

PFP Project Proposal: Parallel Graph Coloring

Neil Daterao | UNI: nrd2140 | Fall 2026

1. Introduction

Graph coloring is the problem of assigning colors to vertices in a graph such that no two adjacent vertices share the same color. For this project, I'll focus on the decision problem: given a graph and an integer k , can we color the graph using at most k colors?

This problem is NP-complete for $k \geq 3$, making it an interesting candidate for parallelization. Since the algorithm involves exploring different possible color assignments, there are opportunities to split up the search and evaluate different branches in parallel.

My goal is to implement a sequential graph coloring algorithm in Haskell, parallelize parts of it, and compare the performance of both implementations. I'm mainly interested in seeing how much of a speedup I can achieve and understanding where parallelization actually helps.

2. Sequential Overview

I'll represent the graph using adjacency lists and use a recursive backtracking algorithm to find a valid coloring.

The algorithm will work as follows:

1. Pick an uncolored vertex.
2. Try assigning it a color that doesn't conflict with any of its neighbors.
3. Recursively continue coloring the remaining vertices.
4. If no valid color can be assigned, backtrack and try a different assignment.

If every vertex can be colored, the algorithm returns a valid coloring. Otherwise, it returns that no solution exists for the given k .

I'll start with a basic implementation and then introduce a simple heuristic where vertices with more neighbors are colored first. The idea is that coloring these vertices earlier should help identify invalid assignments sooner, reducing unnecessary searching.

To verify correctness, I'll test the implementation on graphs with known coloring properties, including complete graphs, paths, cycles, and bipartite graphs. I'll also check that any coloring returned by the algorithm is actually valid.

3. Parallel Approach

The main opportunity for parallelism comes from the backtracking search. When assigning a color to a vertex, each valid choice leads to a separate branch of the search tree. These branches can be explored independently, which makes them good candidates for parallel execution.

I'll use Haskell's *Control.Parallel.Strategies* library, specifically *rpar* and *rseq*, to evaluate different branches concurrently.

Rather than parallelizing the entire search tree, I'll initially focus on the first one or two levels of recursion. Beyond a certain depth, the algorithm will continue sequentially. This should help avoid creating too many small parallel tasks, where the overhead might outweigh the actual performance gains.

I'll experiment with different cutoff depths to see which approach works best. One thing I want to investigate is whether parallelism is actually beneficial when the search can terminate as soon as a valid coloring is found. In some cases, parallel branches might end up doing work that a sequential implementation would have avoided.

Both versions will use the same graph representation and vertex ordering so that the comparison is consistent.

4. Input Data and Performance Measurement

For testing, I'll use a combination of standard graph types and randomly generated graphs. Complete graphs, cycles, paths, and bipartite graphs will be useful for verifying correctness, while random graphs will let me experiment with different graph sizes and edge densities.

I'll use fixed random seeds so that I can run the same inputs across different configurations and get reproducible results. Graph generation and input parsing won't be included in the measured runtimes.

For the performance evaluation, I'll focus on:

1. **Execution time:** Compare sequential and parallel runtimes across different graph sizes, using the median of multiple runs.
2. **Speedup:** Calculate $S_2 = T_1 / T_2$ to measure how much faster the parallel implementation is.
3. **Efficiency:** Calculate S_2 / p to understand how effectively the available processors are being used.
4. **Scalability:** Test with 1, 2, 4, and 8 capabilities, depending on the available hardware.
5. **Cutoff depth:** Compare different levels of parallelization to see where the best performance occurs.

I'll compile the Haskell program and use GHC runtime options like `+RTS -N1`, `-N2`, `-N4`, and `-N8`, to control the number of capabilities.

I'll also use Amdahl's law to estimate how much of the program is effectively parallelizable and compare the expected speedup against the actual results.

Ultimately, I want to understand how well a backtracking algorithm like graph coloring benefits from parallel execution. The project will focus on getting a correct sequential implementation, introducing a relatively simple parallel strategy, and analyzing the tradeoffs between additional parallelism and the overhead it introduces.