MIDDLE EAST TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CEng 577: PARALLEL COMPUTING

FINAL PROJECT REPORT

# TOPIC: *Implementation of the parallel graph coloring algorithms using MPI.*

**Çağlar TEMİZ**

*MSc. Student in Civil Engineering*

**2227940**

**Due Date: 19/01/2025**

# Table of Contents

***Motivation:***

*The first reason that I chose the project topic #PT5: Parallel Graph Coloring in project topics, I really liked graph data structure when I was studying algorithms and data structures as a non-CENG student. With this project, I will try to explore how well it performs with different types of graphs and varying numbers of processors. This project will also provide to increase my understanding of parallel computing concepts like communication strategies, load balancing, and scalability.*

## INTRODUCTION

A graph G is a pair of vertices and edges, G = (V, E). Graph coloring is a concept in computer science that involves assigning labels (colors) to the vertices (V) of a graph where two adjacent vertices do not share the same color. Graph coloring problem (GCP) is significant in scientific computing and engineering application. In spite of its wide range, graph coloring is challenging due to its NP-hard (non-deterministic polynomial hard) nature, meaning that finding the optimal solution is computationally difficult as the problem size grows. Graph coloring can be used in many areas of the life. For example, scheduling, regional map coloring, network topology design are the areas that graph coloring can be used.

The focus of this project is to implement and study one of the parallel graph coloring algorithms shown in the paper named *"Scalable Parallel Graph Coloring Algorithms"* (Gebremedhin & Manne, 2000). In a graph, vertices represent entities, and edges represent relationships between them. The goal of graph coloring is to assign a minimum number of colors to the vertices whereas no two connected vertices have the same color when it is in a parallel computing context, where the work must be divided in between multiple processors.

In this project preproposal and proposal, the previous work is discussed first. Next, the two parallel graph coloring algorithms mentioned in the previous section are introduced, and one of them is selected as the method for this study. Finally, the selected method is analyzed based on its properties. In this final report, the Implementation of the Block Partition Based Coloring algorithm using MPI with C++ is shown, and the effectiveness of this parallel code is discussed according to speedup based on first fit sequential algorithm.

## PREVIOUS SEQUENTIAL ALGORITHMS

Some of the previous heuristic sequential graph coloring algorithms are summarized below. Even though they are sequential, they are fast and efficient.

### Largest Degree First Ordering (LFO):

This algorithm selects vertices based on their degree, vertices with the highest degree will be prioritized with the aim of that to reduce the complexity of subsequent coloring steps. The complexity is O(m), where m is the number of edges.

### Saturation Degree Ordering (SDO):

Vertices are prioritized based on the number of different colors already assigned to their adjacent vertices (saturation degree). Vertices with more diverse neighboring colors are colored earlier, which often leads to fewer overall colors. The complexity is $O(n^2)$, where n is the number of vertices (Brelaz, 1979).

### Incidence Degree Ordering (IDO):

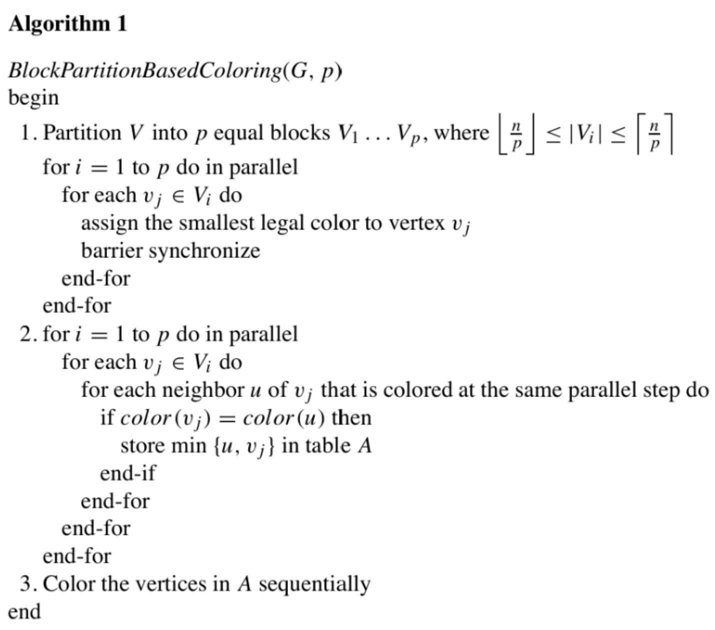
Vertices are prioritized based on the number of vertices that have already been colored. (incidence degree). It aims to resolve conflicts between neighbors earlier. The complexity is O(m), where m is the number of edges.

