Rajdhani
DATE / /

Aim: Design and Analyse Parallel Breadth First search and Depth first search based on existing algorithms using OpenMP. Use a tree or an unidirected graph for BFS and DFS.

Pre-requisites: 64-bit open source linux OS
C/C++ programming language.

Theory: Breadth First Search-
The Breadth First search algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the trees root or graph and searches all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory

Algorithm:
step 1: Consider the graph you want to navigate
step 2: Select any vertex in your graph (say V1) from which you want to traverse the graph.
step 3: Utilize the following two data structures for traversing the graph.

Rajdhani
DATE / /

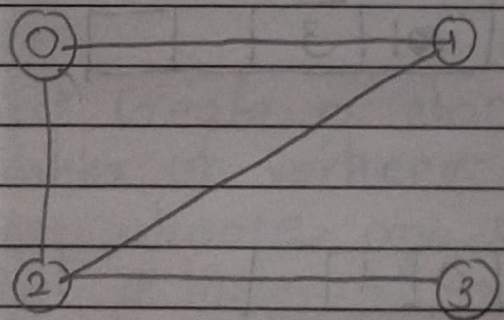
i) Visited array
ii) Queue

step 4: Add the starting vertex to the visited array and afterward, you add v1's adjacent vertices to the queue data structure.

step 5: Now using the FIFO concept, remove the first element from the queue, put it into the visited array and then add the adjacent vertices of the removed elements to the queue.

step 6: Repeat step 5 until the queue is not empty and no vertex is left to be visited.

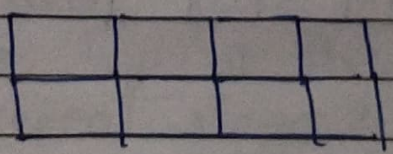
Example,



starting vertex: 2

step ①

visited:



Queue

Rajdhani
DATE / /

step 2:

visited : | 2 | | | | |

Queue | 2 | | | | |

step 3:

visited : | 2 | | | | |

Queue : | 0 | 1 | 3 | | |

traverse
step 4:

visited : | 2 | 0 | 3 | | |

Queue : | 1 | 3 | | | |

step 5:

visted : | 2 | 0 | 1 | | |

Queue : | 3 | | | | |

step 6:

visited : | 2 | 0 | 1 | 3 | Queue : | | | | |

Rajdhani
DATE / /

Parallel Breadth first search:

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 / thread.

Each processor / thread will then perform a breadth first search on its assigned sub-graph concurrently with the other processors / threads.

Depth First search:

Depth first search algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any situation.

Algorithm:

step 1: First create a stack with the total number of vertices in the graph.

step 2: Now, choose any vertex as the starting point of traversal and push that vertex into the stack.

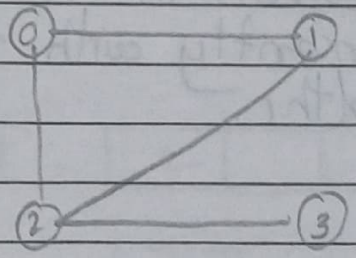
step 3: After that, push a non-visited vertex to the top of the stack.

step 4: Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.

Rajdhani
DATE / /

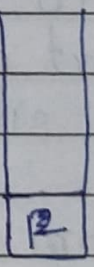
step5: If no vertex is left, go back and pop a vertex from the stack.
step 6: Repeat steps 2, 3, and 4 until stack is empty.

Example:

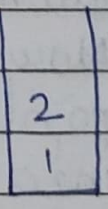


Start Node: 1

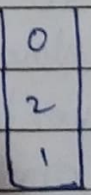
step 1:
visited: 1 stack:



step 2:
visited: 1, 2 stack:



step 3:
visited: 1, 2, 0 stack:



Rajdhani
DATE / /

step 4:
visited: 1, 2, 0 stack: | 2 | 1 |

steps:
visited: 1, 2, 0, 3 stack: | 3 | 2 | 1 |

Parallel Depth First search:

Here, the dfs function uses a stack to keep track of the vertices to visit. The # pragma omp parallel directive creates region and the # pragma omp single directive creates single execution context within that region.

Inside the while loop, the # pragma omp task task directive creates a new task for each unvisited neighbour of the current vertex. This allows each task to be executed in parallel with each other tasks. The first private clause is used to ensure that each task has its own copy of the vertex variable.

This method can be used for both trees as well as graph as both the data structures can be represented by adjustancy list.

Conclusion:

We have implemented Parallel Breadth First Search and Depth First Search based on existing algorithms using Openmp.

Rajdhani
DATE / /

Aim: write a program to implement parallel bubble sort and merge sort using OpenMp. Use existing algorithm and measure the performance of sequential and parallel algorithm

Requirements - 64-bit open source linux distro, C/C++ programming language

Theory: Sorting -
Sorting is a process of arranging elements in a groups in a particular order i.e. ascending order, descending order etc.

Parallel Sorting -
A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data.

Bubble Sort -
Bubble sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It is called "bubble sort" because the algorithm moves the larger elements towards the end of the array in a

manner that resembles the rising of bubble in a liquid.

Bubble sort Algorithm:
① Start at the beginning of array.
② Compare the first two elements. If the element is greater than the second element, swap them.
③ Move to the next pair of elements and repeat step.
④ Continue process untill the end of array is reached
⑤ If any swaps were made in step 2-4 repeat the process from step 1.

