HPC

PRACTICAL NO-01

★ Title: Parallel Breadth first search & Depth first search.

★ Objective: To implement Parallel Breadth first search & depth first search.

★ PROBLEM STATEMENT: Design and implement Parallel BFS & DFS & based on existing algorithms using OpenMP. Use a Tree or undirected graph for BFS & DFS.

★ Theory:

Data structure: To perform BFS & DFS, we need to represent the tree or graph in a data structure. In this case, lets use an adjacency list to represent the undirected graph. for the tree, we can use a binary tree with left and right pointers.

BFS: To perform BFS in parallel using openMP we can use a queue to keep track of the nodes to visit. Each thread can take a node from the queue & visit all of its ~~integration~~ neighbours in parallel. We need to make sure to synchronize the threads after visiting all neighbours of a node before moving on to the next level of the BFS.

Parallel BFS:-

BFS is a graph traversal algorithm that explores all the vertices of a graph or tree level by level. To implement a parallel version of BFS using

openMP. we can use a shared queue data structure that will hold the vertices to be processed. Each thread will pick up a vertex from the queue, process it & add its unvisited neighbors to the queue & continue this process till queue is empty.

In this implementation, we first create a vector of bools to keep track of visited vertices, a queue to hold the vertices to be processed, and we push the start vertex into the queue and mark it visited. Then, we start an openMP parallel region & iterate until the queue is empty. In each iteration we use a critical section to pick a vertex from the queue, process it and add its unvisited neighbors to the queue.

The key feature of this implementation is the parallel for loop that iterates all over the neighbors of the current node in the graph. By using #pragma omp parallel for, openMP automatically distributes the loop iterations among multiple threads, enabling parallel processing of nodes at each level.

* Parallel Depth first search:- (DFS)

DFS is anothers graph traversal algorithm that explores all the vertices of a graph or tree by recursively exploring as far as possible along each branch before backtracking. To implement a parallel

version of DFS using openMP, we can use a stack data structure that will hold the vertices to be processed. Each thread will pick a vertex from the stack, process it and add its unvisited neighbors to the stack. This process will continue till the stack is empty.

In both examples, we are using openMP's critical directive to synchronize access to shared data structures (ie queue & stack). We could also use other synchronization mechanisms such as openMPs atomic directives or mutexes.

★ Conclusion:-
Implemented Parallel BFS (Breadth first search) and DFS (Depth firstsearch).

TSSM
Page No. : 5

PRACTICAL NO-02

* Title: Parallel Bubble sort & merge sort.

* Problem statement: write a program to implement Parallel Bubble sort & merge sort using openMP. Use existing algorithms & measure the performance of sequential & parallel algorithms.

* Theory:

Bubble sort & merge sort are two commonly used sorting algorithms. In this program, we will implement parallel versions of these algorithms using openMP & compare their performance to their sequential counterparts.

Parallel bubble sort is a parallel implementation of the classic bubble sort algo. The concept of parallelism involves executing a set of instructions/code simultaneously instead of line by line sequentially.

The main benefit is faster computation. The degree of parallelism depends on the program & its requirements. We can solve this using multi-core computers. In addition, even sequential programs involve some degree of parallelism.

Modern-day processors can handle multiple instructions & the data generated from them at the same time. The Hardware & the system are designed in such a way so that the data generated & the previous

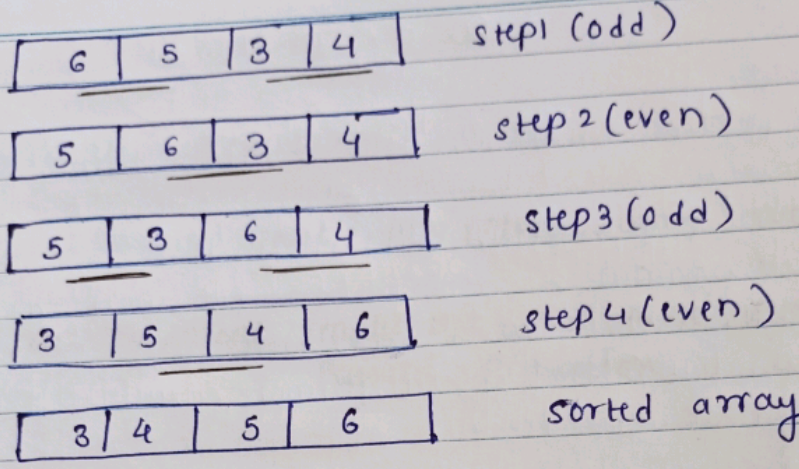
current & future instructions don't interrupt each other.