### First Fit (FF):

Vertices are colored in an arbitrary order, assigning the smallest possible color that has not been used by their neighbors. The complexity is O(m), where m is the number of edges (Grimmet & McDiarmid, 1975).

## MODEL OF THIS STUDY

In this study, our model is selected the first algorithm, Block Partition Based Coloring, in Gebremedhin & Manne (2000). The aim of the algorithm is to achieve a better speedup in parallel environments. Figure 1 shows how this algorithm works basically. Then, the steps are explained how this algorithm is implemented as a MPI/C++ code.



**Algorithm 1**

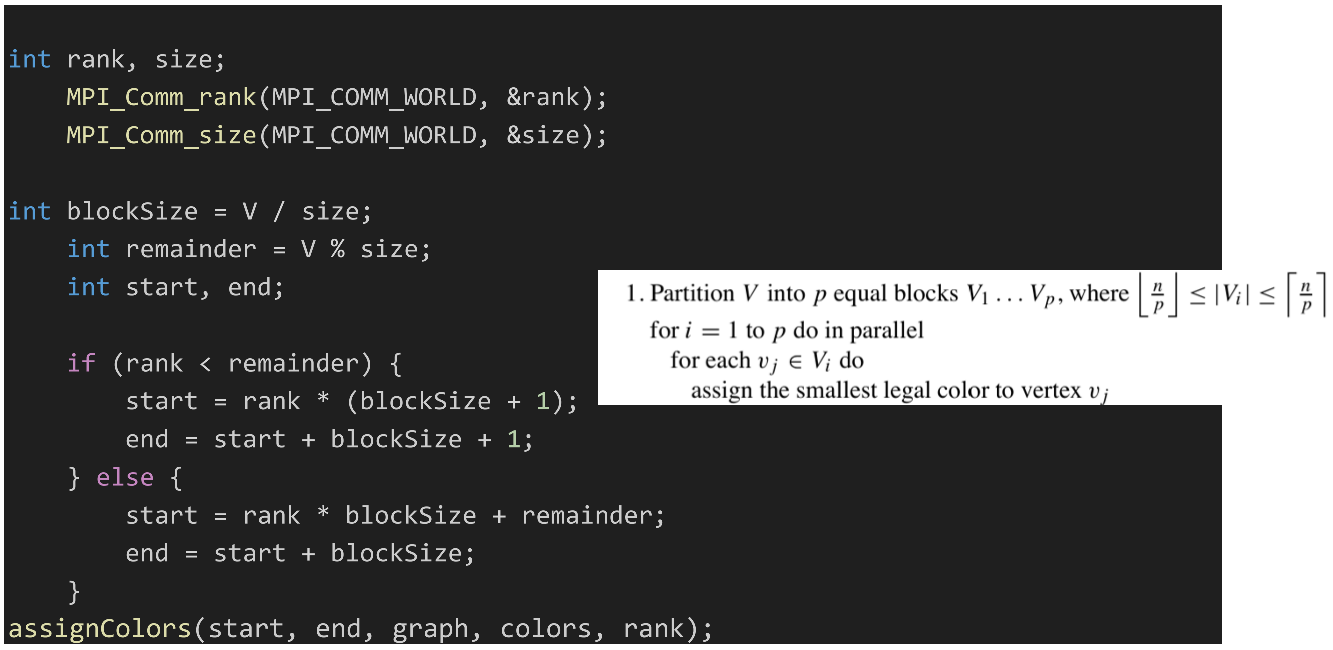
```
BlockPartitionBasedColoring(G, p)
begin
 1. Partition $V$ into $p$ equal blocks $V_1 \dots V_p$, where $\lfloor \frac{n}{p} \rfloor \le |V_i| \le \lceil \frac{n}{p} \rceil$
    for $i = 1$ to $p$ do in parallel
      for each $v_j \in V_i$ do
        assign the smallest legal color to vertex $v_j$
        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
```

