ABSTRACT

**FINAL REPORT:**

**SCALABLE GRAPH COLORING ALGORITHMS**

**CENG577 PARALLEL COMPUTING**

**FALL 2024-2025**

İmre KOSDİK

Yağızcan PANÇAK

**1 ABSTRACT**

Graph coloring involves assigning labels, known as colors, to the vertices of a graph such that no two adjacent vertices share the same color, using the smallest number of colors possible. In parallel computing applications, graph coloring helps identify independent tasks that can be executed simultaneously. Graph coloring is an NP-Hard problem relying on heuristic methods to solve it. The first algorithm in [1] addresses this issue and offers almost a linear speedup comparing sequential graph coloring. This work focuses on implementing this algorithm, which aims to enhance the efficiency of the coloring process, especially in parallel computing environments. Examining the algorithm's performance across different graphs will provide valuable insights into its efficiency and scalability. This evaluation will also help assess the practical applicability of the algorithm for solving real-world graph coloring problems.

**2 INTRODUCTION**

Graph coloring involves assigning "colors" to the vertices of a graph such that no two adjacent vertices share the same color. The objective is to utilize the smallest possible number of colors. This concept is essential for solving scheduling and team-building challenges, identifying short circuits in printed circuit designs, and serving as a preprocessing step to accelerate the computation of Jacobian and Hessian matrices. The minimum number of colors required to solve the graph coloring problem is the chromatic number of a graph.

Determining the chromatic number is classified as an NP-Hard problem, which prompts the use of heuristic methods for resolution. The algorithms outlined in section 3 of [1] adopt greedy techniques and provide nearly linear speedup. These algorithms implement block partitioning by dividing the vertex set of a graph into equal-sized blocks, with each block assigned to a different processing unit for coloring. Given that neighboring vertices may reside in separate blocks, there is a risk that processes may inadvertently assign the same color to adjacent vertices. The algorithm effectively resolves these conflicts by sequentially coloring these vertices.

In this project, we aim to implement the first algorithm and evaluate the performance of our implementation across various sets of sparse graphs.

**3 MOTIVATION**

The motivation for this project originates from the distinct challenges presented by graph coloring in parallel processing environments. Due to its NP-hard nature, graph coloring is inherently complex, making it computationally demanding to find the optimal solution, particularly for large graphs that reflect real-world systems. Sparse graphs appear in various applications such as social network analysis, circuit design, and scientific computation and introduce additional challenges because of their irregular structures, complicating partitioning and parallelization. It is crucial to strike a balance between the accuracy of the solution (in terms of minimizing the number of colors used) and the algorithm's efficiency (regarding time and computational resources) to achieve effective parallel graph coloring. The research conducted by Gebremedhin and Manne on scalable parallel graph coloring algorithms presents a promising strategy for tackling these challenges. By implementing and testing their algorithm across a range of sparse graphs, we can uncover its practical advantages and limitations and gain insights into its scalability concerning graph size and density. This project also aims to investigate how variations in graph structure influence the algorithm's performance and effectiveness, thereby contributing to the broader field of scalable graph algorithms.

**4 BACKGROUND AND RELATED WORK**

Efficient graph coloring plays a vital role in parallel computing: identifying independent tasks, and enabling them to execute concurrently without conflicts. Various algorithms have been developed for this purpose in parallel environments, with Gebremedhin and Manne’s first algorithm especially noteworthy. One of the main advantages of choosing this algorithm over other existing methods is its effective color minimization. Traditional algorithms, such as basic greedy methods, often fail to prioritize reducing the number of colors in parallel settings, resulting in increased computational overhead and resource consumption. In contrast, the first algorithm refines the initial greedy approach by utilizing fewer colors and minimizing conflicts that arise from block partition boundaries. This refinement makes it especially well-suited for sparse graphs, frequently found in real-world scenarios like social networks and scientific computations, where reducing color counts directly impacts performance and resource requirements.

Gebremedhin and Manne’s research proposes scalable parallel graph coloring algorithms focusing on block partitioning, which divides the vertex set of the graph into equal-sized blocks to facilitate parallel computation. They introduced two algorithms that take advantage of this partitioning technique, achieving significant speedups by processing the blocks concurrently. The first algorithm assigns colors to vertices using a greedy approach, while the second algorithm is a refinement that requires fewer colors, enhancing the efficiency of the coloring process. These methods exhibit near-linear scalability, making them suitable for large, sparse graphs encountered in real-world applications. This project builds upon their work by implementing the first algorithm to explore its applicability and performance across various sparse graph structures.