* In parallel bubble sort, we divide sorting of the unsorted into two phases- odd & even. We compare all pairs of elements in the list/array side by side (parallel).
- When it is the odd phase, we compare the element at index 0 with the element at index 1; the element of indext 2 with element at indext 3 and so on. In the even phase we compare element at index 1 with element at indext 2, index 3 element with index 4 element and so on, with our last comparison being at element at third last index & element at second last index.
- When comparing just like in normal bubble sort, we swap the elements, if initial element is greater than next element in comparison.

* Merge sort is a divide and conquer algorithm. Given an input array, it divides it into halves & each half is then applied the same dirision method until individual elements are obtained. A pair of adjacent elements are then merged to form sorted sublists until one fully sorted list is obtained.

To perform parallel merge sort you must know the traditional merge sort algorithm.

The process of parallel bubble sort is shown below:



In the case of multicore processors, both phases occur simultaneously, known as parallel implementation.

* Parallel merge sort algorithm:
The key to designing parallel algorithms is to find steps in the algorithm that can be carried out concurrently That is, examine the sequential algorithm & look for operations that could be carried out simultaneously on a given number of processors.

* Conclusion:
Implemented parallel Bubble sort and Parallel Merge sort using openMP.

PRACTICAL NO-03

* Title: Parallel Reduction
* Objective: To implement Min, Max, Sum & average operations.
* Problem statement: Implement Min, Max, Sum & Average operations using Parallel Reduction.
* Theory:

Parallel Reduction:

One common approach to this problem is parallel reduction. This can be applied for many problems, a minim operation just one of them.

It works by using half the number of threads of elements in the dataset. Every threads calculates the minimum of its own element & some other element. The resultant element is forwarded to the next round. The number of threads is then reduced by half and the process is repeated until there is just a single element remaining which is the result of the operation.

When selecting the 'other element' for a given thread to work with you can do so to do a reduction within the wrap, thus causing significant branch divergence within it. This will hinder the performance as each divergent branch doubles the work for the SM.

A better approach is to drop whole wraps by selecting the other element from the other half of the dataset.

Parallel reduction is a technique used in parallel computing to reduce the results of an operation on a set of input data into a single value. In this case, we will be discussing the implementation of ~~st~~ Min, Max, sum & Average operations using parallel reduction.

To implement these operations using Parallel Reduction, we need to follow these steps:

1) Divide the input data into equal-size chunks among the available threads.
2) Calculate the local Min, Max, Sum & Average for each thread on their respective chunks.
3) Combine the local Min, Max, Sum & Average values across all threads using the parallel reduction technique
4) Finally, obtain the final Min, Max, Sum and Average values by combining the results of all threads.

★ Conclusion :-
We have successfully implemented Min, Max, Sum & Average operations using parallel reduction.

HPC

PRACTICAL NO-04

★ Title: CUDA program.

★ Objective: Demonstrate how to use CUDA C to perform vector addition and matrix multiplication on GPU.

★ Problem statement: Write a CUDA program for: 1. Addition of two large vectors 2. Matrix multiplication using CUDA C.

★ Theory:

• Programming structure of GPU and CPU:-

• CUDA Kernel:-

The function which are executed on GPU are called as kernels. Kernels are full program or function invoke by the CPU & executed on GPU. A kernel is executed N number of times in parallel on GPU by using N number of threads.

Invocation: Kernel_name<< grid, block >>(argument, list);