***Figure 1:*** *Block Partition Based Coloring Algorithm.*

## Steps

- **Partitioning:**

The input vertex set V is partitioned into *p* (number of processors) blocks. All blocks are colored in n/p steps. (n: number of vertices). After the processor id and size is defined, each processor starts to color its own graph part. For example, if the rank is 0, this processor colors graph vertices starting from 0 to 124 when 4 processors are worked.



```
int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

int blockSize = V / size;
    int remainder = V % size;
    int start, end;

    if (rank < remainder) {
        start = rank * (blockSize + 1);
        end = start + blockSize + 1;
    } else {
        start = rank * blockSize + remainder;
        end = start + blockSize;
    }
assignColors(start, end, graph, colors, rank);
```

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

***Figure 2:*** *Partitioning implementation of the algorithm.*

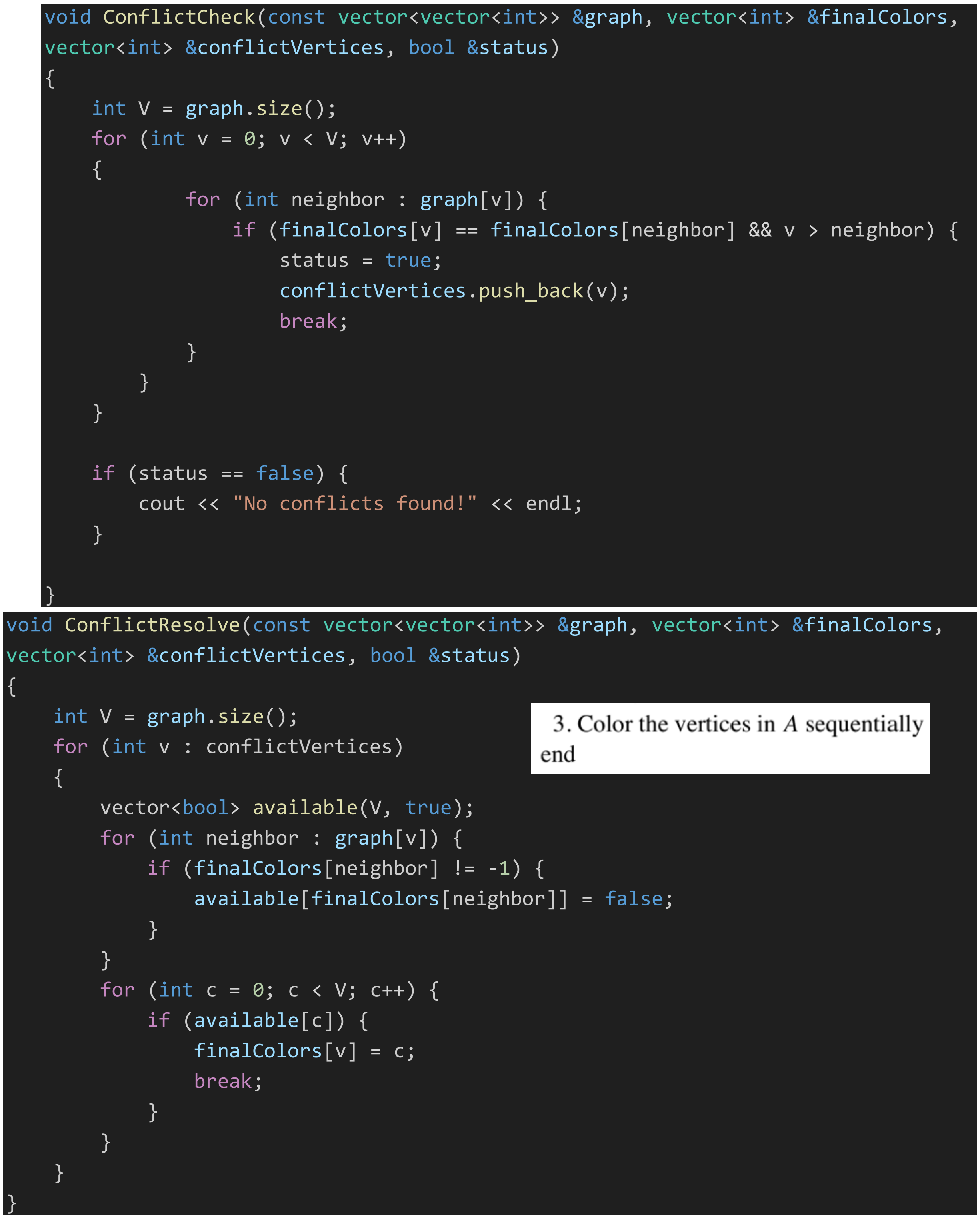
- **Parallel Pseudo-Coloring:** Different processors are colored different vertices with checking if there is a conflict. Then, all colored vertices data is send to MASTER processor (in my case, the processor ranked by 0). MPI_Send and MPI_Recv subroutines are used for communication and MPI_Wtime is used for time check.

```
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
```

```
// Assigning colors to vertices from the process rank's assigned range
void assignColors(int start, int end, const vector<vector<int>> &graph, vector<int>
&colors, int &rank) {

    double startTime = MPI_Wtime();

    int V = graph.size(); // Number of vertices
    vector<bool> available(V, true); // Track available colors

    for (int v = start; v < end; v++) {
        // Reseting available colors for this vertex at each coloring step
        fill(available.begin(), available.end(), true);

        // Mark colors of adjacent vertices as unavailable
        for (int neighbor : graph[v]) {
            if (colors[neighbor] != -1) {
                available[colors[neighbor]] = false;
            }
        }
        // Assigning the smallest available color
        for (int c = 0; c < V; c++) {
            if (available[c]) {
                colors[v] = c;
                break;
            }
        }
    }
    double stopTime = MPI_Wtime();
    cout << "Rank " << rank << " colored vertices in " << stopTime - startTime << "
seconds" << endl;
}
```

