COMPSCI 711 - Parallel Computing | Assignment 2

Algorithm

The algorithm chosen to solve the problem utilises BFS to find the eccentricity of a single node, then repeats this for each node in the graph. The minimum of these eccentricities is calculated to be the radius of the graph. It also checks for connectedness, as this is not guaranteed. As it performs BFS, it marks each node as visited, if not all nodes are marked visited at the end of BFS execution, then an exception is thrown.

Sequential Algorithm

```
1  void findRadiusSequential(Graph graph, int numNodes)
2  {
3    vector<int> eccs(numNodes, 0);
4    bool notConnectedFlag = false;
5
6    for (int i = 0; i < graph.V; ++i)
7    {
8      try
9      {
10       int ecc = graph.BFS(i);
11       eccs[i] = ecc;
12     }
13     catch (const char *msg)
14     {
15       notConnectedFlag = true;
16       break;
17     }
18   }
19   if (notConnectedFlag)
20   {
21     cout << "None" << endl;
22   }
23   else
24   {
25     cout << *min_element(eccs.begin(), eccs.end()) << endl;
26   }
27 }
```

On line 16 it can be seen that the algorithm breaks out of the for loop if it detects that the graph is not complete. The complexity of this algorithm is $O(n^2)$. This would be in the worst case where the graph is fully connected. As n edges would need to be explored for every (n) nodes.

Parallel Algorithm

```
1  void findRadiusParallel(Graph graph, int numNodes)
2  {
3    vector<int> eccs(numNodes, 0);
4    bool notConnectedFlag = false;
5
6  #pragma omp parallel for
7    for (int i = 0; i < graph.V; ++i)
8    {
9      try
10     {
11       int ecc = graph.BFS(i);
12       eccs[i] = ecc;
13     }
14     catch (const char *msg)
15     {
16       notConnectedFlag = true;
17     }
18   }
19   if (notConnectedFlag)
20   {
21     cout << "None" << endl;
22   }
23   else
24   {
25     cout << *min_element(eccs.begin(), eccs.end()) << endl;
26   }
27 }
```

In contrast to the sequential algorithm, the parallel version can not end early as a break expression is not allowed in a pragma for loop. Therefore, theoretically, on non strongly connected graphs, the sequential formulation may be faster.

This parallel formulation does not scale well with #processors, as the maximum amount of processors it can handle at any time is n. Therefore, the isoefficiency is $O(n^2)$, although in cases where $p < n$ the parallel runtime is $O(n)$.

Test Data

Test Cases

- #1-3: The three examples in the assignment brief
- #4: Randomly generated 100 node, 4924 edge, Erdos-Renyi graph
- #5: Randomly generated 300 node, 44822 edge, Erdos - Renyi graph

Reasoning

Test cases #1-3 were chosen to test the correctness of the algorithm, as the expected answer was known for these three. Therefore, these were utilised throughout the development phase to periodically test the algorithm.

Test cases #4 and #5 were needed after the testing phase for performance results, as the original graphs are relatively small (3-5 nodes), and were being processed too quickly. Therefore, graphs test cases 4 and 5 are significantly larger in both number of nodes and edges.

Test cases #3 and #5 are not strongly connected, to measure the difference is performance of the algorithms, as the implementation for this functionality is different for the sequential and parallel algorithms.

The large graphs were generated using the networkx package for python. `n` can be changed to differ the size of the graph generated.

```
n = 300
G = nx.erdos_renyi_graph(n, 0.5, seed=123, directed=True)
for line in nx.generate_adjlist(G):
    print(line)
```

Testing Methodology

The tests were ran manually by running the program. The timing was taken using `chrono::high_resolution_clock` which measures time in "the smallest tick period provided by the implementation."