**4.1 Block Partition-Based Graph Coloring Algorithm**

The algorithm consists of three phases. In the first phase, we partition the vertex set into p equal-sized blocks, where p represents the number of processors. Each process colors vertices in parallel, and each step concludes with a barrier synchronization. A vertex receives its color based on the colors of its colored neighbors. These neighbors can belong to the same block as the vertex or to a different block, allowing adjacent vertices in distinct blocks to be assigned the same color. This coloring method is known as pseudo-coloring.

In the second phase, we check in parallel whether each vertex has a valid coloring by comparing its color with those of its neighbors that were colored during the same parallel step in the first phase. If an invalid coloring is detected, we store one of the vertices involved in the conflicting edge in a table. In the final phase, we sequentially re-color each vertex listed in the table to resolve any conflicts. The pseudocode for the algorithm is below.

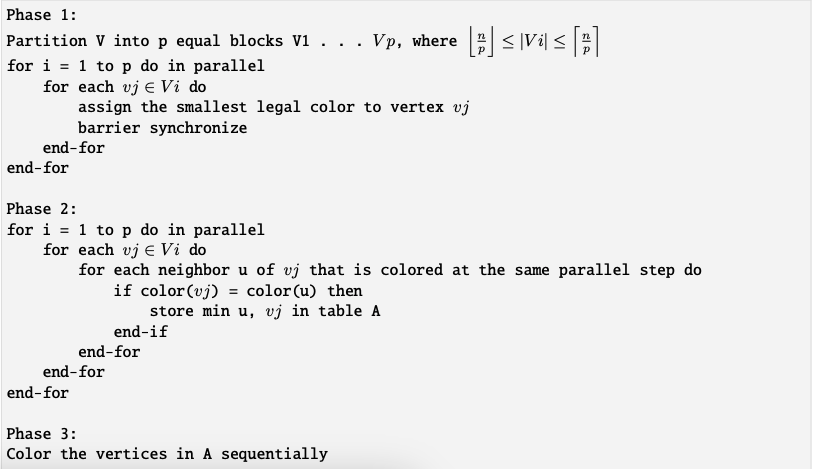
****

Figure 1: Pseudo code of Block Partition-Based Graph Coloring Algorithm

**5 IMPLEMENTATION AND METHODOLOGY**

We chose the C programming language and the MPI library as our development environment. Our application processes undirected random graphs as input, which we obtained from the [SuiteSparse Matrix Collection](http://sparse.tamu.edu/). The input format specifies the number of rows, the number of columns, and the total number of non-zero elements in the sparse matrix on the first line. The subsequent lines indicate the row and column indices of the non-zero elements. In our implementation, the actual values of the non-zero elements are irrelevant, as our goal is to represent the sparse matrix as a graph.

Sparse matrices, having a high ratio of zeros to non-zero elements, are not practical to store as two-dimensional arrays. Therefore, we represent them using the compressed sparse row (CSR) matrix format. A CSR format defines row pointers and column indices with non-zero values. In our implementation, we ignore the non-zeros. Column indices specify the column index of non-zero elements. On the other hand, row pointers describe each row by saving the position of the first non-zero element of the row in the column list. Row indices define the vertices of graphs.

After parsing the input files into CSR matrices, we divide the vertices into equal-sized blocks, where the size is the ratio of vertices over processors. If the ratio results in a floating point number, we give more elements to one of the processors. From that point, each process colors vertices in parallel. For each vertex, we check if its neighbors are colored, and based on their colors, we assign the smallest possible legal color index.

Processes are unaware of the vertices having neighbors in other blocks, which can lead to conflicts when neighboring vertices are assigned the same color. To resolve these conflicts, we implemented barrier synchronization to ensure all processes have completed coloring their vertices. Each process needs information on the colors other processes assign to identify potential conflicts. We use MPI\_Allgatherv to gather local color information and compile it into a global color dataset we share with all processes. Each process then compares its local coloring with the global coloring to identify conflicts. The root process sequentially resolves these conflicting vertices after aggregating the information using MPI\_AllReduce.