***Figure 3:*** *Parallel Coloring implementation of the algorithm.*

- **Conflict Detection and Resolution:** Conflict detection and resolution process is solved sequentially. After all data is sent to master rank, they are solved vertex by vertex as in First Fit Algorithm.



```
void ConflictCheck(const vector<vector<int>> &graph, vector<int> &finalColors,
vector<int> &conflictVertices, bool &status)
{
    int V = graph.size();
    for (int v = 0; v < V; v++)
    {
            for (int neighbor : graph[v]) {
                if (finalColors[v] == finalColors[neighbor] && v > neighbor) {
                    status = true;
                    conflictVertices.push_back(v);
                    break;
            }
        }
    }

    if (status == false) {
        cout << "No conflicts found!" << endl;
    }

}
```

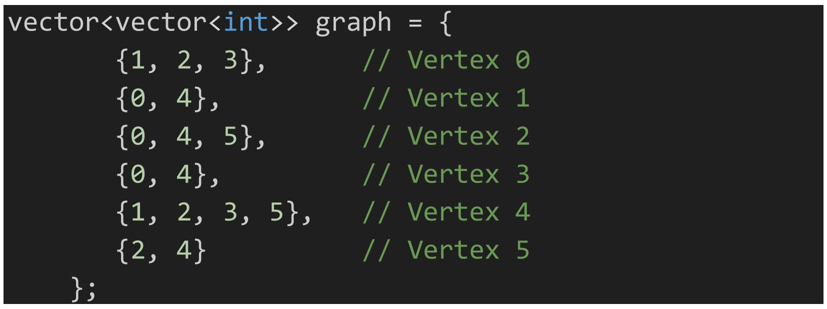
```
void ConflictResolve(const vector<vector<int>> &graph, vector<int> &finalColors,
vector<int> &conflictVertices, bool &status)
{
    int V = graph.size();
    for (int v : conflictVertices)
    {
        vector<bool> available(V, true);
        for (int neighbor : graph[v]) {
            if (finalColors[neighbor] != -1) {
                available[finalColors[neighbor]] = false;
            }
        }
        for (int c = 0; c < V; c++) {
            if (available[c]) {
                finalColors[v] = c;
                break;
            }
        }
    }
}
```

