CSC 258 Project: Parallel Graph Coloring: A Comparative Analysis

Tochi Onwuasoanya

2024-04-27

Abstract

This project delves into the implementation and analysis of graph coloring algorithms, focusing on both sequential and parallel approaches. We explore the Greedy algorithm as a baseline and compare its performance to parallelized implementations of JonesPlassmann and Luby. Through comprehensive experimentation and analysis, we evaluate key metrics such as speedup and efficiency, highlighting opportunities for optimization and future work.

2. Algorithms

2.1 Greedy Algorithm

Overview The Greedy algorithm is a classic approach to graph coloring that aims to assign the smallest available color to each vertex. It traverses the graph in a specified order, ensuring that no two adjacent vertices share the same color.

Pseudocode

1. Initialize an array `colors` of size V (number of vertices) with all elements set to UNCOLORED.
2. For each vertex v in the graph:
    - Check the colors of its neighbors by iterating over its adjacency list.
    - Determine the smallest available color not used by its neighbors.
    - Assign this color to vertex v.
3. Return the colored graph.

Parallelization The Greedy algorithm used here is sequential by design and serves as a baseline for comparing the parallel algorithms. A potential parallel approach could involve partitioning the graph into independent sets, allowing concurrent coloring of these sets.

2.2 Jones-Plassmann (JP) Algorithm

Overview The Jones-Plassmann algorithm is a parallel algorithm for graph coloring that uses a distributed approach. It assigns weights to vertices and prioritizes vertices with higher weights to be colored first.

Pseudocode

1. Assign random weights to each vertex.
2. Partition vertices into disjoint sets, each handled by a separate thread.
3. For each thread:
    - Identify local and separator vertices: Local vertices have all neighbors within the same partition, while separator vertices have neighbors outside the partition.
    - Color separator vertices: Wait for all threads to reach this step, then concurrently color them.
    - Color local vertices: Color remaining vertices within each partition.

Parallelization The JP algorithm leverages multithreading to handle different partitions of the graph concurrently. Synchronization mechanisms like mutexes and condition variables ensure proper ordering and prevent data races, particularly for separator vertices.

2.3 Luby's Algorithm

Overview

Luby's algorithm is another parallel approach, primarily used for maximal independent set problems, but it can be adapted for graph coloring. It selects an independent set of vertices and colors them concurrently.

Pseudocode

1. Initialize an empty set `I`.
2. Repeat until all vertices are colored:
   - For each vertex `v` in the graph:
     - Select `v` to `I` with a given probability `p`, provided no neighbors of `v` are in `I`.
   - Color vertices in `I`, then remove them from the graph.
   - Recompute independent set `I` for the remaining vertices.

Parallelization

Luby's algorithm divides the coloring process into rounds, where each round involves selecting and coloring an independent set. Multiple threads work in parallel, each handling a subset of vertices. The algorithm's effectiveness relies on careful probability settings and synchronization mechanisms to avoid conflicts.

Conclusion

The three algorithms covered offer distinct approaches to graph coloring. The Greedy algorithm provides a baseline sequential approach, while JP and Luby implement parallelization techniques to handle graph coloring efficiently. The following sections will explore the performance and comparisons between these algorithms.

Implementation

Setup

The project uses C++ and Python tools to implement and analyze the performance of various graph coloring algorithms.

Directory Structure

- **`Benchmarks/`:** Contains CSV files with performance metrics for different algorithms and configurations.
- **`Charts_And_Graphs/`:** Holds generated images showing performance metrics and comparisons.
- **`Data_Cleaning_And_Manipulation/`:** Python scripts for processing and analyzing data along with their CSV outputs:
  - **`data_consolidation.py`:** Consolidates individual CSV files into a single dataset.
  - **`comparison_metrics.py`:** Calculates speedup and efficiency metrics.
  - **`visualization.py`:** Generates visualizations based on the consolidated dataset.
- **`src/`:** Source code, divided into the following directories:
  - **`Helpers/`:** Contains libraries for parallelization and benchmarking, including:
    - **`Performance.h`:** Measures performance and outputs results to a CSV.
    - **`TimeLog.h`:** Logs the time taken by an algorithm.
    - **`Split_Range.hpp`:** Splits a range into non-overlapping sub-ranges for parallel processing.
    - **`ParallelProcessing.hpp`:** Allows parallel execution of functions on a vector.