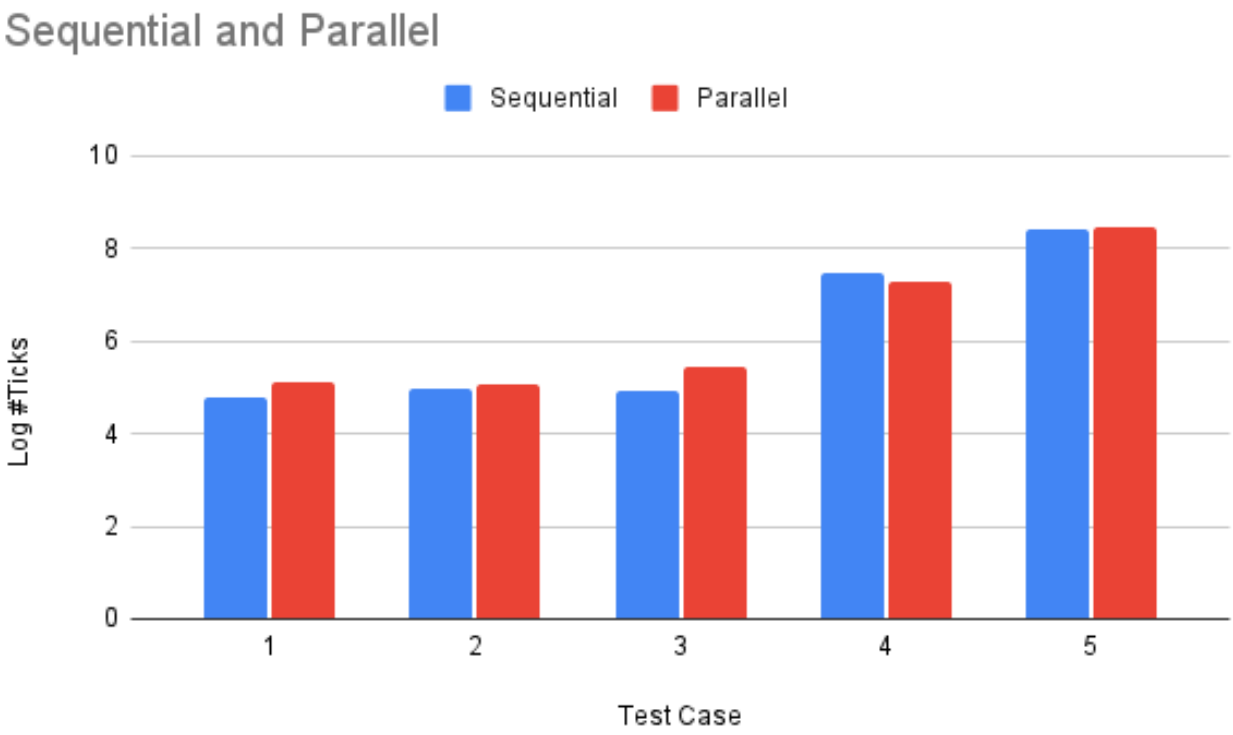
Performance Results

Results:

Algorithm	#1	#2	#3	#4	#5
Sequential	61000	92100	82900	3.02E7	2.69E8

Algorithm	#1	#2	#3	#4	#5
Parallel	133000	114000	263000	1.86E7	2.84E8

Graph of results:



As the tick operations span several orders of magnitude, they were plotted with log applied.

From the performance results, there is not a clear advantage between the sequential and parallel algorithms overall. I expected that the parallel implementation would be faster for all test cases except for #3 and #5 where the sequential algorithm is hypothesized to be faster as it ends early when detecting a not strongly-connected graph.

Possible explanations for this unexpected result:

- Thread overhead is always a consideration when running multiple threads, so with the lower node test cases, this definitely would be a significant factor. It is possible that even for the larger graphs this is a factor.
- The graphs that were tested, even #3 and #5 may not have been large enough. The tests always ran instantly, which may mean that differences are too small and the effectiveness of parallelisation is not visible yet.
- From the algorithm analysis, it is shown that the isoefficiency of the parallel formulation is not improved from the runtime of the sequential algorithm. Therefore, the speedup in theoretical terms is not significant, especially when

$p >> n$ which is the case with test case #4 and #5. However, practically, with 16 local threads there should be a significant speedup.