3. Color the vertices in $A$ sequentially
end

***Figure 4:*** *Conflict Detection and Resolution implementation of the algorithm.*

**Example outputs from the codes**

The code is written for graphs in the form of adjacency list. For tests, small graphs are used. Here is an output and an example representation of a graph.



```
vector<vector<int>> graph = {
        {1, 2, 3},        // Vertex 0
        {0, 4},           // Vertex 1
        {0, 4, 5},        // Vertex 2
        {0, 4},           // Vertex 3
        {1, 2, 3, 5},     // Vertex 4
        {2, 4}            // Vertex 5
    };
```

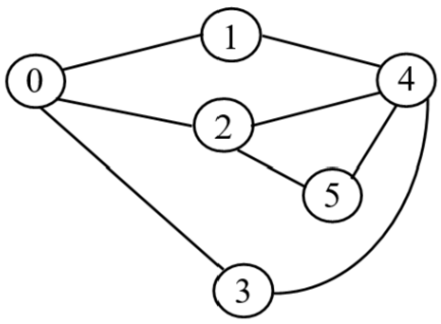
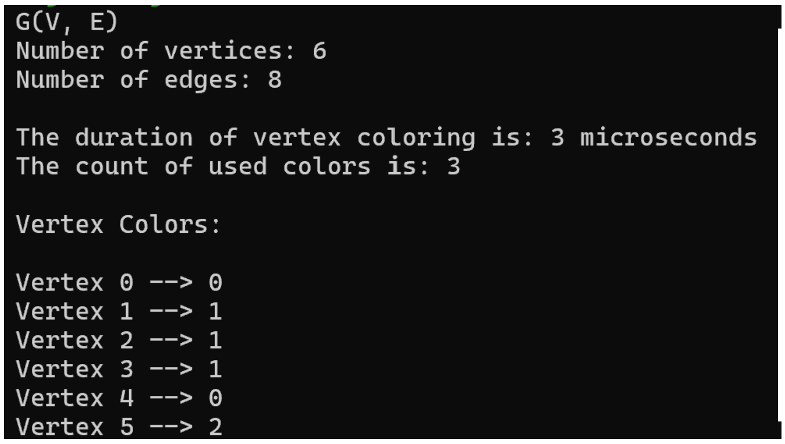


*Figure 4:* *Graph Representation.*



```
G(V, E)
Number of vertices: 6
Number of edges: 8

The duration of vertex coloring is: 3 microseconds
The count of used colors is: 3

Vertex Colors:

Vertex 0 --> 0
Vertex 1 --> 1
Vertex 2 --> 1
Vertex 3 --> 1
Vertex 4 --> 0
Vertex 5 --> 2
```