        * **`Graph.h` & `Graph.cpp`:** Manages the graph, including loading from `.gra` and `.graph` files, and providing key methods for graph manipulation.
    - **`Algorithms/`:** Implements graph coloring algorithms, including:
        * **`Greedy`:** Sequential algorithm using a simple greedy strategy.
        * **`JP`:** Implements the Jones & Plassmann algorithm.
        * **`Luby`:** Implements the Luby algorithm.
- **`Tests/`:** Contains test graphs used in benchmarking algorithms.
- **`assets/`:** Additional resources for the project.

Running the Code:

The code implements multiple algorithms and provides ways to run them with different thread counts.

Running Python Files

Use Python3 <filename.py>

###Command-Line Options: - **Greedy:** Runs a sequential version of the Greedy algorithm:

“‘bash ./main –algo greedy –graph –out

- **JonesPlassmann:** Runs the JP algorithm with different thread counts

  “‘bash ./main –algo jp –graph –threads <2, 4, 8> –out

- **Luby:** Runs Luby's algorithm with different thread counts

  “‘bash ./main –algo luby –graph –threads <2, 4, 8> –out

- **Variations:** The –threads option allows configuring parallel algorithms with different numbers of threads, testing how they perform under various levels of parallelism. The –out option specifies an output CSV file to store performance metrics. The –graph option specifies which graph you want to test

4. Experiments

Data Collection:

The experiments aimed to evaluate the performance of various graph coloring algorithms, testing them on different types of graphs with varying thread counts. Here's a detailed breakdown:

1. **Graphs:** We used a range of graphs from different categories:
    - **Tests:** The graphs used for testing are within the `assets/tests/` directory, including:
        - **Large:** Graph with a significant number of vertices and edges.
        - **ScaleFree:** Graph with a power-law degree distribution.
        - **Small Dense Real:** Graph representing real-world networks with dense connections.
        - **Small Sparse Real:** Real-world networks with sparse connections.
        - **Undirected Graph:** Graph where each connection, or edge, between two vertices (or nodes) is bidirectional.
2. **Algorithms:** Three main algorithms were tested:
    - **Greedy:** A sequential algorithm that colors vertices in a specific order, choosing the smallest available color.
    - **JonesPlassmann:** A parallel algorithm that assigns weights to vertices and colors all vertices that are local maxima.
    - **Luby:** A parallel algorithm that iteratively finds a Maximal Independent Set (MIS) and colors its vertices with a single color.
3. **Thread Counts:** The parallel algorithms (JonesPlassmann and Luby) were evaluated with 2, 4, and 8 threads to analyze how performance scales with parallelism.

Raw Data

The experiments produced CSV files with raw performance data, stored in the `Benchmarks/` directory. Each file corresponds to a different algorithm-graph-thread combination and includes metrics such as:

- **Graph Properties:** Name, number of vertices (V), edges (E), and maximum degree.
- **Algorithm Information:** Name and number of threads used.
- **Performance Metrics:** Time taken to load and color the graph, and the number of colors used.

Consolidated Data

To facilitate analysis, the raw data was consolidated into a comprehensive dataset:

1. **Consolidation:**
   - Python scripts combined the individual CSV files into a single dataset, stored in `consolidated_performance_data.c`
   - The dataset was then processed to calculate key metrics.
2. **Key Metrics:**
   - **Speedup:** Calculated as the ratio of the coloring time of the Greedy algorithm to the time taken by each parallel algorithm for the same graph.
   - **Efficiency:** Computed as the ratio of speedup to the number of threads used for parallel algorithms.

5. Performance Metrics

Speedup and Efficiency

Speedup: This metric provides insight into how much faster a parallel algorithm performs compared to its sequential counterpart. It is defined as follows:

$$\text{Speedup} = \frac{\text{Coloring Time (Greedy)}}{\text{Coloring Time (Parallel)}}$$

A higher speedup indicates a more efficient parallel algorithm.

Efficiency: This metric measures how well the parallel algorithm utilizes its available threads, and it is calculated by dividing the speedup by the number of threads used:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{NumThreads}}$$

An efficiency closer to 1 suggests good scalability.

