Parallel Graph Coloring Using MPI

1. Introduction

Graph coloring is a fundamental problem in graph theory with numerous applications, such as scheduling, register allocation in compilers, and frequency assignment in wireless networks. The problem involves assigning colors to nodes in a graph such that no two adjacent nodes share the same color. As graphs grow larger, sequential solutions to graph coloring become computationally expensive. Parallel graph coloring leverages distributed processing, where different parts of the graph are handled by separate processes, to achieve faster computation times and scalability for large datasets.

This report focuses on implementing and analyzing a parallel graph coloring algorithm using the Message Passing Interface (MPI) framework. Specifically, it translates the algorithm from Gebremedhin and Manne (2000) into MPI code while exploring and evaluating various improvements to the original algorithm to enhance performance and scalability.

2. Background

Graph coloring is a fundamental concept in graph theory with numerous real-world applications across various domains. In scheduling, for instance, it is used to assign time slots to tasks or exams in such a way that no overlapping tasks occur simultaneously, thereby preventing conflicts. In telecommunications, particularly in wireless networks, graph coloring helps in frequency assignments to ensure that adjacent transmitters do not interfere with each other by operating on the same frequency. Additionally, in the realm of compilers, graph coloring is employed in register allocation to optimize the assignment of variables to a limited number of CPU registers during program execution.

The graph coloring problem is classified as NP-hard, indicating that no known algorithm can solve all instances of the problem efficiently. Despite this complexity, several heuristic and approximation algorithms have been developed to find near-optimal solutions within reasonable time frames. For example, the Recursive Largest First (RLF) algorithm [2] is a heuristic that constructs color classes by iteratively selecting a maximal independent set of vertices. While it doesn't always produce the optimal solution, it is effective for certain classes of graphs.

Another approach [3] involves the use of graph neural networks (GNNs) inspired by principles from statistical physics. These GNNs frame graph coloring as a multi-class node classification problem and utilize unsupervised training strategies based on models like the Potts model. This method has shown promise in providing scalable solutions to large graph instances.

These advancements demonstrate the ongoing efforts to address the computational challenges posed by NP-hard problems like graph coloring, aiming to develop algorithms that can provide efficient and practical solutions in real-world applications

3. Some of The Existing Methods for Solving the Problem

Graph Coloring Approaches

Graph coloring is a technique in which each node of a graph is assigned a color such that no two adjacent nodes share the same color. This problem is fundamental in various applications such as job scheduling, register allocation, and parallel processing. Traditional algorithms for graph coloring, such as the greedy approach, are efficient for small to medium-sized graphs but these methods become inefficient for large-scale graphs due to their sequential nature.

Gebremedhin and Manne's Parallel Algorithm

The parallel graph coloring algorithm proposed by Assefaw Hadish Gebremedhin and Fredrik Manne (2000) is a widely recognized method for solving graph coloring problems in distributed memory environments. Their approach is specifically designed to minimize color conflicts and improve the parallel processing of large graphs across multiple processors. The key features of this algorithm include:

1. **Task Distribution**: The graph is decomposed into smaller subgraphs, which are distributed across different processors. This allows for parallel processing, reducing the overall time complexity of the algorithm.
2. **Conflict Reduction**: A key challenge in parallel graph coloring is managing color conflicts between adjacent nodes processed by different processors. Gebremedhin and Manne address this issue through message passing (MPI), which ensures that the state of nodes is consistently maintained across all processors, thus minimizing conflicts.
3. **Scalability**: The algorithm is designed to scale effectively, enabling it to handle large and complex graphs in distributed systems. It allows for efficient graph coloring, even as the size of the graph increases, by balancing the workload across multiple processors.

Parallel Graph Coloring Algorithms for Distributed GPU Environments [4]

This paper explores hybrid MPI+GPU parallel graph coloring algorithms for distributed systems, particularly focusing on improving the performance of large-scale graph coloring tasks. The authors extend existing distributed memory graph coloring algorithms by integrating GPU acceleration, providing a more scalable and efficient solution. Their approach achieves notable success in handling graphs with over 76 billion edges, demonstrating the potential for MPI+GPU algorithms in distributed environments.