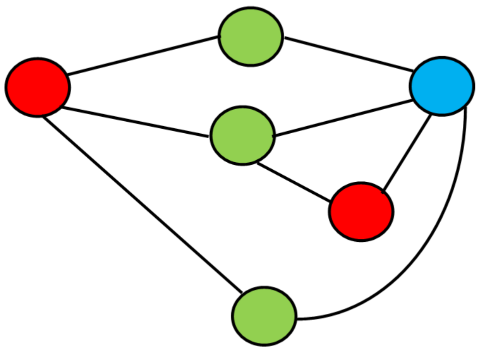
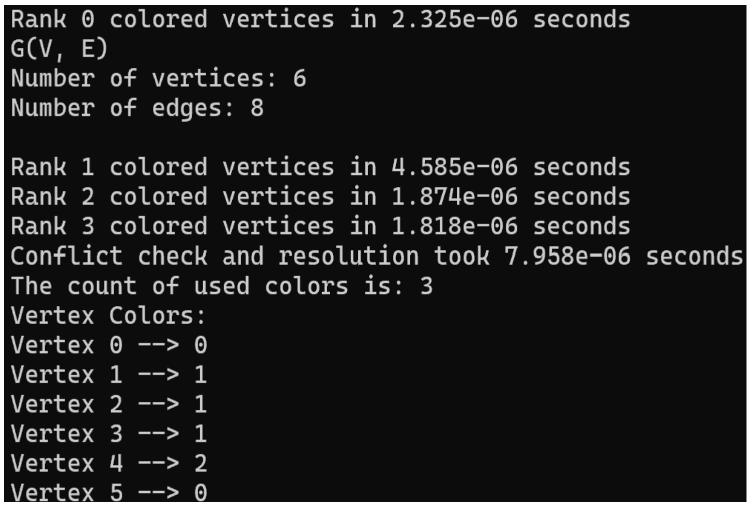


*Figure 5:* *Graph Coloring by FF.*



```
Rank 0 colored vertices in 2.325e-06 seconds
G(V, E)
Number of vertices: 6
Number of edges: 8

Rank 1 colored vertices in 4.585e-06 seconds
Rank 2 colored vertices in 1.874e-06 seconds
Rank 3 colored vertices in 1.818e-06 seconds
Conflict check and resolution took 7.958e-06 seconds
The count of used colors is: 3
Vertex Colors:
Vertex 0 --> 0
Vertex 1 --> 1
Vertex 2 --> 1
Vertex 3 --> 1
Vertex 4 --> 2
Vertex 5 --> 0
```

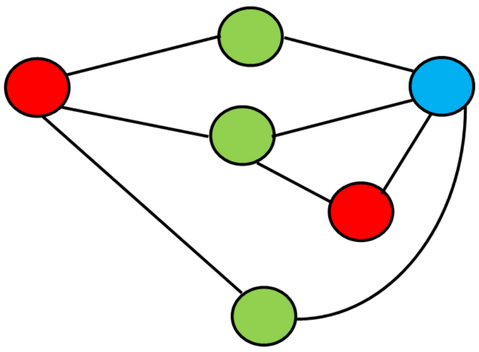


*Figure 6:* *Graph Coloring by Parallel Code.*

## Discussion about the algorithm

According to the study conducted by Gebremedhin and Manne (2000), the algorithm should be simple, fast, and scalable. It offers linear speedup for sparse graphs when the number of processors is proportional to the graph's sparsity, and it performs well on shared-memory systems. The limitation is that number of colors will increase as the number of processors grows. To check the algorithms speedup, FF implementation code written by myself is accepted the best sequential algorithm and the duration is checked for the processor counts 1, 2, 4, 6, and 8.  For large graphs, SuiteSparseMatrixCollection website is used, 494_bus.mtx and 1138_bus.mtx files are downloaded and converted to the usable txt files.

There are four graphs below that represents the results for the duration of First Fit (FF) and Block Partition Based Coloring algorithms for the specified matrices. First two of them shows total duration whereas last two reveals the duration for average time for per processor.

