Assignment No: Group A_01

Aim:

Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. For

1. Use a Tree
2. An undirected graph for BFS and DFS.

Objective:

Student will learn:

1. The Basic Concepts of DFS, BFS.
2. Multiple Compiler Directives, library routines, environment variables available for OpenMP

Theory:

Introduction:

Breadth First Search (BFS)

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

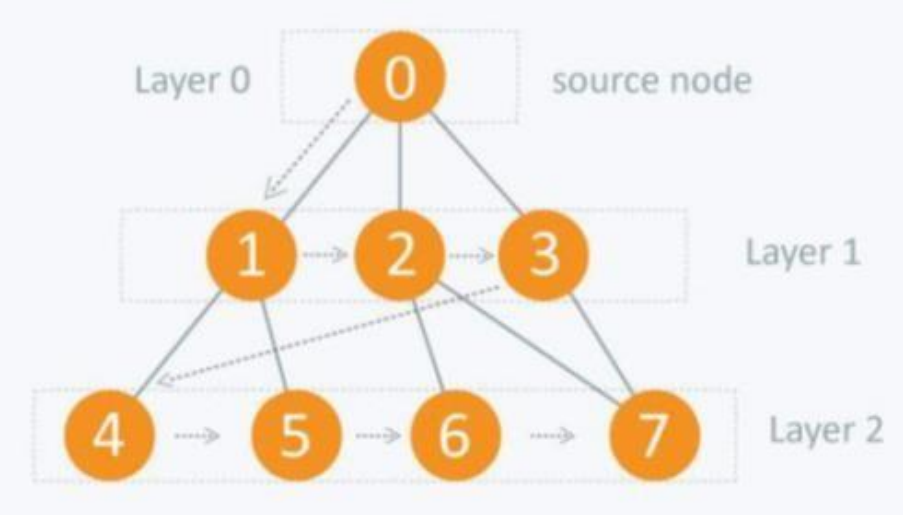
BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbor nodes.

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.



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

Sample Program:

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

using namespace std;

void bfs(vector<vector<int>>& graph, int start, vector<bool>& visited) {

#pragma omp task firstprivate(vertex)

{

for (int neighbor : graph[vertex]) {

if (!visited[neighbor]) { q.push(neighbor); visited[neighbor] = true; #pragma omp task bfs(graph, neighbor, visited);

}}}}

}

void parallel_bfs(vector<vector<int>>& graph, int start) {

vector<bool> visited(graph.size(), false);

bfs(graph, start, visited);

}

Parallel Depth First Search

- Different subtrees can be searched concurrently.
- Subtrees can be very different in size.
- Estimate the size of a subtree rooted at a node.
- Dynamic load balancing is required.

Parameters in Parallel DFS: Work Splitting

- Work is split by splitting the stack into two.
- Ideally, we do not want either of the split pieces to be small.
- Select nodes near the bottom of the stack (node splitting), or
- Select some nodes from each level (stack splitting)
- The second strategy generally yields a more even split of the space.

Sample Program

```
#include <iostream>

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

using namespace std;

void dfs(vector<vector<int>>& graph, int start,

vector<bool>& visited) { stack<int> s; s.push(start); visited[start] = true;

#pragma omp parallel{

#pragma omp single{

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

#pragma omp task firstprivate(vertex){

for (int neighbor : graph[vertex]) {

if (!visited[neighbor]) { s.push(neighbor); visited[neighbor] = true; #pragma omp task dfs(graph, neighbor, visited);

}}}}}

}

void parallel_dfs(vector<vector<int>>& graph, int start) { vector<bool> visited(graph.size(), false);

dfs(graph, start, visited);

}
```

OpenMP Section Compiler Directive:

A parallel loop is an example of independent work units that are numbered. If you have a pre-determined number of independent work units, the sections is more appropriate. In a sections construct can be any number of section constructs. These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.

The sections construct is a non-iterative work sharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task.

Execute different code blocks in parallel

```
#pragma omp sections
{
    #pragma omp section { ... }
     ...
    #pragma omp section { .. }
}
```

OpenMP Critical Compiler Directive

The omp critical directive identifies a section of code that must be executed by a single thread at a time..

```
#pragma omp critical
{
  code_block
}
```

Conclusion: Thus, we have successfully implemented parallel algorithms for Binary Search andBreadth First Search.

Assignment No: Group A_02

Aim:

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

Objective:

Student will learn:

i) The Basic Concepts of Bubble Sort and Merge Sort.

ii) Multiple Compiler Directives, library routines, environment variables available for OpenMP.

Theory:

Introduction:

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

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 (if necessary) until the end of the array is reached. Repeat this process from the beginning of the array n times. Average performance is $O(n^2)$

Bubble Sort Example

Here we want to sort an array containing [8, 5, 1].

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 n^{th} 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] ↔ 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] ↔ A[2i+2]

10. Next k

Parallel Bubble Sort Example

• 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

Merge Sort

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

Steps of Merge Sort:

To sort A[p .. r]:

1. 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].