High-performance and balanced parallel graph coloring on multicore platforms [5]

Giannoula et al. present ColorTM, a high-performance graph coloring algorithm designed for modern multicore platforms. The paper highlights the use of Hardware Transactional Memory (HTM) to minimize synchronization and data access costs, proposing an eager conflict detection and resolution strategy. Furthermore, they introduce BalColorTM, an extension of ColorTM that ensures balanced color classes, aiming to reduce load imbalance and improve resource

utilization. Their algorithms significantly outperform prior state-of-the-art methods in both performance and color balancing.

4. Test Environment

The test environment for this project is an HP Victus 16-R1021NT notebook running Windows 11, powered by an Intel Core i7-14700HX processor. As a high-performance CPU from Intel's 14th generation Raptor Lake series, launched in 2024, it combines robust computational capabilities with efficiency, making it well-suited for demanding tasks like parallel graph processing.

The i7-14700HX features a hybrid architecture with 20 cores and 28 threads, including 8 Performance cores (P-cores) and 12 Efficient cores (E-cores). P-cores operate at a base frequency of 2.1 GHz, turbo boosting to 5.5 GHz, while E-cores range from 1.5 GHz to 3.9 GHz. This design allows for efficient multitasking, rapid computation, and enhanced performance in workloads requiring simultaneous processing.

Built on Intel's advanced 10nm process technology (Intel 7), the processor includes 33 MB of Intel Smart Cache and supports up to 192 GB of DDR5 memory at speeds of up to 5600 MT/s. Additionally, its integrated Intel UHD Graphics and balanced thermal design power (55 W TDP, 157 W turbo) further ensure reliable performance. This setup provides an ideal platform for implementing and testing parallel graph coloring algorithms on complex datasets.

5. Data

The data set for this study comprises five different graphs selected to evaluate the performance and scalability of the parallel graph coloring algorithm. These graphs vary significantly in size and edge density, representing diverse challenges for the algorithm.

Graph	Vertices	Edges	Source
games120.col	120	1276	https://mat.tepper.cmu.edu/COLOR/instances.html
latin_square_10.col	900	307350	https://mat.tepper.cmu.edu/COLOR/instances.html
le450_15d.col	450	16750	https://mat.tepper.cmu.edu/COLOR/instances.html
C2000-5.mtx	2000	999836	https://networkrepository.com/dimacs.php
C4000-5.mtx	4000	4000268	https://networkrepository.com/dimacs.php

The graphs were sourced from reputable online repositories such as the Network Data Repository. The first three graphs [6] (games120.col, latin_square_10.col, and le450_15d.col) are classic benchmarks used in graph coloring research, each featuring varying vertex counts and edge densities. The last two graphs [7] (MatrixMarket1 and MatrixMarket2) were derived from the DIMACS challenge datasets and represent larger, denser structures suitable for testing scalability and parallel performance.

These datasets provide a robust basis for analyzing the algorithm's ability to handle different graph topologies and scales effectively.

6. Algorithms

In this section, I present the algorithms developed for solving the parallel graph coloring problem. These include a sequential algorithm, multiple versions of Algorithm 1 and Algorithm 2 from the paper and a new Algorithm 3, each derived from different strategies and approaches to improve efficiency and performance.

Sequential Algorithm

The sequential algorithm is motivated by the idea behind Culberson's Iterated Greedy (IG) [8] coloring heuristic. IG relies on the concept that coloring can be improved by reordering the vertices in a specific way before applying a greedy coloring algorithm. Specifically, the algorithm uses a technique known as *reverse color class ordering*.

In the context of IG, the reverse color class ordering is an ordering of vertices that groups the vertices of the same color class consecutively. This ordering helps reduce the number of colors used by the greedy coloring algorithm. The key insight here is that if the vertices of a graph are reordered so that those belonging to the same color class are grouped together, and the color classes with higher numbers are processed first, applying the First Fit (FF) algorithm will either maintain or reduce the total number of colors required for coloring the graph. This approach enhances the efficiency of sequential coloring by leveraging the structure of the previous coloring and minimizing color conflicts.

