Justin Bahr
NUID: 001592707

High Performance Computing
Homework 3 - Question 3

a. I ran my program on the COE Vector Linux platform (Intel ® Xeon ® CPU E5-2698 v4 @ 2.20GHz, 20 cores per socket, 2 sockets, with 792237220 KB of memory, and Linux 4.18.0). I created a graph class, where graphs were represented using a boolean adjacency matrix (1 for an edge, 0 for no edge). I tested my implementation by creating random graphs with a specified number of vertices. Each pair of vertices was randomly assigned to be an edge or not based on a 50% random probability. I tested strong scaling by using 1, 2, 4, 8, 16, and 32 threads on random graphs with 10,000 vertices. I tested weak scaling by using 1, 2, 4, 8, 16, and 32 threads on random graphs with 100 vertices per thread used. I then calculated a throughput of vertices per millisecond based on the run times observed. Results are shown and explained in part b. An example run is shown below:

```
How many threads would you like to use?
1
How many vertices would you like on this graph?
8

Here is the graph:
0 1 0 1 0 1 1 1
1 0 1 1 0 1 0 0
0 1 0 1 0 1 0 1
1 1 1 0 0 0 1 0
0 0 0 0 0 1 1 1
1 1 1 0 1 0 1 1
1 0 0 1 1 1 0 0
1 0 1 0 1 1 0 0

Colors of each vertex:
Vertex 0: Red
Vertex 1: Blue
Vertex 2: Red
Vertex 3: Green
Vertex 4: Red
Vertex 5: Green
Vertex 6: Blue
Vertex 7: Blue

Colors used: 3
Time to execute: 67060 ns
```

Figure 1. Example Single-threaded Run with 8 Vertices

b. My program showed effective speedup with strong scaling using up to 8 threads, then began to slow down slightly for additional threads. Of the thread amounts I tested, 8 appeared to be the most effective at reducing runtime for this problem size, resulting in a speedup of 4.288 compared to the single-threaded trial. The plot below shows the runtimes for different numbers of threads:

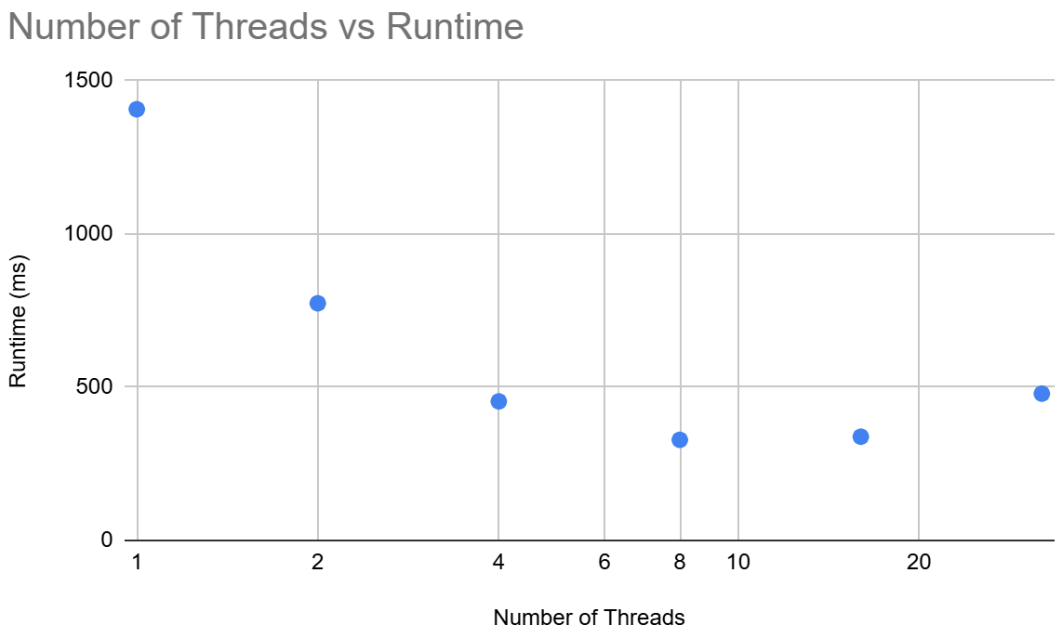

Figure 2. Number of Threads vs Runtime for 10,000 Vertices

At first glance, my program did not show as much promise for weak scaling, with throughput decreasing linearly with the number of threads used (figure 3), which did not make sense. However, this was when I defined throughput as the number of vertices in the graph divided by the runtime. This graph coloring algorithm has a time complexity of O(V^2+E), where V is the number of vertices, and E is the number of edges. In big O notation, this is an O(n^2) algorithm. Due to this, I needed to look at throughput as vertices^2 divided by the runtime. Once I applied this change, I saw a linear increase in throughput as I increased the number of threads up to about 8 threads, then the throughput curve flattened out (figure 4). With 32 threads, I was able to obtain a throughput of 400 vertices^2 per microsecond.