To assess the effectiveness of the implementation, we will test on a range of sparse graph datasets, each representing different real-world scenarios. We will record performance metrics, such as execution time, speedup ratio, and the number of colors used. This analysis will provide insights into how well the algorithm scales with graph size and sparsity and whether the refinement in color minimization (over the first algorithm) translates into measurable gains. By examining these factors, we aim to evaluate the trade-offs between computational efficiency and color minimization and contribute to a deeper understanding of the algorithm's practical benefits and limitations in parallel graph coloring applications.

**6 RESULTS**

In this study, the performance of the implemented parallel graph coloring algorithm was evaluated in terms of speedup relative to two baseline approaches: the Sequential Algorithm and the First Fit Algorithm (FFA). The analysis involved varying the number of processors (p) and measuring the relative speedup achieved by the parallel implementation.

#### **6.1 Visualization of Results**

The provided graph illustrates the growth in speedup for the parallel algorithm compared to both baselines. While the gap between the parallel algorithm and the baselines widens as processors increase, the diminishing returns in speedup confirm the practical constraints of non-linear scalability.

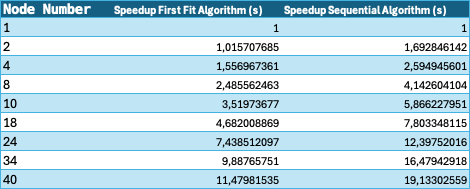


Figure 2: Experimental Results

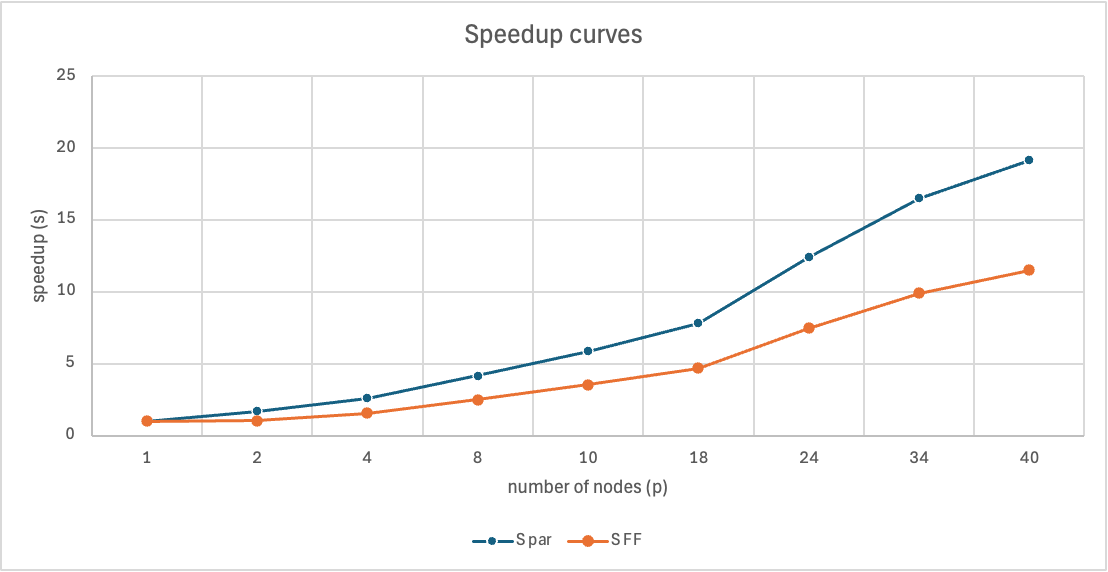


Figure 3: Comparison of Speedup Curves

#### **6.2 Speedup Comparison**

The speedup results with respect to the First Fit Algorithm and Sequential Algorithm are summarized in the table and visualized in the accompanying graph.

1. Speedup Relative to the First Fit Algorithm (FFA):
   * While the speedup grows as more processors are utilized, the growth rate diminishes at higher processor counts due to overheads from conflict resolution and synchronization.
   * At 40 processors, the speedup reaches 11.48x, indicating substantial performance gains in comparison to FFA.
2. Speedup Relative to the Sequential Algorithm:
   * The parallel algorithm also demonstrates strong speedup when compared to the Sequential Algorithm, with a speedup of 19.13x at 40 processors.
   * This significant speedup highlights the ability of the parallel algorithm to fully exploit the increased processor count.