After the initial greedy coloring step, the algorithm enters an iterative improvement phase with a maximum limit of 200 iterations. During each iteration, the vertices are reordered using the reverse color class ordering technique, where color classes are processed starting from the highest-numbered class. This reordering is followed by reapplying the First Fit (FF) algorithm to attempt to reduce the total number of colors used.

To optimize performance, the algorithm employs an early stopping mechanism. If the number of colors used does not decrease for five consecutive iterations, the improvement phase is terminated early, avoiding unnecessary computations. This strategy ensures efficiency while still striving to minimize the total number of colors required. The results obtained from experiments on five different graph datasets demonstrate that, thanks to early stopping, the algorithm completed its execution without reaching the maximum of 50 iterations,

The combination of the reverse color class ordering and early stopping helps balance computational effort with the quality of the resulting coloring, making this approach both practical and effective for sequential graph coloring.

Algorithm 1 Block Partition-Based Coloring

This algorithm introduces a parallel graph coloring strategy divided into three main phases. In the first phase, the graph's vertex set V is divided into p equal-sized blocks (V1,V2,…,VP), with each block assigned to a separate processor. The vertices within each block are then colored in

parallel using a greedy approach. At each step, synchronization barriers ensure that all processors progress simultaneously. However, this phase may result in “pseudo-coloring" because two adjacent vertices handled by different processors can be assigned the same color.

The second phase involves conflict detection, where each processor checks whether the coloring is valid for the vertices in its block by comparing the colors of vertices and their neighbors that were processed in the same step. Conflicting vertices are identified, and their details are stored in a table. In the final phase, these conflicting vertices are sequentially recolored to resolve all conflicts, ensuring the graph is properly colored. This hybrid approach balances parallel efficiency and correctness, leveraging parallelism in the initial coloring and conflict detection phases while addressing conflicts sequentially for accuracy.

I implemented this algorithm in three different variations, each exploring a unique approach to improve or adapt the original strategy:

Algorithm 1

BlockPartitionBasedColoring(G, p)
begin
1. Partition V into p equal blocks $V_1 \dots V_p$, where $\left\lfloor \frac{n}{p} \right\rfloor \le |V_i| \le \left\lceil \frac{n}{p} \right\rceil$
   for $i = 1$ to p do in parallel
      for each $v_j \in V_i$ do
         assign the smallest legal color to vertex v_j
         barrier synchronize
      end-for
   end-for
2. for $i = 1$ to p do in parallel
      for each $v_j \in V_i$ do
         for each neighbor u of v_j that is colored at the same parallel step do
            if $color(v_j) = color(u)$ then
               store min $\{u, v_j\}$ in table A
            end-if
         end-for
      end-for
   end-for
3. Color the vertices in A sequentially
end

Added barrier synchronize

Figure 1.1 Algorithm 1

begin
1. Partition V into p equal blocks $V_1 \dots V_p$, where $\left\lfloor \frac{n}{p} \right\rfloor \le |V_i| \le \left\lceil \frac{n}{p} \right\rceil$
   for $i = 1$ to p do in parallel
      for each $v_j \in V_i$ do
         assign the smallest legal color to vertex v_j
      ~~barrier synchronize~~
      end-for
   end-for

Added barrier synchronize

While conflicts exist:

Step 1: Parallel Conflict Detection

Each processor detects conflicts for vertices in its block:

Collect conflict information from all processors (global synchronization).

Step 2: Parallel Conflict Resolution

Each processor recolors its conflicting vertices:

Figure 1.2 Algorithm 1 Asynchronous

Algorithm 1 - Basic

Algorithm 1 Basic, depicted in **Figure 1.1**, serves as the simplest implementation of the graph coloring algorithm. It utilizes fundamental MPI functions like MPI_GATHER and MPI_BARRIER for inter-process communication and synchronization. In this approach, the graph is divided into equal parts, and each process performs greedy coloring in parallel, followed by a global synchronization step. Conflicts are resolved sequentially after this phase, which can lead to inefficiencies due to the lack of advanced parallelism. This algorithm provides a straightforward and foundational method for comparison with more optimized approaches.