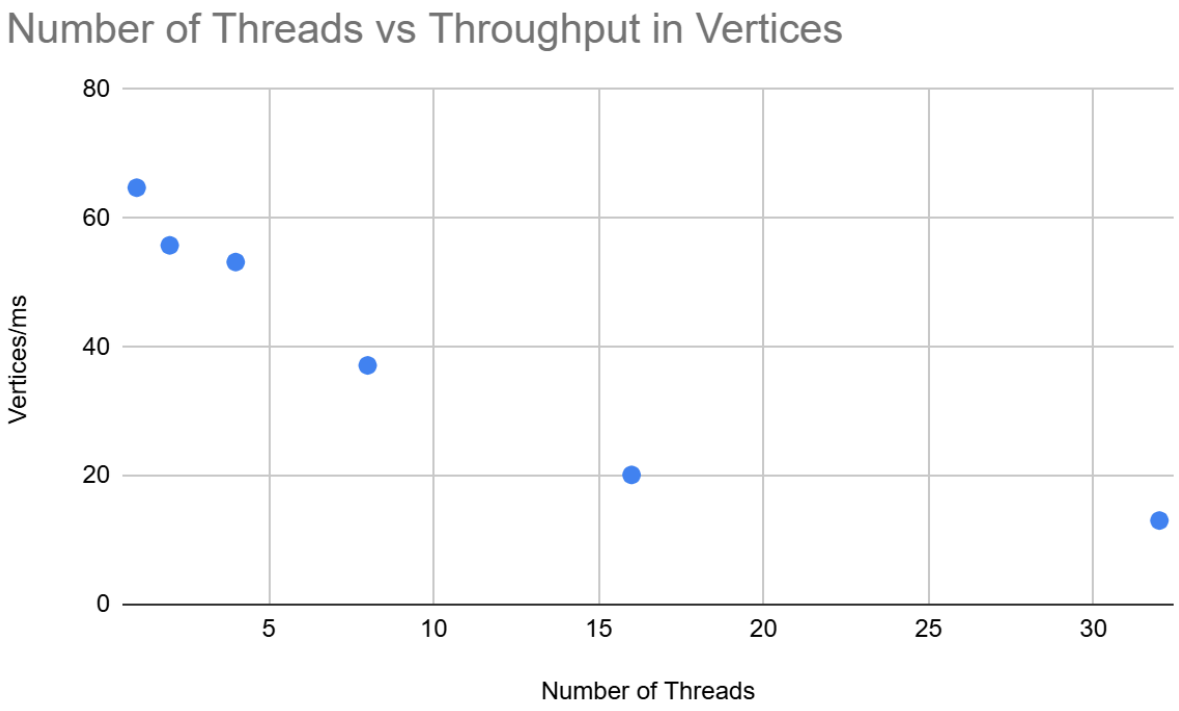

Figure 3. Number of Threads vs Throughput in Vertices

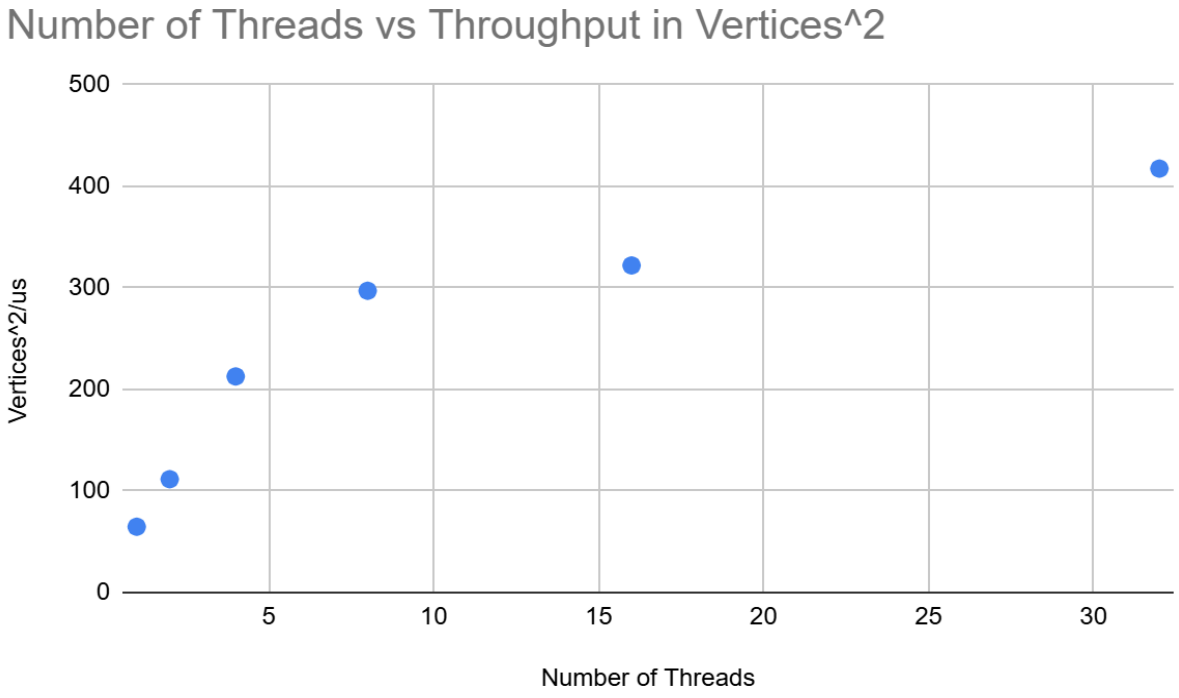

Figure 4. Number of Threads vs Throughput in Vertices^2