PRATHAMESH YADAV
CO4085 (BE COMP)
B3 Batch

H.P.C

EXPERIMENT - 01

* AIM:- Design and Implement Parallel Breadth First Search and Depth First Search based on existing algorithms using Open MP.

* OBJECTIVE :- Use a Tree or an undirected graph for BFS and DFS.

* Hardware / Software Requirment :-

Vs Code; gcc compile, i3 processor, 8 gb RAM, 500 gb SSD.

* THEORY :-

Parallel Breadth First Search

Steps for bfs

1. In this implementation, the parallel bfs function take in a graph represented as an adjacency list, where each element in the list is a vector of neighbouring vertices, and a starting vertex.

2. The bfs function uses a queue 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.

3. Inside the while loop, the # pragma omp task directive create a new task for each unvisited neighbour of the current vertex.

4. This allows each task to be executed in parallel with other task. The first private clause is used to ensure that each task has its own copy of the vertex variable.

5. This is just one implemention, and there are many ways to improve it depending on the specific requirement of your application. For example, you can use omp atomic or omp critical to protect the shared resource queue.

Parallel Depth-First Search

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 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. 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.

8. The # pragma omp parallel directives create a parallel region & the # pragma omp single directive create a single execution context within that region.

CONCLUSION :-

Hence we have successfully performed bfs and dfs algorithm.

PRATHAMESH YADAV
C040 85 (BE COMP)
B3 Batch

H.P.C

EXPERIMENT - 02

* AIM :- Write a program to implement Parallel Bubble Sort and Merge sort using open MP. Use existing algorithm and measure performance of sequential & parallel algorithms.

* OBJECTIVES :-
  1. Implement parallel bubble and merge sort.
  2. measuring the performance using sequential and parallel algorithm.

* Hardware / Software Requirements :-
  VS Code, gcc Compiler, i3 processor, 8 gb RAM, 500 gb SSD.

* THEORY :-

BUBBLE SORT

1. The complexity of bubble sort is $\theta(n^3)$
2. Bubble sort is difficult to parallelize since the algorithm has no concurrency.
3. A simple variant, though, uncovers the concurrency.

Parallel Odd-Even Transposition

1. Consider the one item per processor case.
2. There are n iterations, in each iteration, each processor does one compare exchange.
3. The parallel run time of this formulation is $\theta(n)$.
4. This is cost optimal with respect to the base serial algorithm but not the optimal one.

$$T_p = \theta\left(\frac{n}{p}\log\frac{n}{p}\right) + \theta(n) + \theta(n)$$

local sort · comparison · communication.

Parallelize Bubble sort Algorithm

1. The # pragma omp parallel for directive tells the compiler to create a team of threads to execute the for loop within the block in parallel.
2. The bubble sort function takes in an array, and it sort it using the bubble sort algorithm.

3. The main function create a sample array & calls the bubbleSort function to sort it. The sorted array is then printed.

4. In this implementation, the bubble_sort_odd_even function takes in an array and sorts it using the odd-even transposition algorithm.

5. The two # pragma omp parallel for inside while loop, one for indexes and one for odd indexes, allows each thread to sort the even and odd indexed elements simultaneously & prevent the dependency.

Parallel Merge Sort

Given a set of elements A = {a... ~, ~, ..., an}, A odd and A even are defined as the set of elements of A with odd and even indices, respectively.

For example, Aodd = {a, ... a3, a'', ..} & Aeven = {~, a4, a6, ..} regarding a set of elements A = {a, ... an};

III'ly, let a set of elements B = {b.... bn}. We can then define the merge operation as:

Merge (A,B) = {a, b, a2, b2, a3, b3, ... an, bn}

for example,

if A = {1, 2, 3, 4} & B = {5, 6, 7, 8}
then Merge ({1, 3, 3, 4}, {5, 6, 7, 8}) = {1, 5, 3, 6, 3, 7, 4, 8}

Join (A,B) = (Merge (A,B), Odd-Even (A,B))

CONCLUSION :-

Hence, we have successfully performed the parallel bubble sort and merge sort algorithm & check the performance of sequential & parallel algorithm.

