M-Coloring Parallelization Project Proposal

Zhonghao Liu

November 23, 2025

1 Introduction to M-Coloring Algorithm and Problem Setup

1.1 The M-Coloring Problem

The m-coloring problem states that: given a graph $G = (V, E)$ with V vertices connected by edges E, determine whether it is possible to assign one of m colors to each vertex such that no two adjacent vertices share the same color. This problem serves as a great candidate for parallelization, especially when its standard solution involves backtracking depth-first search.

The worse-case complexity for the problem is $O(m^{|V|})$. In this project, I will discover speedup techniques through parallelization over a random graph input.

1.2 Problem Setup

I will generate a random graph using a Random Geometric Graph (RGG) model with the following technique:

- Scatter N points randomly in a unit square $[0, 1] \times [0, 1]$.
- Connect two points if their Euclidean distance $\leq$ a given radius r.

For this project, I will start with $n = 20$, $r = 0.2$, and $m = 4$. Then I will adjust m and r experimentally to ensure the graphs are solvable yet computationally demanding enough to measure parallel speedup. I will also reorder the colors in order to ensure as many nodes in the search tree are being visited as possible, as compared to finding a solution by only visiting the first branch towards the first leaf.

2 Speedup Techniques

2.1 Sequential Pre-Parallelization Optimization

- **Adjacency Representation:** Instead of keeping edges as a list of `[(Int, Int)]`, I will use `Data.IntMap` to provide $O(\log n)$ lookup for vertex neighbors.

- **Constraint Checking:** Instead of using a list to keep track of used colors for neighbors, I will use `Data.IntSet`, which reduced runtime by approximately 2x (according to the n-queens project provided in class).

2.2 Parallelization Strategies

2.2.1 Strategy A: Parallelize the Root Branches

I plan to first parallelize the branches of the first vertex by assigning each of the m colors to it.

```
mColoring n m graph = sum (firstVertexChoices `using` parList rseq)
  where
    firstVertexChoices = map (solveRest graph ...) [1..m]
```

Expected Outcome: This approach generates only m sparks. For small m (e.g., 4), this will likely result in poor load balancing and "plateaus" on my 16-thread machine.

2.2.2 Strategy B: Parallelize deeper levels

To address the load balancing issues of Strategy A and the irregular nature of random graphs, I will further parallelize deeper branches of the search tree in order to generate more sparks. This means for each neighbor of the root vertex, I will parallelize assigning each of the $m-1$ colors.

```
import Control.Parallel.Strategies (using, parList, rseq)

mColoring :: Int -> Graph -> Int
mColoring m graph = count 0 emptyColoring
  where
    numVertices = length (vertices graph)
    -- Depth 2 or 3 ensures enough sparks are generated
    -- to keep 16 threads busy without flooding the runtime.
    parallelDepth = 2

    count :: Vertex -> Coloring -> Int
    count v currentColoring
      | v == numVertices = 1
      | v < parallelDepth =
          sum (map recursiveStep validColors `using` parList rseq)
      | otherwise =
          sum (map recursiveStep validColors) -- Sequential cutoff
      where
        validColors = filter (\c -> isSafe v c graph currentColoring) [1..m]
        recursiveStep c = count (v + 1) (insertColor v c currentColoring)
```

Expected Outcome: By tuning `parallelDepth`, I aim to maximize the number of "converted" sparks while minimizing "fizzled" sparks caused by branches that fail fast.

3 Evaluation Techniques

The programs will be run on a PC with a CPU of 8 physical cores and 16 hardware threads (SMT).

3.1 Runtime and Scalability Analysis

I will evaluate the strong scaling of the algorithm by running the best parallel implementation against the sequential baseline.

- **Metric:** Speedup $S_N = T_1/T_N$, where T_1 is the runtime on a single thread and T_N is the runtime on N threads.
- **Target:** I will plot Speedup vs. Threads ($N = 1$ to 16). Ideally, this should approach the linear limit derived from Amdahl's Law.
- **Observation:** I expect speedup to plateau after 8 threads (physical cores) due to resource contention in SMT, as seen in standard parallel benchmarks.

3.2 Threadscope and Spark Statistics

To diagnose the efficiency of the parallel strategy, I will use GHC's event logging (`+RTS -l -s`).

- **Spark Conversion Rate:** I will measure the ratio of *converted* to *fizzled* sparks. In random graphs, invalid colorings are detected quickly, leading to high fizzle rates. I will use these stats to tune the `parallelDepth`.
- **Load Balancing:** Using Threadscope, I will visually inspect the execution timeline for "staircase" patterns or gaps in the green activity bars, which indicate that threads are starving for work.

3.3 Garbage Collection Tuning

Since graph coloring involves creating many partial immutable state objects (`IntSet`), GC pressure will be high.

- **Experiment:** I will compare runtimes using default GC settings versus increased Allocation Area sizes (e.g., `-A16M`, `-A64M`).
- **Expectation:** While larger nurseries reduce GC frequency, prior reports suggest they may degrade cache performance. I will report the "Mutator Time" vs. "GC Time" to determine the optimal setting.

4 References

- Edwards, Stephen A. "Parallel N-Queens in Haskell." Columbia University, 2025.
- "M-Coloring problem". `https://www.geeksforgeeks.org/dsa/m-coloring-problem/`