasks.

Cost effectiveness -

Additionally, the hardware architecture that allows for parallel programming is more cost-effective than systems that only allow for serial processing. This means that they produce more results in less time than serial programs and hold more financial value over time.

Speed -

Another benefit of parallel computing is its ability to solve complex problems. Parallel programs can divide complex problems down into smaller tasks and process these individual tasks simultaneously.

7. Write down commands used in openmp?

→ i. omp_set_num_threads

sets the number of threads in upcoming parallel regions, unless overridden by a num_threads clause.

ii. omp_get_num_threads

Returns the number of threads in parallel regions.

iii. ~~Open~~ Omp_get_max_threads

Returns an integer that is equal to or greater than the number of threads that would be available if a parallel region without num_threads were defined at that point in the code.

iv. omp_in_parallel

Returns nonzero if called from within a parallel region.

v. omp_set_dynamic

Indicates that the number of threads available in upcoming parallel regions can be adjusted by the run time.

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

→ A race condition occurs in parallel program execution when two or more threads access a common resource, e.g. a variable in shared memory, and the order of the accesses depends on the timing i.e., the progress of individual threads.

Practical No - 02

Aim: Write a program to implement Parallel Bubble Sort and merge sort using Open MP. Use existing algorithms and measure performance of sequential and parallel algorithms.

Objective: To understand and implement Parallel Bubble Sort and merge sort using Open MP.

Outcome: Students will be able to implement Parallel Bubble sort and merge sort using Open MP.

Pre-requisites: -Students should know basic concepts of Bubble sort and merge Sort.
- 64 bit Open source Linux
- Programming Languages : C++ / JAVA / Python / R.

Theory: 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 list into certain order.
- make data become easier to access
- Speed up other operations such as searching and merging.

Bubble Sorting →

The idea of bubble sort is to compare two adjacent elements. If they are not in the right order, switch them. Do this comparing and switching until the end of the array is reached. 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.

Algorithm of Bubble Sort :

Step 1: Start at the beginning of array.

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

Step 3: move the next pair of elements and repeat step 2.

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

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

Example of Bubble Sort →

Here we want to sort an array containing [8, 5, 1]. The following figure shows how we can sort this 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 ~~out~~ end.

Parallel Bubble sort algorithm →

(i) For k = 0 to n-2
(ii) If k is even then
(iii) For i = 0 to (n/2)-1 do in parallel
(iv) If A[2i] > A[2i+1] then
(v) Exchange A[2i] ↔ A[2i+1]
(vi) Else
(vii) For i = 0 to (n/2)-2 do in parallel
(viii) If A[2i+1] > A[2i+1] ↔ A[2i+2]
(ix) Exchange A[2i+1] ↔ A[2i+2]
(x) Next k.

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.

Algorithm:

(i) Divide the input array into two halves.
(ii) Recursively sort the left half of array.
(iii) Recursively sort the right half of array.
(iv) merge two sorted halves into a single sorted output array.
(v) Divide step
(vi) Conquer step
(vii) Combine step

Parallel merge sort →

- Parallelize processing of sub-problems

- Max parallelization achieved with one processor per node.

Parallel merge sort algorithm:

(i) Procedure parallel merge sort
(ii) Begin
(iii) Create processors Pi where i = 1 to n.

(iv) if i > 0 then recieve size and parent from the root.
(v) recieve list, size and parent from root.
(vi) end if
(vii) midvalue = list size / 2
(viii) if both children is present in tree then
(ix) send midvalue, first child
(x) send list size - mid, second child
(xi) send list, midvalue, first child
(xii) send list from midvalue, list size - midvalue, second child
(xiii) call mergelist (list0, midvalue, list, midvalue+1, listsize, temp0, listsize)
(xiv) store temp in another array list2
(xv) else
(xvi) call parallel merge sort (list, 0, listsize)
(xvii) endif
(xviii) if i > 0 then
(xix) send list, list size, parent
(xx) end if.

Conclusion: In this way we have implemented Bubble sort and merge sort in parallel way using Open MP; also came to know how to measure performance of serial and parallel algorithm.

Assignment Questions →

1. What is sorting ?

→ Sorting is a process of ordering or placing a list of elements from a collection in some kind of order.

It is nothing but storage of data in sorted order.

Sorting can be done in ascending and descending order.