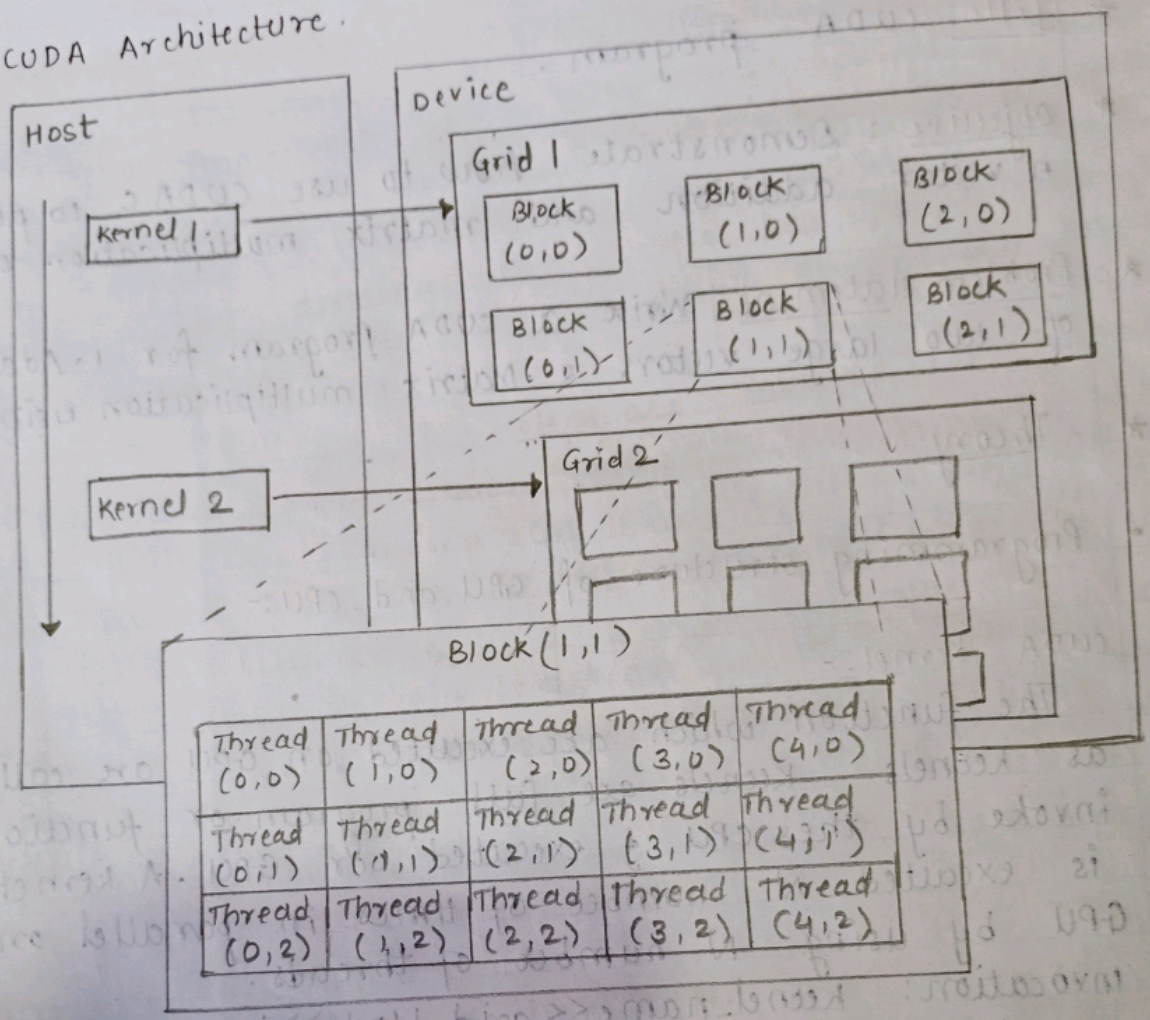
Kernel is defined as:

```
__global__ void
kernel_name(arguments)
{
....
}
```

★ Parallel Vector addition:-

Addition of Two Large Vectors:- Vector addition is

* CUDA Architecture.



using cudaMemcpy()

simple yet computationally intensive operation that involves adding the corresponding elements of two vectors. In CUDA C, vector addition can be implemented by dividing the vectors into blocks and threads & performing the addition in parallel on the GPU.

The following steps are involved in implementing vector addition using CUDA C:

- Allocate memory on the device (GPU) using cudaMalloc()
- Copy the input vectors from host (CPU) using cudaMemcpy() to device
- Launch the kernel on the GPU using the <<<>>> syntax
- Wait for the kernel to finish using cudaDeviceSynchronize()
- Copy the result back from device to host using cudaMemcpy()
- Free the memory on the device using cudaFree()

★ Matrix Multiplication:-

Matrix multiplication is a fundamental operation in linear algebra that involves multiplying 2 matrices to produce a third matrix. It is a computationally intensive operation that can benefit from parallelization on GPUs.

The following steps are involved in implementing matrix multiplication using CUDA C:

- Allocate memory on the device (GPU) using cudaMalloc()
- copy the input matrices from host (CPU) to device using cudaMemcpy()

- Launch the kernel on the GPU using the <<<>>> syntax
- wait for the kernel on th finish using cudaDeviceSynchronize()
- copy the result back from device to host using cudaMemcpy()
- free the memory on the device using cudaFree()

★ Conclusion:-

we have successfully implemented a recurrent neural network to create a classifier. we have used CUDAC to perform vector addition and matrix multiplication on the GPU.