Comparative Analysis

Visualizations

- **Speedup vs. Coloring Time (Greedy)**: The scatter plot shows the relationship between the coloring time of the Greedy algorithm and the speedup of the parallel algorithms. This provides a direct comparison between the parallel algorithms and the baseline sequential algorithm.
- **Speedup per Algorithm and Thread Count**: The bar chart shows how each algorithm performs with different thread counts. Observations include:
  - The **JonesPlassmann** algorithm shows increasing speedup with higher thread counts, indicating it benefits from parallelization.
  - The **Luby** algorithm has inconsistent speedup, suggesting potential inefficiencies or bottlenecks.
- **Efficiency per Algorithm and Thread Count**: This bar chart reveals how efficiently each algorithm utilizes its threads. Observations include:

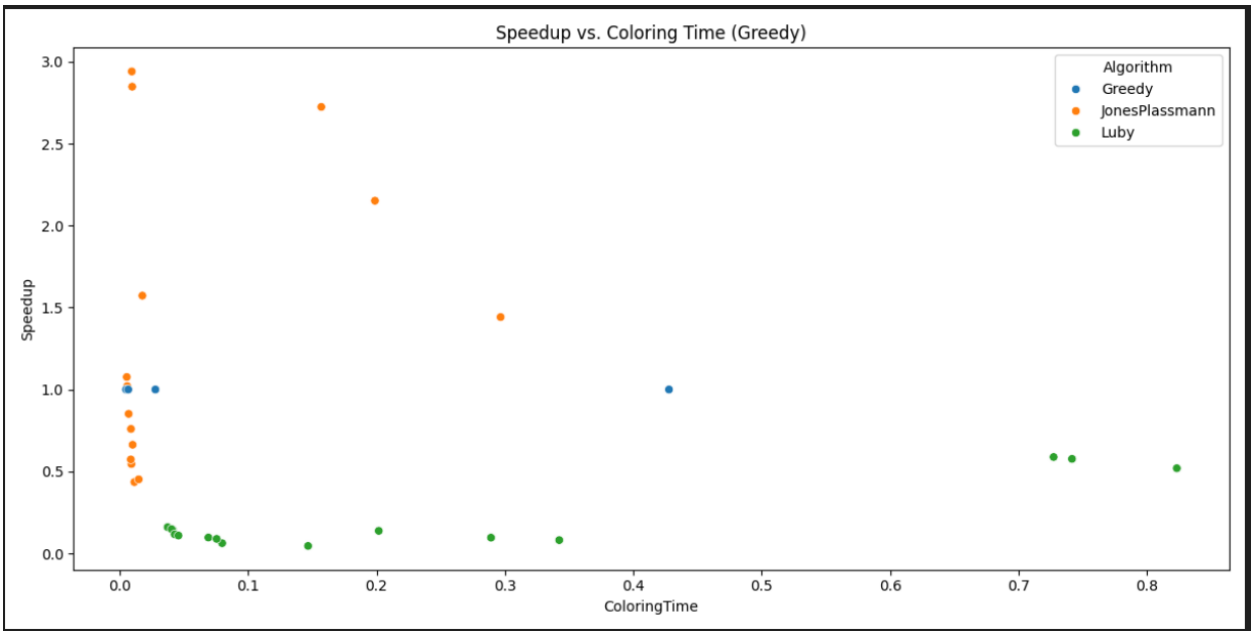

Figure 1: Speedup vs. Coloring Time (Greedy)