2. Conquer Step

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

3. Combine Step

Combine the elements back in A[p .. r] by merging the two sorted subarraysA[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)

Parallel Merge Sort Example

- Perform Merge Sort on the following list of elements. Given 2 processors, P0 & P1.
- 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

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(nlogn)**

Conclusion: Thus, we have successfully implemented parallel algorithms for Bubble Sort and Merger Sort.

Assignment No: Group A-3

Aim: Implement parallel reduction using Min, Max, Sum and Average Operations.

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

Pre-requisites:

64-bit Open source Linux or its derivative

Programming Languages: C/C++

Theory:

OpenMP:

OpenMP is a set of C/C++ pragmas 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 coordinating threads. For example, the OpenMP directive.

OpenMP Core Syntax:

Most of the constructs in OpenMP are compiler directives:

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

1. 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.
2. Similarly, the max_reduction function finds the maximum value in the array, sum_reduction function finds thesum 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.
3. 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. Inthe main function, it creates a vector and calls the functions min_reduction, max_reduction, sum_reduction, andaverage_reduction to compute the values of min, max, sum and average respectively.

Example:

```
#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) {
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(vecto <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);
}
```

Conclusion: Thus, we have successfully implemented parallel reduction using Min, Max, Sum and Average Operations.

Experiment No: 4 Group A

Aim: To write a CUDA program for, find.

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

Objective: To study and implement the CUDA program using vectors.

Pre-requisites:
64-bit Open source Linux or its derivative
Programming Languages: C/C++, CUDA

Theory:

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 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 the data. 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.

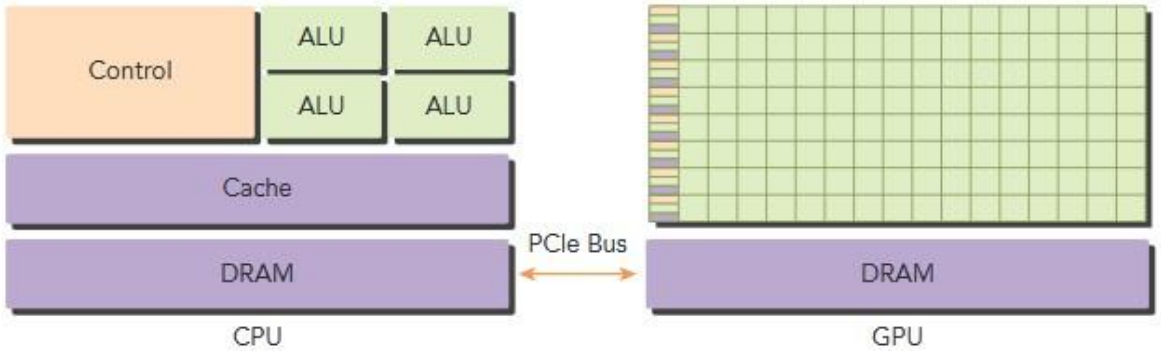
CUDA Architecture:

A heterogeneous application consists of two parts:

- Host code
- Device code

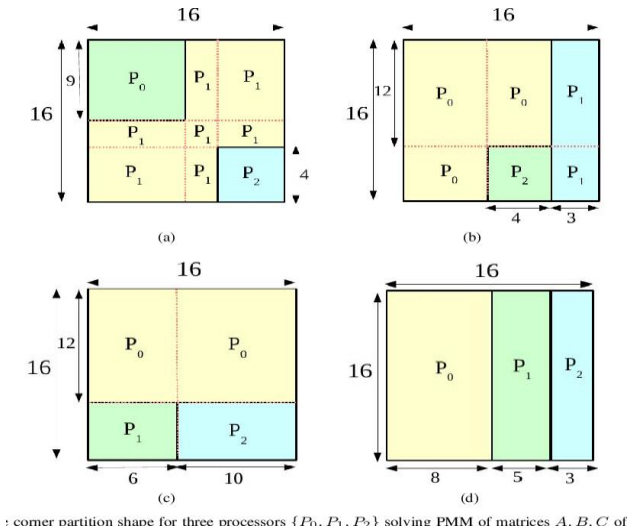
Host code runs on CPUs and device code runs on GPUs. An application executing on a heterogeneous platform is typically initialized by the CPU. The CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks on the device. With computational 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.



Matrix-Matrix Multiplication

- Consider two *n* x *n* matrices *A* and *B* partitioned into *p* blocks $A_{i,j}$ and $B_{i,j}$(0≤ *i*<j) of size each. $(n/\sqrt{p}) \times (n/\sqrt{p})$
- Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix.
- Computing submatrix $C_{i,j}$ requires all submatrice $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$
- All-to-all broadcast blocks of *A* along rows and *B* along columns.
- Perform local submatrix multiplication



: corner partition shape for three processors $\{P_0, P_1, P_2\}$ solving PMM of matrices A, B, C of

Conclusion: Thus, we have successfully implemented the Addition of two large vectors and Matrix Multiplication using CUDAC Programming.