Evaluating Parallel Graph Coloring for Conflict-Free Graph Processing

Amin Hong(minhong) Jinrae Kim (jinraek)

December 1, 2025

URL: https://aminh0.github.io/15-618-project/

1 Summary

- We are going to evaluate how different graph coloring algorithms impact the performance of parallel graph processing.
- depending on the number of colors we use, we will compare the runtime and find the optimal number of colors.
- Using simple parallel kernels such as Laplacian smoothing and community-detection-like neighbor aggregations, we will measure and compare runtime, scaling behavior, and synchronization overhead across multiple coloring strategies on a multicore CPU system.

2 Background

Graph analytics often involve computations where each vertex updates its state using information from its neighbors (e.g., Laplacian smoothing, PageRank-like updates, community detection, or general neighbor-sum kernels). However, running these updates in parallel can introduce race conditions or require heavy synchronization, since adjacent vertices may attempt to read/write shared data simultaneously[4]. These dependencies reduce available parallelism and can significantly degrade performance due to locking, atomic operations, or barrier-heavy execution.

To mitigate these conflicts, we plan to use graph coloring as a preprocessing step. A valid vertex coloring partitions the graph into a sequence of color batches such that no two vertices in the same batch are adjacent. This property allows each batch to be processed in parallel without any conflicts. Many well-known coloring algorithms (e.g., greedy coloring, largest-first, DSATUR, Luby-Jones-Plassmann) produce different numbers of colors and exhibit different runtime and parallelism characteristics[1, 3]. These differences lead to distinct performance tradeoffs when the coloring is used to construct a parallel schedule.

After computing a coloring using several existing algorithms, we will evaluate their effectiveness by running a representative graph computation over the resulting color batches. A typical example is Laplacian smoothing, where each vertex repeatedly replaces its value with the average of its neighbors' values. Since vertices with the same color never share an edge, each color batch update step is naturally parallelizable and requires no synchronization.

We will compare multiple coloring algorithms by measuring (1) their coloring runtime, (2) the number of colors they produce (which affects the number of parallel phases), and (3) the performance of the resulting parallel computation. This approach highlights how graph coloring can be used to restructure irregular computations into conflict-free parallel schedules and exposes tradeoffs between coloring overhead, batch granularity, and computational throughput.

Below is a simplified version of the parallel schedule:

```
for color = 1..C:
    parallel_for (v in vertices_with_color[color]):
        out[v] = f( neighbors(v) )
    barrier
```

This pattern is widely applicable because it avoids all race conditions and ensures deterministic updates without requiring locks or atomics. By analyzing the interplay between coloring strategy and overall computation time, we aim to understand how graph coloring affects the performance of irregular parallel workloads[2].

3 Challenges

Our project faces three major challenges in applying graph coloring to improve the performance of parallel graph algorithms:

- **Irregular Workload and Dependency Structure.** Real-world graphs exhibit highly irregular degree distributions and non-uniform connectivity patterns. This makes it difficult to predict workload balance across color groups: some colors may contain many vertices while others contain very few. Such imbalance can limit parallel speedup, especially when each color batch requires a global synchronization barrier. Furthermore, high-degree vertices increase the likelihood of conflicts during coloring, affecting the quality of the schedule.
- **Memory Access Locality and Communication Overhead.** Graph computations such as Laplacian smoothing require repeatedly accessing each vertex's neighbors. This results in irregular, pointer-chasing memory accesses with poor spatial locality. When vertices mapped to different threads or NUMA domains share edges, the cost of cross-thread or cross-core communication becomes significant. Coloring attempts to reduce such conflicts, but does not completely eliminate locality issues, and the benefit depends heavily on the quality of the coloring algorithm.
- **Mapping the Workload to the Hardware Efficiently.** Even with a valid coloring, mapping each color class to threads is non-trivial. Some color classes may be extremely small, causing underutilization of hardware parallelism, while others may be too large, causing contention on shared resources. Choosing between dynamic and static scheduling impacts both load balance and synchronization cost. Additionally, to obtain meaningful runtime (e.g., around 10 seconds), we must carefully tune dataset size, edge density, and the number of smoothing iterations to expose computation-to-communication ratios that stress the parallel system.