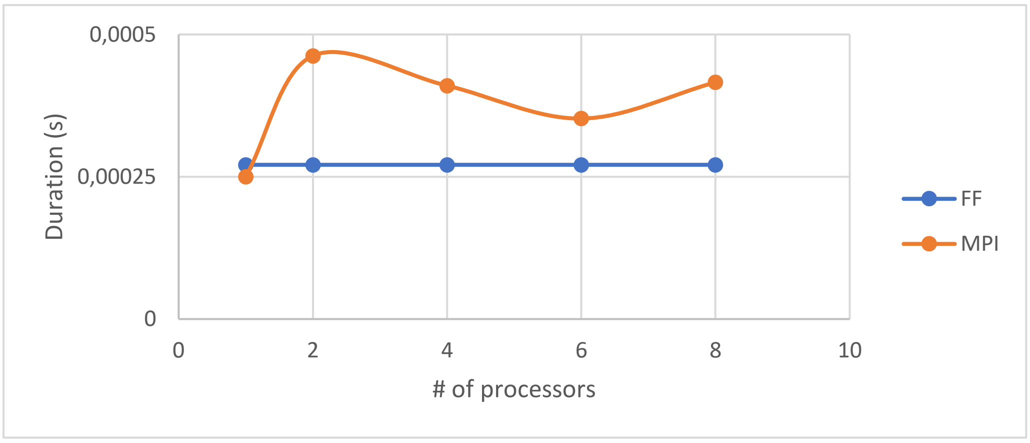


***Figure 7:*** *Total duration for 494_bus.mtx*

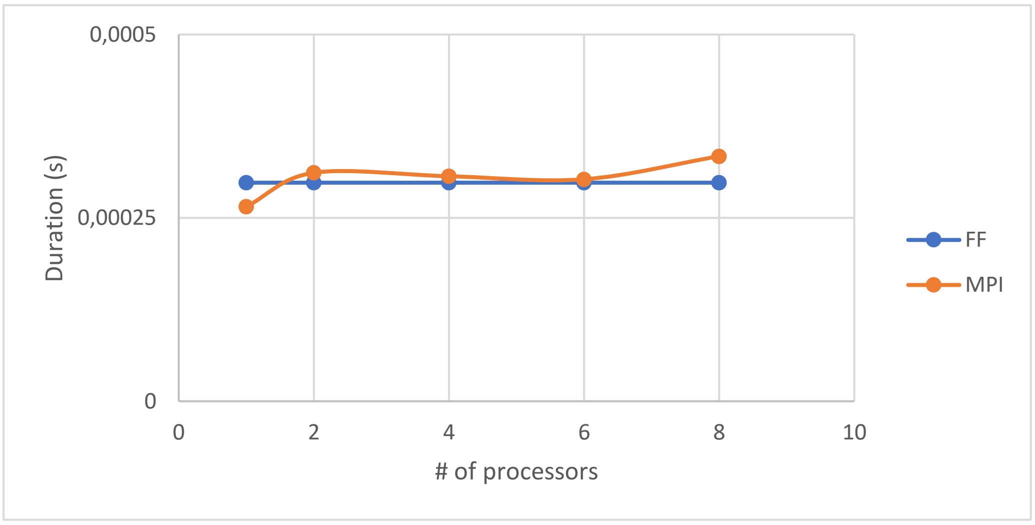


***Figure 8:*** *Total duration for 1138_bus.mtx*

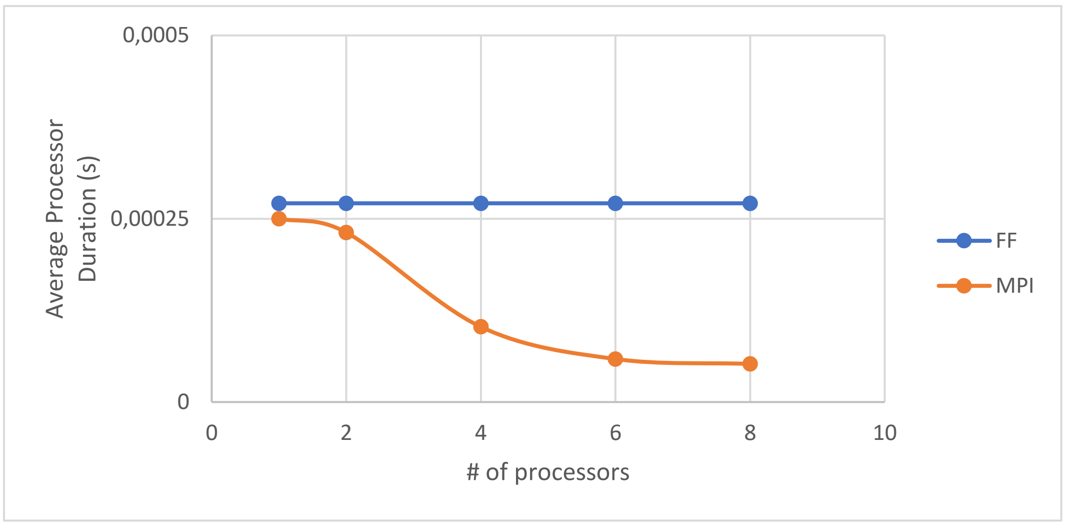


***Figure 8:*** *Average duration for 494_bus.mtx*

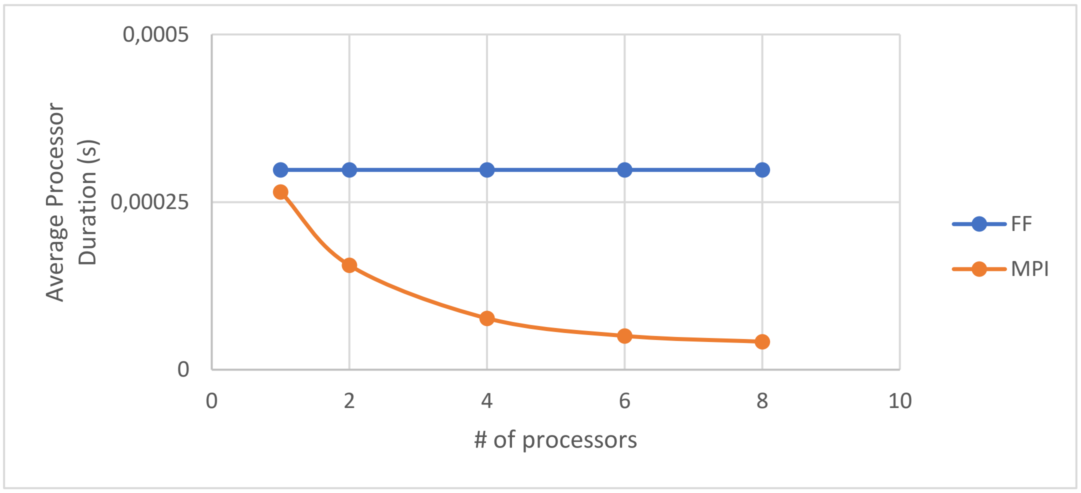


***Figure 9:*** *Average duration for 1138_bus.mtx*

These outputs reveal that:

- Due to the sequential part in MPI, total process is longer than FF. However, when the matrix size increases, MPI will run better than FF.
- In an ideal case, each processor works equally, there is a significant improvement in speedup. Even the study (Gebremedhin & Manne, 2000) concludes a linear speedup, it looks logarithmic according to these figures.
- The only communication used in this study is point to point. Conflict detection could not parallelize effectively. If collective communication is used and deadlocks are prevented, and the code is written effectively (especially the sequential part), the results will be closer to the referenced article.

- When the processor size increases, number of minimum colors could not be found correctly. This proves there is a ideal processor number depends on the amount of work as the article said.

## CONCLUSION

In this study, a parallel graph coloring algorithm from the referenced article was implemented in the MPI/C++ environment. Previously, my working space was greyfurt.ceng.metu.edu.tr, where I mainly worked on lecture-related tasks. However, I switched to using my own computer because I encountered numerous "pending status" warnings when trying to run the code. Consequently, I was able to execute the parallel code on only 8 processors.

To be honest, I believe that better parallel or sequential codes could have been written. However, as a civil engineering student, I did not give up and always tried my best. Thank you for offering this course. My key learning outcomes include improving my Linux environment and C++ skills, as well as gaining experience with parallel computing using MPI.

## REFERENCES

Brelaz D. New methods to color the vertices of a graph. Communications of the ACM 1979; 22(4):251–256.

Gebremedhin, A. H., & Manne, F. (2000). Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience, 12*(13), 1131–1146.

Grimmet GR, McDiarmid CJH. On coloring random graphs. Mathematical Proceedings of the Cambridge Philosophical Society 1975; 77:313–324.