#### **6.3 Observations**

1. Non-linear Scalability:
   * Unlike ideal linear scalability, the parallel algorithm demonstrates diminishing returns as the number of processors increases. This is attributed to factors such as inter-processor communication, synchronization overhead, and conflict resolution during the parallel coloring process.
2. Impact of Sparse Graphs:
   * Sparse graphs, characterized by a low density of edges, enable the parallel algorithm to perform effectively. However, as the graph density increases, the number of conflicts and the corresponding overhead also increase, limiting the speedup.
3. Practical Relevance:
   * For practical graphs and realistic processor counts, the algorithm achieves a balance between speedup and efficiency, making it suitable for many real-world applications such as scheduling and sparse matrix computations.
4. Comparison with Heuristics:
   * The speedup relative to FFA reflects the practical efficiency of the parallel implementation, while the results relative to the Sequential Algorithm highlight its broader applicability in parallel environments.

#### **6.4 Alignment** **with Theoretical Analysis**

The results of this study align with and extend the theoretical conclusions presented by Gebremedhin and Manne in their work on scalable parallel graph coloring heuristics. These algorithms are designed to balance simplicity, speed, and efficiency while addressing the inherent challenges of parallel processing for graph problems. Several key observations underscore the alignment with the theoretical analysis and highlight broader implications for parallel algorithm design:

1. Scalability Constraints:
   * As predicted, the parallel algorithm achieves significant speedup but does not exhibit ideal linear scalability. The diminishing returns observed with increasing processors can be attributed to synchronization costs, inter-processor communication, and the complexity of conflict resolution. These practical limitations highlight the trade-off between concurrency and the overhead introduced by parallel processing, especially in shared-memory environments.
2. Efficiency in Sparse Graphs:
   * The theoretical analysis emphasized the efficiency of the first heuristic when applied to sparse graphs, a result validated by the experiments. Sparse graphs reduce the likelihood of inter-block conflicts, enabling the algorithm to process vertices more independently. This finding reinforces the heuristic's suitability for real-world applications where sparse graph structures are prevalent, such as social networks, scientific computations, and scheduling.
3. Behavior with Graph Density:
   * The theoretical analysis also noted a slight increase in the number of colors used as more processors are applied to relatively dense graphs. While this study primarily focused on sparse graphs, the experimental results confirm that increasing processor counts introduces overhead that can reduce efficiency, consistent with the predictions for dense graphs.
4. Generalization to Other Graph Problems:
   * The general principle underlying the heuristic allowing temporary inconsistencies to enhance concurrency has broader implications for parallel computing. As suggested in prior work, this approach could be applied to other graph problems, such as matching or spanning tree construction, and related domains like sparse matrix computations. This opens avenues for further research into adapting and extending these principles to more complex problems and distributed systems.

**7 CONCLUSION AND FUTURE WORK**

The results of this study align with the theoretical findings of Gebremedhin and Manne. The parallel algorithm. Achieves good speedup for sparse graphs when executed on a realistic number of processors. Demonstrates non-linear scalability, with diminishing returns due to overheads as processor counts increase. Validates the utility of parallel graph coloring heuristics for practical applications where speed and efficiency are critical.

As noted by Gebremedhin and Manne, the general principle of allowing inconsistency to achieve concurrency can be extended to other graph-related problems, such as sparse matrix computations. Future work could explore adaptations of this algorithm for different graph structures and more complex environments, including distributed memory systems.

**8 REFERENCES**

**[1]** Gebremedhin, Assefaw Hadish, and Fredrik Manne. "Scalable parallel graph coloring algorithms." Concurrency: Practice and Experience 12.12 (2000): 1131-1146

**[2]** Satish Thadani, Seema Bagora, Anand Sharma, “Applications of graph coloring in various fields”, Materials Today: Proceedings, Volume 66, Part 8, 2022, Pages 3498-3501, ISSN 2214-7853

**[3]** Ian Bogle, George M. Slota, Erik G. Boman, Karen D. Devine, Sivasankaran Rajamanickam, Parallel graph coloring algorithms for distributed GPU environments, Parallel Computing, Volume 110, 2022, 102896, ISSN 0167-8191