Algorithm 1 - Half Asynchronous

Algorithm 1 Half Asynchronous, based on **Figure 1.2**, improves upon the basic version by leveraging more advanced MPI functions such as MPI_Allgather and MPI_Allreduce. These functions enable efficient global data sharing and conflict detection in parallel, significantly reducing communication overhead. After the initial parallel coloring phase, the algorithm iteratively detects and resolves conflicts in parallel, alternating between resolving "small" (lower vertex ID) and "large" (higher vertex ID) conflicts in each iteration. This alternation is targeted to help balance the computational load across processes and enhances scalability, making this version more suitable for larger and more complex graphs.

Algorithm 1 - Asynchronous

Algorithm 1 Asynchronous extends the Half Asynchronous approach by introducing non-blocking MPI communication functions, such as MPI_Isend and MPI_Irecv, to overlap computation and communication. This version further optimizes parallelism by minimizing idle times and reducing synchronization delays between processes. Like the Half Asynchronous algorithm, it alternates between resolving "small" and "large" conflicts during the iterative conflict resolution phase to even distribute the workload.

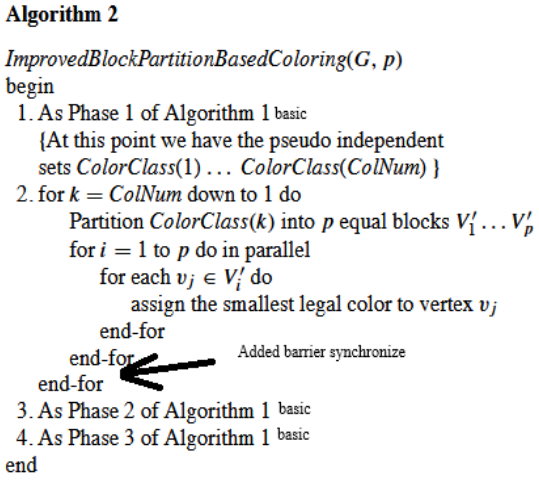

Figure 2.1 Algorithm 2

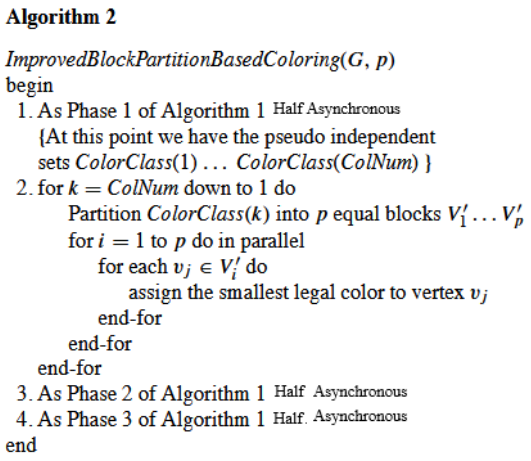

Figure 2.2 Algorithm 2 Asynchronous

Algorithm 2 - Basic

Algorithm 2 Basic, shown in **Figure 2.1**, provides a foundational implementation of the graph coloring algorithm using basic MPI functions like MPI_GATHER and MPI_BARRIER. Like Algorithm 1 Basic, this version divides the graph into parts and performs greedy coloring in parallel. However, its conflict detection and resolution are simplified and handled sequentially after synchronization, limiting its performance on larger datasets. This algorithm serves as a straightforward starting point for understanding the enhancements made in subsequent versions.

Algorithm 2 - Parallel

Algorithm 2 Parallel builds upon Algorithm 2 Basic, introducing an improvement inspired by Algorithm 1 Half Asynchronous. Specifically, it adopts the advanced conflict detection and resolution methods from Algorithm 1 Half Asynchronous, where conflicts are resolved in parallel. This allows for more efficient handling of coloring conflicts while retaining the overall structure of Algorithm 2 Basic. As a result, this version strikes a balance between simplicity and improved scalability compared to the basic approach.