4 Resources

- **Graph Coloring Libraries (Baseline)** We will use existing graph coloring libraries such as the Boost Graph Library (BGL) or networkx (Python) to generate the baseline colorings. These libraries are used only to obtain initial colorings for comparison; our project will implement the parallel scheduling and computation phases from scratch.
- **Custom Coloring Implementations** We will additionally implement several greedy or heuristic graph coloring algorithms ourselves (e.g., Largest-Degree-First, Smallest-Last, random greedy) to compare scheduling quality and runtime. This supports our goal of evaluating how different coloring strategies influence overall parallel performance.
- **Dataset Generation Tools** We will generate synthetic graph datasets of varying density and size (e.g., 100 nodes, 500 nodes; easy/medium/heavy based on edge density) using small Python scripts or C++ utilities. These datasets will allow controlled experiments on how graph structure affects scheduling and parallel performance[4].
- **Reference Lectures** We will reference lecture materials from 15-618 related to Laplacian smoothing, graph algorithms, and coloring heuristics.

5 Goals and Deliverables

- **50% Completion:** Graph dataset generator implemented (easy/medium/heavy graph densities for 100 and 500 nodes). Baseline graph coloring completed using existing libraries (e.g., BGL, networkx). Sequential Laplacian smoothing implemented to establish a baseline computation kernel.
- **75% Completion:** Custom greedy coloring algorithms are implemented (Largest-Degree-First, Smallest-Last, Random-Greedy). Color-batch parallel schedule implemented using OpenMP. Initial parallel Laplacian smoothing integrated with coloring-based scheduling.

- **100% Completion:** Full evaluation of multiple coloring algorithms on all datasets. Performance comparison across different colorings (runtime, speedup, conflict-free execution). Detailed analysis of how graph structure (density, degree distribution) affects parallel speedup.
- **125% Completion (Stretch Goal):** Introduce "fixed palette size" experiments: Force color counts (e.g., 100, 80, 60, 40, 20 colors) and compare runtime differences. Analyze how reducing available colors affects parallelism and workload balance.
- **150% Completion (Stretch Goal):** Estimate or search for the "optimal" number of colors that balances parallelism and batch size. Implement a simple heuristic or adaptive coloring refinement to target near-optimal color counts. Conduct expanded evaluations on larger datasets or more smoothing iterations.

6 Platform Choice

- **Parallel Computing Platforms** We will conduct all performance evaluations on the CPUs available in the CMU GHC machines and the PSC Bridges-2 cluster. These systems provide multi-core processors (typically 16 hardware threads), which are sufficient to evaluate the scalability of our parallel color-based scheduling algorithm.
- **Programming Environment** Our implementation will be written in C++ using OpenMP for multi-threaded parallelism. OpenMP provides an easy-to-use model for parallel loops (e.g., `pragma omp parallel for`) and barriers, which aligns well with our color-batch execution strategy.

7 Schedule and Work Partitioning

Our project timeline spans three weeks. We divide the development process into incremental milestones (50%, 75%, 100%, 125%, 150% completion) following the course guidelines.

- **Week 1 (50% Completion):**
  Implement basic graph generation utilities (easy/medium/heavy datasets). Integrate baseline graph coloring methods (Boost, NetworkX, or simple greedy). Develop a sequential version of the Laplacian smoothing computation. Verify correctness of the baseline sequential implementation.
- **Week 2 (75% Completion):**
  Implement the color-based parallel scheduling framework using OpenMP. Run parallel Laplacian smoothing using precomputed color partitions. Measure preliminary performance results (speedup vs. number of threads). Prepare scripts for automated experiments. Prepare for the milestone report.
- **Week 3 (100% Completion):**
  Complete evaluation on all datasets (100-node / 500-node, easy/medium/heavy). Analyze the impact of different graph coloring algorithms on parallel performance. Finalize plots showing runtime vs. colors, runtime vs. graph density, and scalability. Prepare for the final report.
- **Extension 1 — 125% Completion:**
  Introduce "color count sweeps" (reducing number of available colors) and measure how performance changes under artificially constrained palettes. Evaluate overhead such as increased conflicts or synchronization cost.
- **Extension 2 — 150% Completion:**
  Implement an additional optimization or heuristic (e.g., recoloring, color merging). Search for the "optimal" number of colors that maximizes parallel performance. Compare the optimized schedule with all prior baselines.
- **Work Partitioning:**
  - **Amin Hong:** Parallel implementation, evaluation pipeline, color sweep experiments.
  - **Jinrae Kim:** Dataset design, graph-coloring algorithm implementation, analysis scripts.

References

[1] Brélaz, D. New methods to color the vertices of a graph. *Communications of the ACM 22*, 4 (1979), 251–256.

[2] Gebremedhin, A., and Manne, F. Scalable parallel graph coloring algorithms. In *International Conference on High Performance Computing* (2000), Springer, pp. 241–256.

[3] Jones, M., and Plassmann, P. A parallel graph coloring heuristic. In *SIAM Journal on Scientific Computing* (1993).

[4] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 1 (1998), 359–392.