Paralled bubble sort :-
It is implemented as a pipeline. let local-size = n/no-process. we divide the array into no-proc parts and each process executes bubble sort on its part, including comparing the last element with the first one belonging to the next thread. Implement with the loop for (j=0; j<n-1; j++)

Rajdhani
DATE / /

Algorithm for parallel bubble sort:

```
for k=0 to n-2
    if k = even then
        for i=0 to (n/2)-1 do in parallel
            if A[2i] > A[2i+1] then
                exchange A[2i] ↔ A[2i+1]
        else
            for i=0 to (n/2)-2 do in parallel
                if A[2i+1] > A[2i+2] then
                    exchange A[2i+1] ↔ A[2i+2]

next k
```

Merge sort -

Merge sort is a sorting algorithm that user a divide and conquer approach to sort an array or a list of element.

The algorithm work by recursively dividing the input array into two halves, sorting each half and then merging the sorted halves to produce a sorted output.

Parallel Merge sort:

Parallel merge sort is a paralellized version of merge sort algorithm that taker advantage of multiple processors or corer to improve its performance. The merging step in parallel merge sort as performed

in a similar way to merging step in sequential merge sort algorithm.

Parallel merge sort can provide-
Significant performance benefits for large input arrays with many elements especially additional running on hardware with multiple processors.
However, it also requires additional overhead to manage parallelization and may not always provide performance improvements for smaller input sizes.

Conclusion:
In this way we had implemented bubble sort in parallel way using openMp.

Rajdhani
DATE / /

Aim: Implement min, max, sum and average operations using parallel reduction.

Objectives:
Students to study and implement derivative based parallel programming model

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

Theory: OpenMP.
OpenMP is a set of C/C++ programs which provides the programmer a high-level frontend interface which gets translated at calls to threads. The key phrase here is 'higher level'. The goal is to better enable the programmer to think parallel. i.e. OpenMp directive.
OpenMP core Syntax
#pragma OMP construct [clause [clause]...]

#pragma omp parallel num threads (4)

Rajdhani
DATE / /

Most OpenMp constructs apply to a 'structured block'.

Implementation of max-

```
void main_reduction (vector<int> &arg){
    int maxvalue = 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;
}
```

The max_reduction function finds the maximum value in the array. Sum_reduction function finds the sum of the elements of array and average of the elements of array by dividing the sum by the size of the array.

The min_reduction function finds the minimum value in input array using # pragma omp parallel for reduction

Rajdhani
DATE / /

```
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]
        }
    }
```

The min_reduction function Finds the minimum value in input array using # pragma omp parallel region and divides the iterative among the available threads. Each threads perform comparison operations in parallel and updates the min_value variable if a smaller value is found.

Conclusion:
We have implemented parallel reduction using min, max, sum and average operations.

Page No. 1
Date

HPC Practical Assignment 4

Aim.
Write a CUDA program for:
1. Addition of two large vectors
2. Matrix Multiplication using CUDA C.

Prequerites
- 64 bit Open source linux or its derivative
- C++ | JAVA | C | R | PYTHON

Theory
CUDA stands for Compute Unified Device Architecture
It is parallel computing platform & application programming interface (API) model created by NVIDIA.
CUDA allows developers to harness the power of NVIDIA GPUs (Graphics processing units) for general purpose computing tasks, beyond their traditional use in graphics rendering.

GPUs are highly parallel processors designed for efficient computing tasks, beyond their traditional use in graphics rendering.

GPUs are highly parallel processors designed for efficient computation of large amounts of data. CUDA enables developers to write programs that can execute on the GPU, taking advantage of its massive parallelism & high performance capabilities. This is particularly useful for computationally intensive tasks such as scientific simulations, deep learning, data analytics & image processing.

Addition of two large vectors using CUDA:

1. Allocate memory on GPU for the input vectors & output vector
2. Copy the input vectors from the CPU memory to the GPU memory.
3. Define the kernel function that will be the executed on the GPU. This function will perform the addition of the corresponding elements from the input vectors

& store the result in the output vector.

④ launch the kernel on the GPU specifying the number of threads & blocks to use.

⑤ wait for GPU computation to complete.

⑥ copy the result from Device (GPU) memory to Host (CPU) memory.

⑦ free the allocated GPU memory.

Matrix Multiplication

① Allocate memory on the CPU for the input matrices (A, B) & out matrix C.

② Initialize the input matrices A & B

③ Allocate the input ~~mate~~ memory on the GPU for A, B and C.

④ Copy the matrices from CPU to GPU memory.

⑤ Define the block size and grid size for launching the kernel.

⑥ Define the kernel function that will be executed on the GPU. This function will perform the matrix multiplication of the corresponding elements from A & B and store the result in matrix C.

⑦ launch the kernel on GPU using specified block size & size, passing the GPU memory pointers for d-A, d-B & d-
⑧ wait for GPU computation
⑨ free the allocated GPU memo
⑩ Process or display the result matrix 'c' as derived.
⑪ free the allocated CPU memo

Conclusion: We successfully implem addition of two large ~~matrices~~ vectors & matrix multiplication in CUDA.