Algorithm 2 - Half Asynchronous

Algorithm 2 Half Asynchronous, based on Figure 2.2, introduces advanced communication and parallelism strategies similar to those in Algorithm 1 Half Asynchronous. It uses MPI_Allgather and MPI_Allreduce for efficient global conflict detection and parallel conflict resolution. During each iteration, the algorithm alternates between resolving "small" (lower vertex ID) and "large" (higher vertex ID) conflicts, ensuring balanced workload distribution. Additionally, the reverse color ordering in the second phase is implemented differently from what is shown in Figure 2.2. Here, processes handle only their assigned subgraph and perform the reverse ordering within that area, checking conflicts in reverse order only once. This localized reverse-order adjustment improves efficiency by limiting the scope of conflict checks to relevant areas, making the algorithm more scalable for larger graphs.

Algorithm 2 - Asynchronous

Algorithm 2 Asynchronous also relies on **Figure 2.2** and extends the Half Asynchronous version by incorporating non-blocking MPI operations, such as MPI_Isend and MPI_Irecv. These asynchronous methods overlap computation with communication, reducing idle times and improving performance. Like Algorithm 2 Half Asynchronous, it alternates between resolving "small" and "large" conflicts during iterative conflict resolution.

Algorithm 3

Algorithm 3 closely resembles Algorithm 2 Asynchronous, with one key difference in how conflicts are resolved during each iteration. Instead of alternating between resolving "small" and "large" conflicts, the data distribution shifts as though the ranks of the processes were rotated forward by one in each iteration. This modification ensures that the workload distribution varies across iterations, potentially leading to better load balancing and improved parallel performance for certain types of graphs.

7. Results and Discussion

The experimental results indicate that all the proposed algorithms, when executed with one process, were faster than the sequential algorithm. This speed advantage arises because the sequential algorithm employs an epoch mechanism, which introduces additional computational overhead.

Despite their speed, the parallel algorithms consistently used approximately 5% more colors than the sequential algorithm when using only 1 processor. This discrepancy stems from differences in the coloring strategies between the parallel and sequential implementations, despite sequential algorithm employing extra work for reducing color number with epoch mechanism I think the difference is not that big. For more than one processor the difference in general is bigger than %10. When I increased the number of processes to 20, it dropped to about 10%.

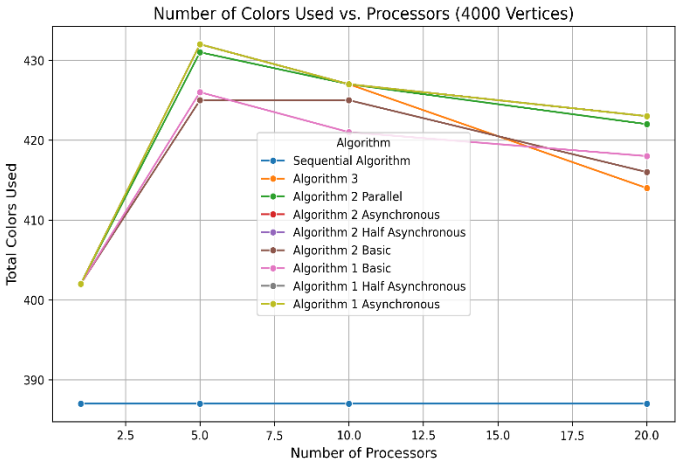

Figure 3 Number of Colors vs Number of Processor

Interestingly, as the number of processes increased beyond five, the total number of colors used by the parallel algorithms decreased. This behavior can be attributed to enhanced parallel conflict resolution, where dividing the graph among more processes reduces the likelihood of conflicts in specific subgraphs. The localized nature of conflict detection and resolution in the parallel algorithms allows for more efficient use of colors, especially as the graph is divided into finer partitions.

