Report: Exercises 5

Sevde Yanik & Sima Esmaeili

Problem 1

In this exercise, we tackle the List Ranking algorithm. Initially, we present a sequential implementation of the algorithm. Afterwards, we provide a parallel implementation that utilizes OpenMP. To analyze the algorithm's performance, we conduct experiments using various list sizes and processor configurations. We examine how the running time scales with the problem size and evaluate the results in terms of speedup and efficiency.

Sequential List Ranking Algorithm:

1. Generate an array of fixed size with random values.
2. Create a sorted copy of the list in ascending order.
3. Iterate through each element in the original list.
4. Use binary search to find the position of each element in the sorted list.
5. The position represents the rank of the element.
6. Store the computed rank in a separate array called "ranks."
7. Print the "ranks" array to display the ranks of the elements in the original list.

Results:

List size	50	100	200	500	1000
Runtime	12 µs	25 µs	56 µs	346 µs	743 µs

Parallel List Ranking Algorithm using OpenMP:

1. Again, an array of fixed size with random values is generated.
2. Just like the sequential algorithm, we create a sorted copy of the list in ascending order.
3. We initialize an array called "ranks" to store the ranks of the elements.
4. To speed up the computation, we divide the work among multiple threads using OpenMP.
5. Each thread independently calculates the ranks for a subset of the elements.
6. For each element in the subset, we use a binary search to find its position in the sorted list.
7. The position in the sorted list represents the rank of the element.
8. We assign the computed rank to the corresponding element in the "ranks" array.
9. Once all the threads finish their computations, we synchronize them to ensure the ranks are correctly calculated.
10. Lastly, we print the "ranks" array, which displays the ranks of the elements in the original list.

Results:

List size	50	100	200	500	1000
Run time (#thread = 2)	61 µs	85 µs	152 µs	194 µs	296 µs
Run time (#thread = 4)	170 µs	173 µs	215 µs	344 µs	383 µs
Run time (#thread = 8)	237 µs	264 µs	366 µs	491 µs	538 µs

Speedup = normal runtime / parallel runtime:

List size	50	100	200	500	1000
Speedup (#thread = 2)	0,197	0,294	0,368	1,784	2,51
Speedup (#thread = 4)	0,071	0,145	0,26	1,01	1,94
Speedup (#thread = 8)	0,051	0,095	0,153	0,705	1,381

The results show that as the list size increases the speedup values show improvement, with some speedup values exceeding 1 for larger numbers of threads. This means that the parallel execution is outperforming sequential execution for larger problem sizes.

Another important thing to notice here is that the speedup values decrease as the number of threads increases for all list sizes. This means that the parallel execution with more threads is not achieving significant performance improvement compared to the sequential execution, the reason for this might be that overhead is introduced.

Efficiency = speedup / #threads

List size	50	100	200	500	1000
Efficiency (#thread = 2)	0,099	0,147	0,184	0,892	1,255
Efficiency (#thread = 4)	0,018	0,036	0,065	0,252	0,485
Efficiency (#thread = 8)	0,006	0,012	0,019	0,088	0,173

Looking at these results we can say that parallel execution is relatively more efficient for larger problem sizes, however the optimal utilization of the available threads is not achieved.

Problem 2

Parallel Algorithm for Computing the Diameter of a Graph using the CREW PRAM Model

Pseudocode:

```
function parallelBFS(graph, startVertex):
    n = number of vertices in the graph
    distances = array of size n initialized with infinity values
    distances[startVertex] = 0

    for level = 1 to ceil(log2(n)):
        newDistances = array of size n initialized with infinity values

        parallel for each vertex v:
            for each neighbor u of v in graph:
```

```
            newDistances[u] = min(newDistances[u], distances[v] + 1)

        distances = newDistances

    maxDistance = 0
    for each d in distances:
        maxDistance = max(maxDistance, d)

    return maxDistance

function computeDiameter(graph):
    n = number of vertices in the graph
    diameter = 0

    parallel for each sourceVertex = 0 to n-1:
        sourceDiameter = parallelBFS(graph, sourceVertex)
        diameter = max(diameter, sourceDiameter)

    return diameter

function main():
    n = number of vertices in the graph
    m = number of edges in the graph
    graph = adjacency list representation of the graph

    diameter = computeDiameter(graph)

    print "The diameter of the graph is: " + diameter
```

Algorithm Description:

1. Input: graph
2. Initialization: Set diameter = 0.
3. Parallel Iterations:
   Parallel for each source vertex in the graph:
   - Run a breadth-first search (BFS) starting from the source vertex.
   - Compute the maximum distance (eccentricity) from the source vertex to any other vertex.
   - Update diameter by taking the maximum of the current diameter and the computed eccentricity.
   - Output: final diameter

Analysis:

By performing BFS iterations in parallel, the algorithm can utilize O(n^2) processors, leading to a time complexity of O(n log n).

To explain in details:

- The BFS iterations in the algorithm are performed in parallel.
- Each BFS iteration explores a level of the BFS tree, which represents the shortest paths from the source vertex to all other vertices at that level.
- The number of levels in a BFS tree is proportional to the diameter of the graph.

- In each iteration, the algorithm updates the distances from the source vertex to all other vertices at the current level.
- The updates can be performed concurrently using O(n^2) processors, as each vertex can have connections to all other vertices in the worst case.
- Since there are O(log n) iterations (one for each level of the BFS tree), each iteration takes O(n) time due to the parallelism achieved using O(n^2) processors.
  Therefore, the total time complexity of the algorithm is O(n log n).