PRATHAMESH YADAV
C04085 (BE COMP)
B3 Batch

H.P.C

EXPERIMENT - 03

* AIM :- Implement Min, Max, Sum and Average operations using Parallel Reduction.

* OBJECTIVE :- Perform the Min, Max, Sum and Average operation using parallel reduction technique.

* Hardwaih / Softwaih Requirement :-
Vs Code., gcc compiler, i3 processor, 8 GB RAM, 500 gb SSD.

* THEORY :-

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 divide the loop iteration among the available threads. Each thread performs the comparison operation in parallel and update 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 operation are used for the min_reduction and max_reduction function respectively, and the + operator is used for the sum_reduction and average_reduction function. In the main function, it creates a vector and calls the function min_reduction, max_reduction, sum_reduction, and average_reduction to compute values of min, max, sum and average respectively.

Min_reduction function :-

The function takes in a vector of integers as input & finds the minimum value in the vector using parallel reduction.

Max_Reduction function :-

The function takes in a vector of integers as input & finds the maximum value in the vector using parallel reduction.

Sum_Reduction function :-

The function takes in a vector integers as input & finds the sum of all the values in the vector using parallel reduction.

Average_Reduction function :-

The function takes in a vector of integers as input & finds the average of all the values in the vector using parallel reduction.

* CONCLUSION :-

We have implemented the Min, Max, Sum and Average operations using parallel reduction in C++ with Open MP.

H.P.C

PRATHAMESH YADAV
C04085 (BE COMP)
B3 Batch

EXPERIMENT - 04

* AIM :- Write a CUDA Program for :
  1. Addition of two large vectors
  2. Matrix Multiplication using CUDA C

* OBJECTIVE :- Perform the addition operation of two vector and Multiplication of matrix using CUDA C.

* Hardware / Software Requirement :- VS Code, gcc compiler, i3 processor, 8 GB RAM, 500 gb SSD.

* THEORY :-

CUDA (Compute Unified Device is a parallel computing platform and programming model developed by NVIDIA. It support popular programming language like C, C++ and Python, and provide a simple programming model that abstracts away much of the low-level detail of GPU architecture.

CUDA, developers can exploit the massive parallelism and high computational power of GPU to accelerate computationally intensive tasks.

such as matrix operations, image processing, and deep learning.

Addition of Two vector

In this program, the 'addVectors' kernal takes in the two input vectors 'A' and 'B', the output vector 'C', and the size of the vector 'n'. The kernal uses the 'blockIdx.x' and 'threadIdx.x' variable to calculate the index 'i' of the current thread. If the index is less than 'n', the kernal performs the addition operation 'C[i] = A[i] + B[i]'.

In the 'main' function, the program first allocates memory for the input and output vectors on the host and initializes them. Then it allocates memory for the vectors on the device and copies the data from the host to the device using 'cudaMemcpy'.

Matrix Multiplication

In this program, the 'matmul' kernel takes in the two input matrices 'A' and 'B', the output matrix 'C', and the size of matrices 'N'. The kernal uses the 'blockIdx.x', 'blockIdx.y', 'threadIdx.x', and 'threadIdx.y' variable to calculate the indices of the current thread. If the indices are less than 'N', the kernal perform the matrix multiplication operation 'Pvalue += A[Row * N+k] * B[k*N+Col]' and store the Pvalue in 'C[Row*N+Col]'.

In the 'main' function, the program first allocates memory for the input and output matrices on the host and initializes them. Then it allocates memory for the matrices on the device using 'cudaMemcpy'.

Next, the program launches the kernel with the appropriate grid and block dimensions. The kernal uses a 2D grid of thread block to perform the matrix multiplication in parallel.

Finally, it copies the data from device to host using cudaMemcpy and prints the result using nested for loop. And it also free the memory used.

CONCLUSION :-

Hence we had succesfully implemented the addition and Multiplication operation in CUDA C.