However, it is worth noting that all the parallel algorithms exhibited their fastest performance when executed with only one process. This counterintuitive result is likely due to the experimental setup, as the tests were conducted on a Windows-based system. The Windows operating system may introduce additional overhead or inefficiencies in managing inter-process communication (MPI) compared to a Linux-based system, which is generally more optimized for high-performance computing tasks

Example Results for The Graph named C4000-5

Algorithm Name	Processor Number	Total Colors Used	Execution Time (ms)	Speed Up	Speed Up S	Efficiency
Sequential Algorithm	1	387	45884.307	1.0	1.0	1.0
Algorithm 3	20	414	299794.85	0.026	0.153	0.001
Algorithm 3	10	427	108753.795	0.072	0.422	0.007
Algorithm 3	5	432	47136.295	0.167	0.973	0.033
Algorithm 3	1	402	7853.366	1.0	5.843	1.0
Algorithm 2 Parallel	20	422	15262.056	0.527	3.006	0.026
Algorithm 2 Parallel	10	427	13663.099	0.589	3.358	0.059
Algorithm 2 Parallel	5	431	13911.217	0.579	3.298	0.116
Algorithm 2 Parallel	1	402	8049.329	1.0	5.7	1.0
Algorithm 2 Asynchronous	20	423	10023.43	0.786	4.578	0.039
Algorithm 2 Asynchronous	10	427	9205.074	0.855	4.985	0.086
Algorithm 2 Asynchronous	5	432	9417.461	0.836	4.872	0.167
Algorithm 2 Asynchronous	1	402	7874.62	1.0	5.827	1.0
Algorithm 2 Half Asynchronous	20	423	10489.482	0.753	4.374	0.038
Algorithm 2 Half Asynchronous	10	427	9414.263	0.839	4.874	0.084
Algorithm 2 Half Asynchronous	5	432	9434.396	0.838	4.864	0.168
Algorithm 2 Half Asynchronous	1	402	7902.97	1.0	5.806	1.0
Algorithm 2 Basic	20	416	15570.496	0.512	2.947	0.026
Algorithm 2 Basic	10	425	15466.5	0.515	2.967	0.052
Algorithm 2 Basic	5	425	14531.024	0.548	3.158	0.11
Algorithm 2 Basic	1	402	7969.456	1.0	5.758	1.0
Algorithm 1 Basic	20	418	11000.017	0.257	4.171	0.013
Algorithm 1 Basic	10	421	16740.322	0.169	2.741	0.017
Algorithm 1 Basic	5	426	9820.658	0.288	4.672	0.058
Algorithm 1 Basic	1	402	2828.746	1.0	16.221	1.0
Algorithm 1 Half Asynchronous	20	423	10332.066	0.279	4.441	0.014
Algorithm 1 Half Asynchronous	10	427	9260.948	0.311	4.955	0.031
Algorithm 1 Half Asynchronous	5	432	9216.474	0.313	4.979	0.063
Algorithm 1 Half Asynchronous	1	402	2882.596	1.0	15.918	1.0
Algorithm 1 Asynchronous	20	423	10248.238	0.247	4.477	0.012
Algorithm 1 Asynchronous	10	427	9205.467	0.275	4.984	0.028
Algorithm 1 Asynchronous	5	432	8877.042	0.285	5.169	0.057
Algorithm 1 Asynchronous	1	402	2531.584	1.0	18.125	1.0

Figure 4 Example Results

All results can be found in GitHub [9] or can be found in the python code that I shared with this report. In GitHub you can also find all my code for this project.