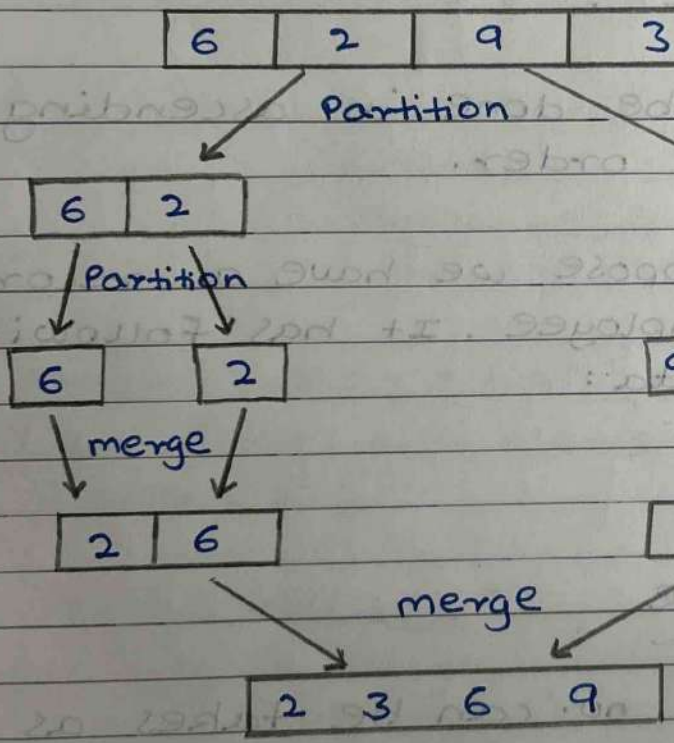
For example: suppose we have a record of employee. It has following data:

Employee No.
Employee Name
Employee Salary
Department Name

Here, employee no. can be takes as key for sorting the records in ascending or descending order. Now, we have to search a Employee with employee no. 116, so we don't require to search that complete records, simply we can search between the Employee with employee no. 100 to 120.

2. What is parallel sort ?

→ merge sort first divides the unsorted list into smallest possible sub-lists, compares it with the adjacent list, and merges it in a sorted order. It implements parallelism very nicely by following the divide and conquer algorithm.



3. How to sort the elements using Bubble Sort?

→ - Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

- Algorithm:

Input : arr[] = {6, 3, 0, 5}

First pass: Bubble sort starts with very first two elements, comparing them to check which one is greater.

(6305) -> (3605), Here, algorithm compares the first two elements, and swap since 6 > 3.

(3605) -> (3065), Swap since 6 > 0

(3065) -> (3056), Swap since 6 > 5

Second Pass: Now, during second iteration it should look like this:

(3056) -> (0356), Swap since 3 > 0

(0356) -> (0356), No change as 5 > 3

Third Pass: Now, the array is already sorted, but our algorithm does not know if it is completed.
The algorithm need one whole pass without any swap to know it is sorted

(0 3 5 6) -> (0 3 5 6), No change as 3 > 0

Array is now sorted and no more pass will happen.

4. How to sort the element using Parallel Bubble Sort?

→ - Parallel bubble sort is a parallel implementation of the classic Bubble sort algorithm.

- The concept of parallelism involves executing a set of instructions / code simultaneously instead of line by line sequentially.

- we divide sorting of the unsorted into two phases - odd and even.

- we compare all pairs of elements in the list / array side by side.

6	5	3	4

step 1 (odd)

5	6	3	4

step 2 (even)

5	3	6	4

step 3 (odd)

3	5	4	6

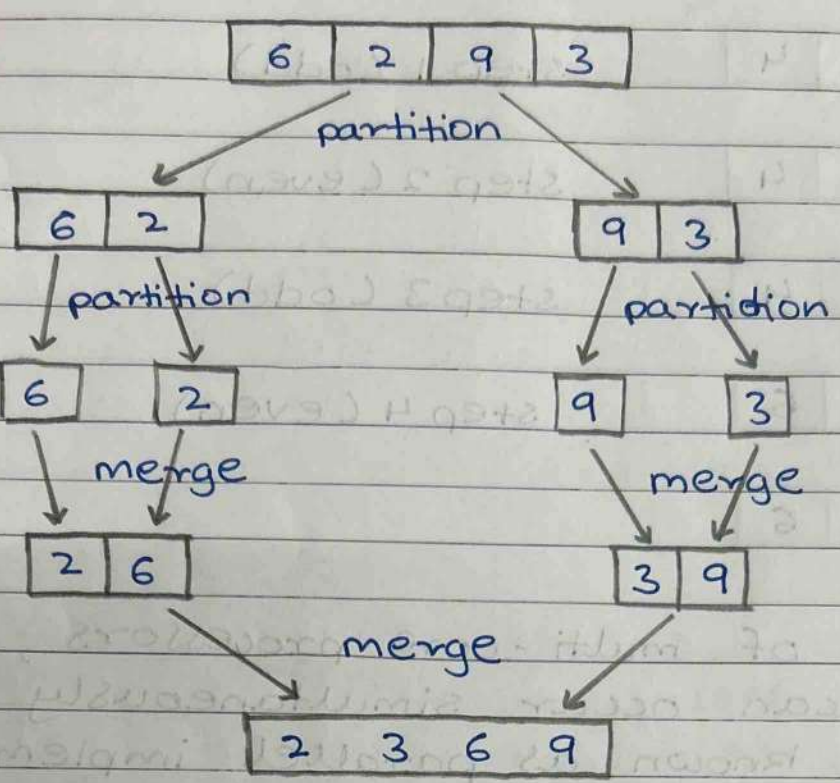
step 4 (even)

