BE-10 Suhani Gaikwad

LP V High Performance Computing

TSSM

Page No.

Assignment-1

AIM :- Design and implement parallel Breadth first search and Depth first search based on existing algorithms using OpenMp. use a tree or undirected graph for BFS & DFS.

THOERY :-

Parallel BFS :-

1. To design and implement parallel BFS, we will need to divide the graph into smaller sub-graphs & assign each sub graph to a diffrent processor or thread.
2. Each processor or thread will then perform a BFS on its assigned sub-graph concurently with other processors or threads

IMPLEMENTATION - to design & implement parallel BFS using openMP, we can use the existing BFS algorithm & parallelise it using OpenMps, palatalisation constructs.

1. In this implementation, BFS uses 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 # pragma omp single directive creates a single execution context within that region.

2. inside the while loop, the # pragma omp task directive creates a new task for each unvisited neighbour of the current vertex.

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

4. This is just one possible implementation of there are many ways to improve it depending on the specific
Eg:- you can use omp atomic or omp critical to protect the shared resources queue.

Parallel DFS :-

DYNAMIC LOAD BALANCING :-

- when a processor runs out of work, it gets more work from another processor.
- this is done using work requests and responses in message passing machines & locking and extracting work in shared address space machines.
- On reaching final state at a processor, all processors terminate.
- unexpected states can be stored as local stacks at processors, the entire space is assigned to one processor.

PARAMETER IN PARALLEL DFS : WORK SPLITTING :-

- work is split by splitting stack into two.
- Ideally we don't want either of the split pieces to be small
- select nodes near the bottom of the stack (node splitting)
- Select some nodes from each level (stack splitting)
- the second strategy generally yields a more even split of space.

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

Parallel Merge sort :-

ALGORITHM:-

```
Algorithm odd-even ( A,B,S)
begin
    if A and B are of length 1 then.
        Merge A & B using one compare-and-exchange
    else
    begin
        compute Sodd and Seven in parallel do
        Sodd = Merge (Aodd, Bodd)
        Seven = Merge (Aeven, Beven)
        Sodd-Seven = Join ( Sodd, Seven)
        end
    end if
end.
```

IMPLEMENTATION :-

1) Given a set of elements A- {a, --- an}

2) Aodd and Aeven are defined as the set of elements of A with odd & even indices, respectively.

Example- Suppose the set of elements is:

S = { 2, 3, 6, 10, 15, 4, 5, 8 }

A = { 2, 6, 10, 15 }

B = { 3, 4, 5, 8 }

Then,

Merge (Aodd, Bodd) = { 2, 3, 5, 10 }

Merge (Aeven, Beven) = { 4, 6, 8, 15 }

The join operation

Join (A, B) = { Merge (A, B), odd-even (A, B) }

requires a merge operation, which results in

Merge (A, B) = { 2, 4, 3, 6, 5, 8, 10, 15 }

and an odd-even operation, which obtains final sorted list

odd-even { 2, 4, 3, 6, 5, 8, 10, 15 }

= { 2, 3, 4, 5, 6, 8, 10, 15 }

Conclusion -

In this way we have implemented bubble sort and merge sort using parallel computing.

BE-10 Suhani Gaikwad
LP V High Performance Computing

TSSM
Page No.

Assignment-2

AIM :- Write a program to implement parallel bubble sort and merge sort using OpenMP. Use existing algorithm and measure the performance of sequential & parallel algorithm

THEORY :

Bubble Sorting :-

PARALLEL ODD-EVEN TRANSPOSITION :-

1. Consider the one ste 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 run time respect to the bare serial algorithm but not the optimal one.

```
Procedure ODD-EVEN-PAR (n)
begin
    id:= process's label.
    for i=1 to n do begin.
        if i is odd then
            if id is odd then
                compare-exchange min(id+1);
            else: compare-exchange-max(id-1);
        if i is even then
            if id is even then: compare-exchange min(id+1);
            else: compare-exchange-max(id-1);
    end for
end ODD-EVEN-PAR
```

TSSM
Page No.

IMPLEMENTATION -

1. This program uses OpenMP to parallelize the bubble sort.

2. this #pragma omp parallel directive tells the compiler to create a team of threads to execute for loop in parallel.

3. Each thread will work on a diffrent iteration of the loop, in this case on comparing & swapping the elements of array

4. The bubble sort function takes in an array, and it sort it using the bubble sort algorithm. the outer loop iterations from 0 to n-2 and the inner loop iterates from 0 to n-i-1, where i is the index of the the outer loop. the inner loop compares the current element with the next element, & if the element is greater than the next element they are swapped.

5. The main function creates a simple array & calls bubbleSort function to sort it.

6. In this implementation, the bubble-sort-odd-even function takes in an array & sorts it using the odd-even transpos algorithm. the outer while loop continues until the array is sorted. Inside the loop, the #pragma omp parallel directive creates a parallel region and divides the loop iterations among the available threads. Each thread performs the swap operation is parallel, improving the performance of the algorithm.

TSSM
Page No.

LOAD-BALANCING SCHEMES :-

1. Asynchronous Round Robin : Each processor - maint s a counter and makes request in a round-robin fashion.
2. Global Round Robin : The system maintains a global counter and requests are made in a round-robin fashion globally.
3. Random pooling : Request a randomly selected processor for work

ANALYZING DFS :-

- We cannot compute, analytically, the serial work W or parallel time, instead we quantify total overhead to in terms of W to compute scalability.

- for dynamic load balancing, idling time is subsumed by communication.

- We must quantify the total numbers of request in system.

TERMINATION DETECTION

- processor P0 has all the work & a weight of one is associated with it. when it work is partitioned & send to another processor, processor P0 retain half of the weight & gives half of the weight & gives half of it to the processor receiving the work.

- Each time the work at processor is partitioned weight is halved when a processor completes its computation, it return its weight to processor from which it received.

- Termination is signaled when the weight w0 at processor P0 becomes one & processor P0 has finished its work

IMPLEMENTATION:-

1. In this implementation, dfs does DFS util() for all unvisited

2. The DFS util function is utility function to do DFS of graph recursively from a given vertex u.

3. Uses a stack to keep track of the vertices to visit & a boolean visited away to keep track of which vertices have been visited.

4. The # pragma omp parallel directive creates a parallel region & # pragma omp single directive creates a single execution context within that region.

5. This implementation is suitable for both tree & undirected graph, since both are represented as an adjacency list and the algorithm is using a stack to traverse the graph.

Conclusion :-

In this way we have implemented Breadth first search and Depth first search using Parallel Computing.

BE-10 Suhani Gaikwad

LP V High Performance Computing

TSSM Page No.

Assignment-3

AIM :- Implement Min, Max, sum and Avg operations parallel Reduction.

THOERY -

- The min-reduction function finds are minimum values 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 threads performs the comparission operation in parallel and updates the min_value variable if a smaller value is found.

- Similarly, the max_reduction finds the maximum value is the array, sum_reduction finds the sum of elements of array and average_reduction finds the average of the elements of array by dividing the sum by the size of the array.

- The reduction closure is used to continue the results of mutiple threads into a single value, which is then returned by the function. the min and max operations are used for the min-reduction & max-reduction functions, resp and the + operator is used for the sum_reduction and average-reduction functions. In the main function, it creates a vector and calls the functions min_reduction, max_reduction, sum_reduction and average_reduction to compute the values of min, max, sum & average resp.