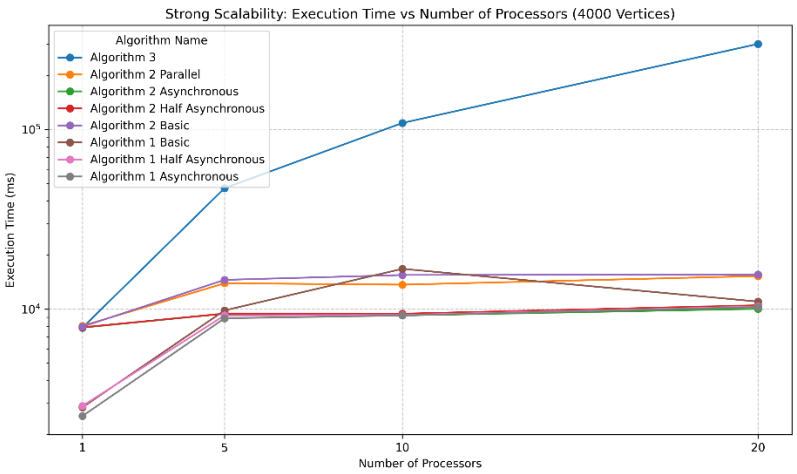

Figure 5 Scalability Graph

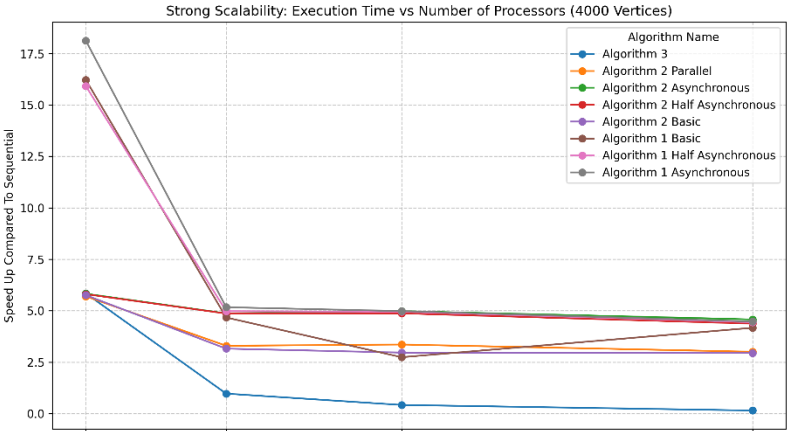

Figure 6 Scalability Graph against Sequential Algorithm

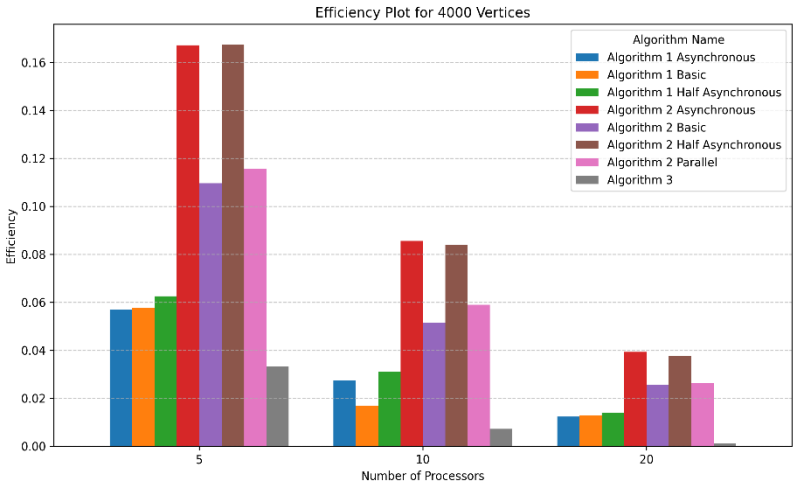

Figure 7 Efficiency Plot

Figures 5, 6, and 7 provide insights into the performance of the algorithms. Except for Algorithm 3, all parallel algorithms outperformed the Sequential Algorithm in terms of execution speed. Among these, the Half Asynchronous and Asynchronous versions emerged as the fastest, showcasing the benefits of reduced synchronization overhead and improved parallelism in their designs.

However, as previously mentioned, increasing the number of processes from 1 to 5 resulted in a significant speed loss across all algorithms. Beyond the 5 processes, there were no substantial speed gains or losses, except for Algorithm 3. This algorithm exhibited substantial performance degradation as the number of processes increased.

The observed trends likely stem from the test environment, particularly the Windows operating system and its implementation of the MPI library. It appears that Windows—or possibly my computer—fails to effectively utilize processors beyond the first one. These inefficiencies in inter-process communication are more pronounced on Windows compared to Linux, which is generally better optimized for parallel high-performance tasks.

