Question 3:

A) Implementation

My program implementation with OpenMP along with its README is included in this folder.

Brief Explanation of the Solution:

The Graph is initialized as an object that stores the number vertices along with its adjacent vertices in a list. The greedy algorithm initializes an array to store the final array with all vertices colored, and a Boolean array to store whether certain colors have been taken or not. Colors are represented as numbers in this situation. The greedy aspect is evident as the next available color will be used rather than attempting to minimize the total number of colors used, so colors (numbers in this case) aren’t reused regardless of whether or not there exists adjacent vertices with the same color.

Using OpenMP, a parallel directive specifying threads was utilized to speed up the algorithm. It was tested across varying graph sizes with the same number of threads for each graph. To evaluate accuracy, the output was manually reviewed to ensure it matched the criteria of the problem.

B) Strong vs Weak Scaling + Runtime

Runtime:  
The runtime of this implementation was tested by comparing the number of vertices and number of adjacent vertices to the overall runtime. Initially, each vertex was given one adjacent vertex.

For One Adjacent Vertex (For 10 Threads)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # Vertices | 10,000 | 20,000 | 40,000 | 80,000 |
| Runtime (s) | 0.064 | 0.185 | 0.614 | 3.966 |

Based on this factor, it appears the general runtime is O(nlogn), as the ratio between the differing vertices and differing runtimes resemble about a nlogn increase across trials. It’s not linear, but the increase in runtime while doubling the number of vertices is not quadratic either.

Varying # Adjacent Vertices For 50,000 Vertex Graph (For 10 Threads)

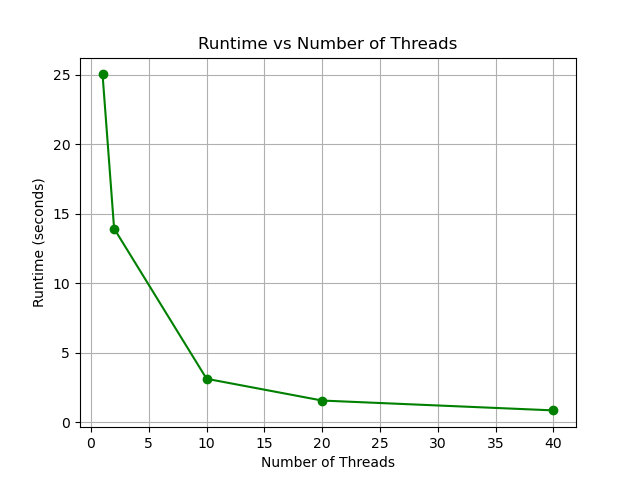
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # Adjacent Vertices | 1,000 | 2,000 | 4,000 | 8,000 |
| Runtime (s) | 4.117 | 7.198 | 15.103 | 31.116 |

This implementation appears as though it’s a linear relationship between number of adjacent vertices and runtime, showing O(n).

Strong:

To measure strong scaling, the number of threads was modified for a specific graph (100,000 vertices). Only one adjacent vertex was added.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # Threads | 1 | 2 | 10 | 20 | 40 |
| Runtime (s) | 25.038 | 13.934 | 3.113 | 1.545 | 0.836 |

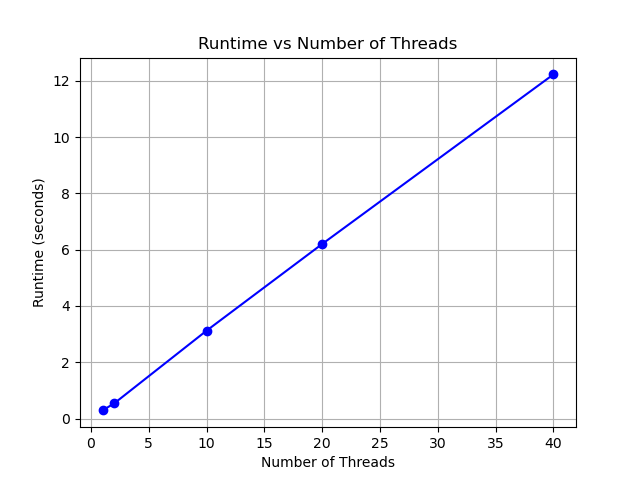


This program demonstrates very good strong scaling, as the problem size remains fixed, but the runtime initially decreases a very much greater slope than towards the end. There is a leveling off that takes place, though 40 threads are still more efficient than 20, showing that as scale between threads decreases, the change in runtime is minimal, whereas at the beginning the change in runtime is much greater.

Weak:

To measure weak scaling in this case, the number of threads provided to OpenMP varied with the size of the graph. Only one adjacent vertex was added for each.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # Threads | 1 | 2 | 10 | 20 | 40 |
| Graph Size | 10,000 | 20,000 | 100,000 | 200,000 | 400,000 |
| Runtime (s) | 0.289 | 0.536 | 3.124 | 6.205 | 12.232 |

This shows that the program exhibits poor weak scaling, as the runtime should remain about constant to show that the work per thread is remaining the same. The speedup here will likely not change, meaning that each thread does not have the same amount of work as the size of the workload and the number of threads changes.