3	4	5	6

In case of multi-core processors, both phases can occur simultaneously, which is known as parallel implementation.

5. How to start the element using Parallel merge sort?

→ merge sort first divides the unsorted list into smallest possible sub-lists, compares it when the adjacent list, and merges it in a sorted order. It implements parallelism very nicely by following the divide and conquer algorithm.



6. How to sort the elements using merge sort?

→ - Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarray back together to form the final sorted array.

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

- merge sort algorithm :

step1 : start

step2 : declare array and left , right , mid variable

step3 : perform merge function.

```
if left > right
   return
mid = ( left + right ) / 2
mergesort (array , left , mid)
mergesort (array , mid +1 , right)
merge (array , left , mid , right)
```

step4 : stop.

7. Difference between serial merge sort and parallel merge sort.

→ merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays , sorting each subarray , and then merging the sorted subarrays back together to form the final sorted array.

Parallel merge sort first divides the unsorted list into smallest possible sub-lists , compares it with adjacent

list, and merges it in a sorted order. It implements parallelism using divide and conquer algorithm.

Time complexity for merge sort is $T(n) = 2T(n/2) + O(n)$.

Time complexity for parallel merge sort is $O(\log(n)2)$.

8. Difference between serial bubble sort and parallel bubble sort.

→

Serial bubble sort	Parallel bubble sort.
- Bubble sort works by repeatedly swapping the adjacent elements if they are in wrong order.	Concep of parallelism involves executing a set of instructions / codes simultaneously instead of line by line.
- Time complexity is $O(n2)$.	Time complexity is $O(n\log(n))$.

Practical No - 03

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

Objective: To study and implementation of directive based parallel programming tool.

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: Open MP →

Open MP is a set of C/C++ programs which provide the programmer a high-level front-end interface which get translated as calls to threads. 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 co-ordinating threads. For example, the Open MP directive.

Open MP Core Syntax →

most of the constructs in Open MP 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 Open MP 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.

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 threds.

- 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 array.

- The reduction clause is used to combine the results of multiple threads into a single value, which is then retured 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.

Conclusion: we have implemented parallel reduction using min, max, Sum and Average operations.

Practical No - 04

Aim: Write a CUDA Program for:
(i) Addition of two large vectors
(ii) matrix multiplication using CUDA C.

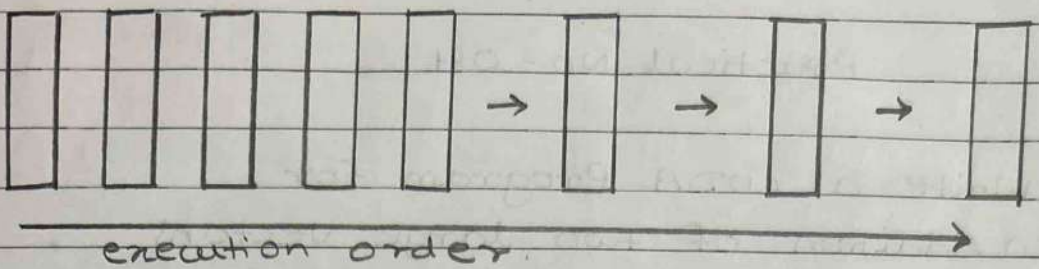
Objectives: Students should be able to learn parallel programming CUDA architecture and CUDA processing flow.

Outcome: Students will be understanding and implement $n \times n$ matrix parallel addition, multiplication using CUDA, use shared memory.