Regarding Algorithm 3, I do not believe the modifications introduced in its design are significant enough to explain its unique behavior. Instead, the performance issues are likely due to system-related factors rather than algorithmic design.

It is important to note that these conclusions are based on experiments conducted on a graph named C4000-5 ensuring consistency across all measurements. I used C4000-5 because it is the biggest graph in my project.

8. Conclusion

In this study, I think I successfully implemented and analyzed parallel graph coloring algorithms using MPI, based on the work of Gebremedhin and Manne (2000) [1]. The primary goal was to translate their algorithm into an MPI-based parallel implementation, explore variations of the algorithm, and evaluate their performance. My findings highlight the trade-offs between execution speed, parallel efficiency, and the number of colors used. The experimental results showed that all parallel algorithms, except Algorithm 3, were faster than the sequential algorithm when executed with a single process, primarily due to the absence of the epoch mechanism. However, this speed advantage came at the cost of slightly higher color usage.

The analysis revealed key insights into the behavior of algorithms as the number of processes increased in the test environment I used. While all parallel algorithms initially experienced significant speed losses when scaling from 1 to 5 processes, their performance stabilized beyond 5 processes. Notably, the Half Asynchronous and Asynchronous algorithms consistently outperformed the others in terms of execution speed. Algorithm 3, however, exhibited significant performance degradation as the number of processes increased, likely due to inefficiencies.

In conclusion, this study demonstrates the feasibility and challenges of implementing parallel graph coloring algorithms using MPI. While the parallel algorithms achieved notable speedups compared to the sequential version likely due to not having epoch mechanism, their performance was probably limited by the experimental setup, particularly the inefficiencies of the MPI library on Windows. Future work could focus on optimizing the implementation for Linux-based systems to see if the real reason for performance degradation when using more than one process is because of the test environment or exploring alternative parallel computing frameworks. Additionally, further investigation into the performance anomalies of Algorithm 3 and the scalability of these algorithms on larger and more complex graphs would provide deeper insights into their potential for real-world applications.

References

[1] Gebremedhin, A. H., & Manne, F. (2000). Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, *12*(12), 1131–1146. https://doi.org/10.1002/1096-9128(200010)12:12%3C1131::aid-cpe528%3E3.0.co;2-2

[2] Leighton, F. T. (1979). A graph coloring algorithm for large scheduling problems. *Journal of Research of the National Bureau of Standards*, *84*(6), 489. https://doi.org/10.6028/jres.084.024

[3] Schuetz, M. J. A., Brubaker, J. K., Zhu, Z., & Katzgraber, H. G. (2022). Graph coloring with physics-inspired graph neural networks. *Physical Review Research*, *4*(4). https://doi.org/10.1103/physrevresearch.4.043131

[4] Bogle, I., Slota, G. M., Boman, E. G., Devine, K. D., & Rajamanickam, S. (2022). Parallel graph coloring algorithms for distributed GPU environments. *Parallel Computing*, *110*, 102896. https://doi.org/10.1016/j.parco.2022.102896

[5] Giannoula, C., Peppas, A., Goumas, G., & Koziris, N. (2022). High-performance and balanced parallel graph coloring on multicore platforms. *The Journal of Supercomputing*, *79*(6), 6373–6421. https://doi.org/10.1007/s11227-022-04894-6

[6] *Graph Coloring Instances*. (n.d.). Mat.tepper.cmu.edu. https://mat.tepper.cmu.edu/COLOR/instances.html

[7] Network Data Repository. (2015). *DIMACS Networks*. Network Data Repository. https://networkrepository.com/dimacs.php

[8] Culberson, J. (1992). *Iterated Greedy Graph Coloring and the Difficulty Landscape*. https://doi.org/10.7939/r3m32nh6q

[9] b21945815. (2025). *GitHub - b21945815/METU-MPI-GraphColoringProblem*. GitHub. https://github.com/b21945815/METU-MPI-GraphColoringProblem