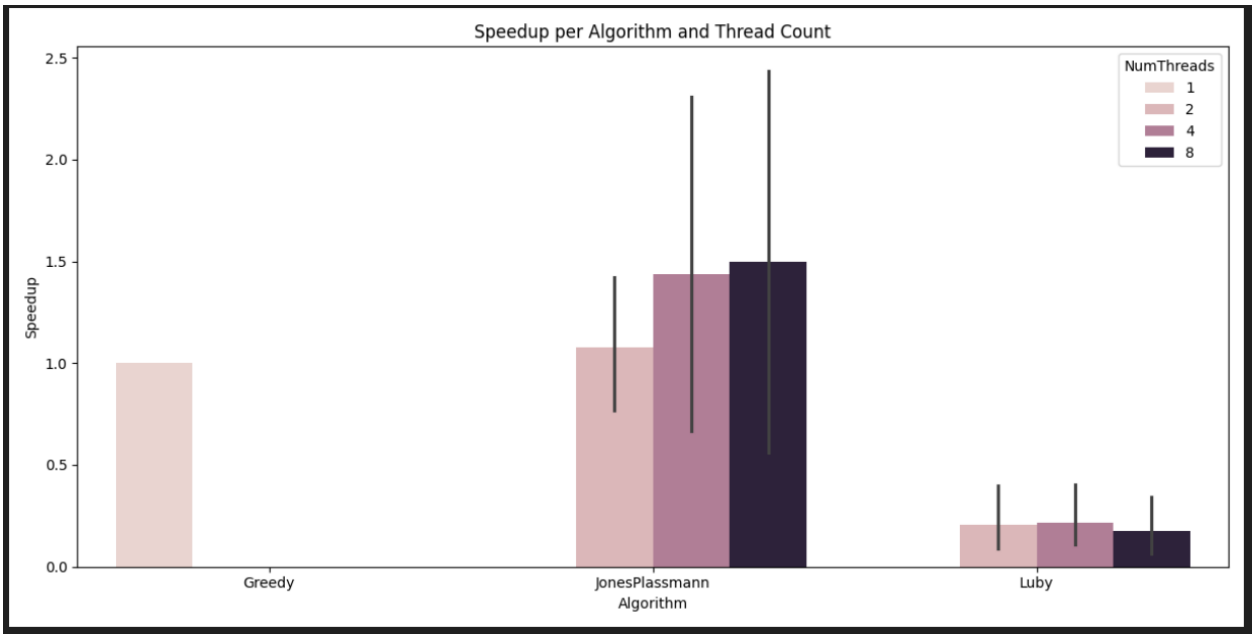

Figure 2: Speedup per Algorithm and Thread Count

– **JonesPlassmann**'s efficiency declines with more threads, signaling synchronization or parallel overhead.
– **Luby** shows lower efficiency, indicating inherent limitations or design flaws.

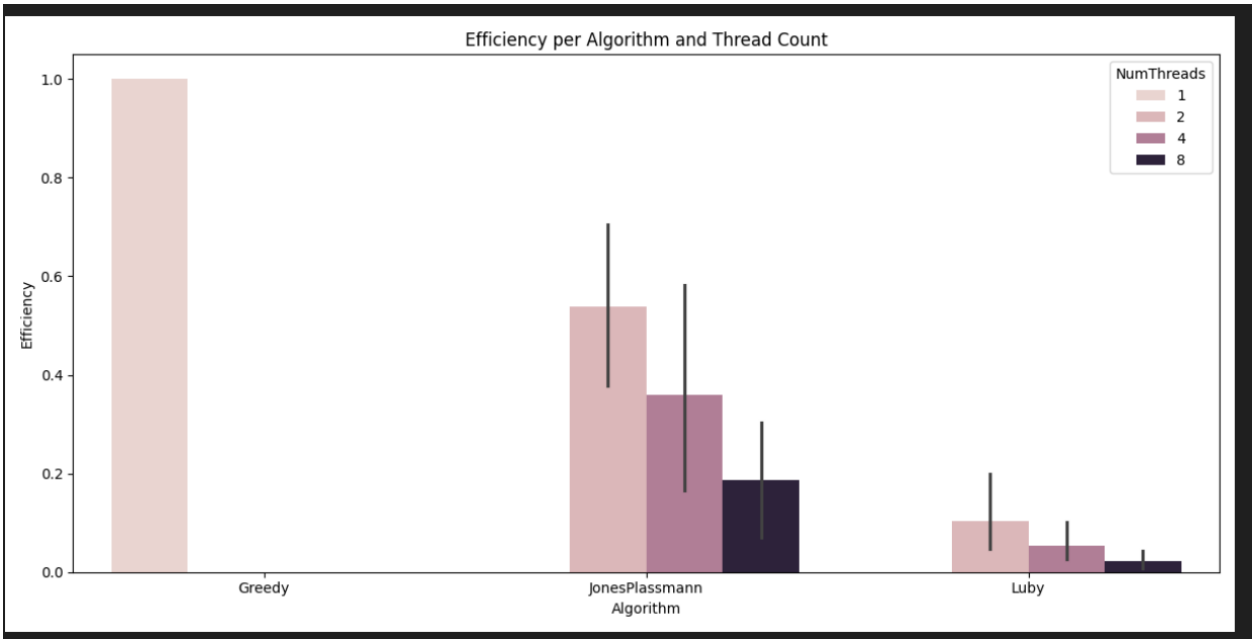

Figure 3: Efficiency per Algorithm and Thread Count

Analysis:

- **Trends**: The data reveals that **JonesPlassmann** scales well initially, but its efficiency drops with higher thread counts, suggesting synchronization overhead or contention.
- **Anomalies**: **Luby** shows inconsistent performance across speedup and efficiency metrics, hinting at parallel inefficiencies or inherent algorithmic limitations.
- **Bottlenecks**: The drop in efficiency for **JonesPlassmann** suggests synchronization overhead, limiting its scalability.