Pre requisites: 64 bit Open source linux or its derivative
Programming Language: C++/ JAVA/ Python / R.
Concept of matrix addition, multiplication. Basics of CUDA programming.

Theory: Sequential Programming →

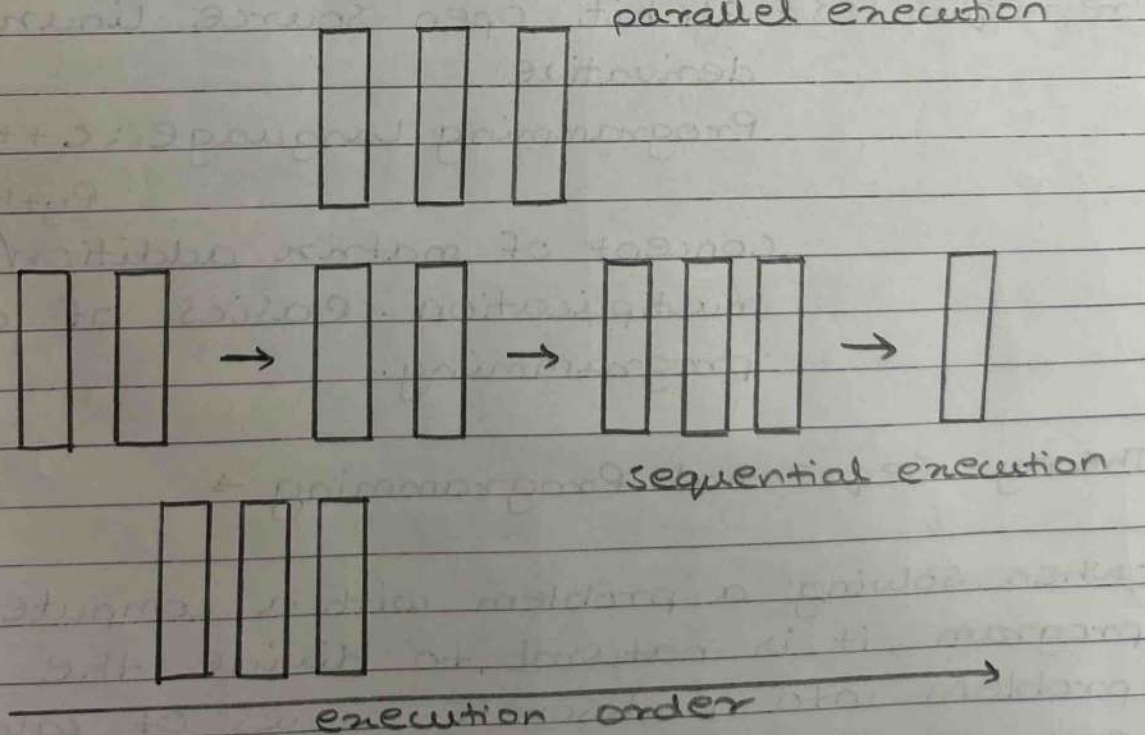
- When solving a problem with a computer program, it is natural to divide the problem into a discrete series of calculations, each calculation performs a specific task, as shown in following figure.



Parallel Programming →

- Task parallelism arises when there are many tasks or functions that can be operated independently and largely in parallel.

- Data parallelism arises when there are many data items that can be operated on at the same time.



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 - data parallel model to speed up the computations.

- Data-parallel processing maps data elements to parallel threads.

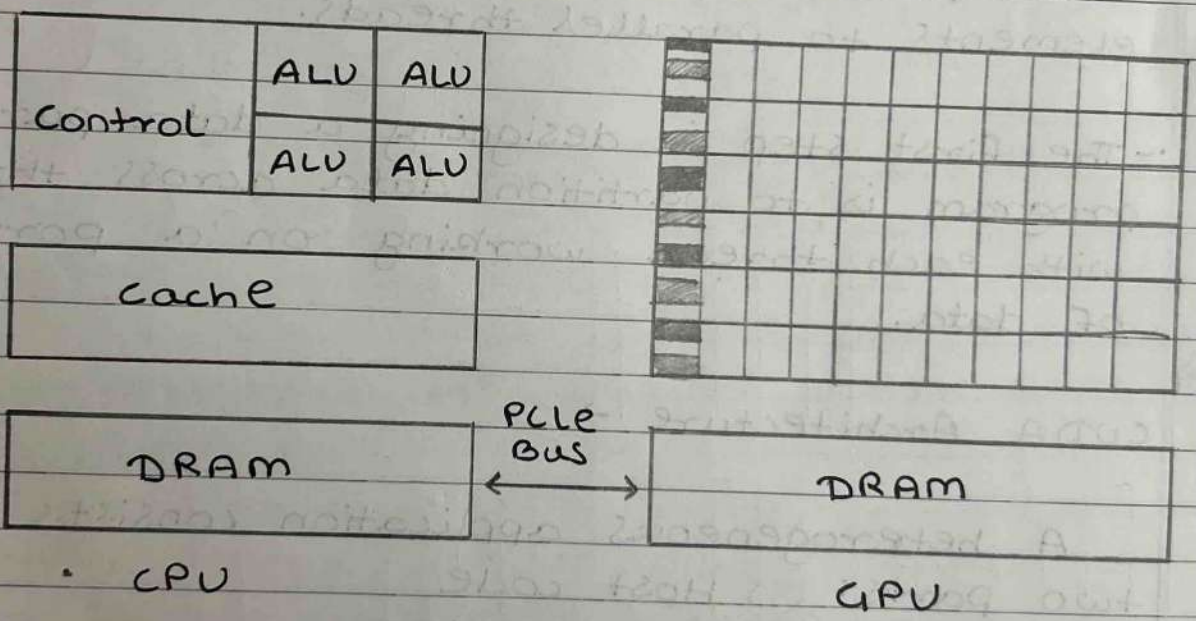
- The first step in designing a data parallel program is to partition data across threads, with each thread working on a portion of data.

CUDA Architecture →

A heterogeneous application consists of two parts: (i) Host code
(ii) 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 task on 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 hardware component that is physically separate from the CPU is used to accelerate computationally intensive sections of an application, it is referred to as hardware accelerator.



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 extented with keywords for labelling data parallel functions, called kernels. This device code is further compiled by Nvcc.

matrix multiplication using CUDA C →

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

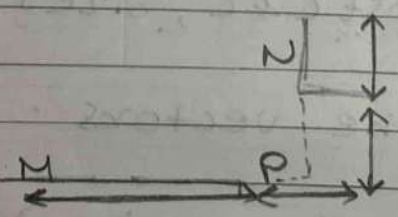
(i) Leave shared memory usage until later
(ii) Local, register usage
(iii) Thread ID usage.

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:

(i) matrix data transfer
(ii) Simple Host Code in C
(iii) Host - side main Program code
(iv) Device - side Kernel function
(v) Some Loose Ends.

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 engineering :

Conclusion : We learned parallel programming with the help of CUDA architecture.

Assignment Questions →

1. What is 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 data parallel model to speed up computations.

- Data parallel processing maps data elements to parallel threads.

- The first step in designing a data parallel program is to partition data across threads, with each thread working on a portion of data.

2. Explain Processing flow of CUDA programming.

→ (i) Load data into CPU memory

(ii) Copy data from CPU to GPU memory -
eg. cudaMemory(..., cudaMemcpyHostToDevice)

(iii) Call GPU kernel using device variable -
eg. kernel <<<>>>(gpuVar)

(iv) Copy results from GPU to CPU memory - eg. cudamemcpy(..., cudamemcpy DeviceToHost)

(v) Use results on CPU.

3. Explain advantages and limitations of CUDA.

→ Advantages -

(i) CUDA has full support for bitwise and integer operations.

(ii) Scattered read codes can be read from any address in memory.

(iii) Integrated memory (CUDA 6.0 or later) and Integrated virtual memory (CUDA 4.0 or later).

Disadvantages -

(i) CUDA source code is given on host machine or GPU, as defined by C++ syntax rules.

(ii) CUDA has unilateral interoperability with transferor languages like OpenGL.

(iii) CUDA supports only NVIDIA hardware.

4. Make the comparison between GPU and CPU.

→	CPU	GPU
-	Generalist component - Handles main processing functions of a computer.	Specialized component - Handles graphic and video rendering.
-	Core count - 2 - 64 (most CPUs)	Core count - thousands
-	Runs processes serially	Runs processes in parallel
-	Better at processing one big task at a time.	Better at processing several smaller tasks at a time.

5. Explain various alternatives to CUDA.

→ (i) OpenCL - It is open, royalty free standard for cross-platform

(ii) OpenGL - It is cross language, cross-platform application programming interface.

(iii) TensorFlow - It is open source software library for numerical computation.

(iv) PyTorch - It is not Python building into a mondothic C++ framework.

(v) Keras - Deep learning library for Python.

6. Explain CUDA hardware architecture in detail.

→ A heterogeneous application consists of two parts : (i) Host code
(ii) 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.