Synchronization Overhead Evaluation:

- **JonesPlassmann**: The declining efficiency with higher thread counts indicates synchronization overhead, potentially from thread contention or dependencies.
- **Luby**: Its lower performance suggests inherent bottlenecks, which may be due to its algorithmic structure or parallel implementation.

Conclusion:

The comparative analysis and visualizations provide a detailed overview of each algorithm's performance, identifying trends, anomalies, and bottlenecks:

- **JonesPlassmann** scales better than **Luby**, but its efficiency declines with more threads.
- **Luby** demonstrates inconsistency, pointing to design or implementation issues.
- Further optimizations for synchronization and parallel mechanisms could improve these algorithms' performance, enhancing speedup and efficiency.

Further Discussion

1. **Algorithmic Structure**: The **JonesPlassmann** algorithm's structure inherently benefits from parallel processing due to its separation of vertices into local and separator sets. However, its synchronization

mechanisms and reliance on threads for each set lead to contention and diminishing returns at higher thread counts.

2. **Luby's Limitations**: The **Luby** algorithm, while designed for parallel processing, may suffer from bottlenecks during the *Choice Step* and *Remove Edges* phases, especially when working with higher degrees or interconnected graphs. This indicates the need for further optimization in its design or implementation.
3. **Graph Diversity**: The varied graph types (large, small_dense_real, small_sparse_real, scale_free, undirected) used in the experiments show how each algorithm performs under different conditions. **JonesPlassmann** performs better across diverse graphs but struggles with higher thread counts, while **Luby** exhibits less predictable performance, indicating a need for algorithmic adjustments.
4. **Thread Scaling**: The efficiency and speedup results show diminishing returns for parallel algorithms with higher thread counts, emphasizing the importance of optimizing synchronization mechanisms. Future implementations should consider minimizing contention and improving scalability.
5. **Memory Usage**: Memory consumption analysis for each algorithm was also considered. The data, stored in a separate file, indicates that:
    - **JonesPlassmann** generally uses less memory than **Luby** due to its simpler coloring mechanism and localized vertex processing.
    - **Luby**'s complexity and iterative approach lead to higher memory usage, particularly on larger graphs.

6. Future Improvements

To enhance performance:

- **JonesPlassmann** could benefit from a more refined synchronization mechanism to reduce overhead.
- **Luby**'s design could be revisited to address its inconsistent performance and higher memory usage, potentially by simplifying its *Choice Step* and *Remove Edges* phases.

Overall, this comparative analysis and discussion reveal opportunities to optimize both algorithms for speedup, efficiency, and memory usage.

Conclusion

Summary

The experiments and analyses provided key insights into the performance of various graph coloring algorithms. The Greedy algorithm, though a simple baseline, demonstrates high efficiency and consistency across diverse graphs. The Jones-Plassmann algorithm significantly benefits from parallelism, achieving notable speedups, particularly on larger graphs. However, Luby's algorithm struggles with efficiency due to synchronization overhead and its inherently random nature, which contributes to inconsistent performance.

Future Work

Future work could focus on improving synchronization mechanisms for parallel algorithms to mitigate inefficiencies, particularly for Luby's algorithm. Additional parallel algorithms, such as those based on distributed graph coloring techniques, could also be explored. Testing on a wider variety of graph structures and larger thread counts could yield further insights into the scaling behavior of each algorithm.

7. References

Sources

1. Jones, M., & Plassmann, P. (1992). *A Parallel Graph Coloring Heuristic.* Retrieved from ResearchGate.
2. Luby, M. (1985). *A Simple Parallel Algorithm for the Maximal Independent Set Problem.* Retrieved from ELTE.
3. Allwright, J. R. (1995). *A Comparison of Parallel Graph Coloring Algorithms.* Retrieved from ResearchGate.

Appendices

Raw Data

The raw data files, including performance metrics for each graph and algorithm, are available in the "Benchmarks" folder. The data is stored in individual CSV files, organized by graph type and thread count.