# Parallel Graph Coloring Using MPI

## CSC 770 – Parallel Computing

Term Project

Presented by: Sanaria & Ben

### Algorithm 1: Partitioning Strategy

Parallel greedy coloring with priority-based conflict resolution

### Algorithm 2: Divide-and-Conquer Strategy

Maximal Independent Set (MIS)-based approach

Implemented in C with MPI

Graph Coloring Algorithms

# Project Overview

## Graph Coloring Algorithms

Parallel Computing - CSC 770

### Project Details

Topic chosen from the course project list: Graph Coloring

Goal: **design two different parallel algorithms** for the problem

Algorithms in this project:

- Algorithm 1: Partitioning Strategy
- Algorithm 2: Divide-and-Conquer Strategy

Main focus of this work:

- Explain how each algorithm colors the graph in parallel
- Compare their performance as the number of processes increases

# Graph Coloring Problem

## Problem Definition

**Given:** A graph $G=(V,E)$

**Assign:** A color $C(v)$ to each vertex $v \in V$

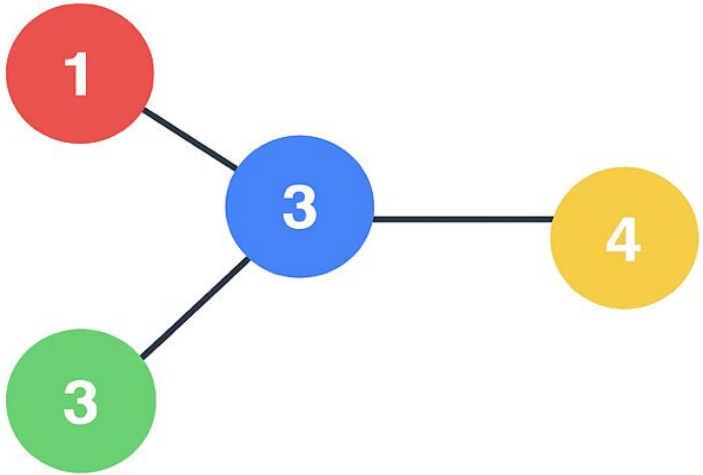
**Constraint:** If $(u,v) \in E$, then $C(u) \neq C(v)$

> **Goal:** Find a valid coloring
> Prefer fewer colors, but optimal minimum is NP-hard

### Applications:

- Register allocation in compilers
- Timetable and exam scheduling
- Frequency assignment in networks

## Example




**Valid coloring:** Each vertex has a different color from its neighbors.

**Uses 4 colors:** Not optimal, but valid.

# Why Parallel Graph Coloring?

## Problem Context

Real-world graphs can be **very large**

Sequential coloring may be **too slow** for practical applications

## Parallelism Benefits

- Faster coloring for large graphs
- Using multiple cores / processors

## Theoretical Models

PRAM Model

Parallel Random Access Machine

EREW CREW CRCW

## Practical Implementation

MPI Message Passing

Implementation model

Designing with PRAM-style thinking

↓

Implementing with MPI in C

# Background: PRAM & MPI

## PRAM

Parallel Random Access Machine

- Many processors with shared memory
- Synchronous execution model
- Our two algorithms apply PRAM-style ideas but are implemented using MPI.

### Submodels:

**EREW**
Exclusive Read Exclusive Write

**CREW**
Concurrent Read Exclusive Write

**CRCW**
Concurrent Read Concurrent Write

## MPI

Message Passing Interface

- Each process has its own memory
- Processes have ranks in MPI_COMM_WORLD

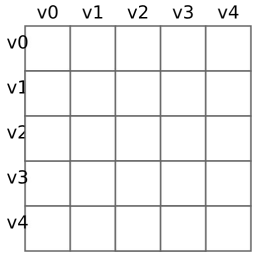
### Basic Functions:

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Recv

# Graph Representation & Data Structures

## Graph Representation

- **n** = number of vertices
- **adj[i][j]** = 1 if edge between i and j, 0 otherwise




## MPI Partitioning

Vertices divided into blocks among processes:

Process r handles vertices from **start_r** to **end_r**

## Arrays Used in Both Algorithms

**color[v]** – current color (0 = uncolored)

**priority[v]** – random weight for tie-breaking

## Extra for Divide-and-Conquer

**active[v]** – 1 if still in remaining graph, 0 if already colored

# Two Parallel Graph Coloring Strategies

## Algorithm 1: Partitioning Strategy

Vertices partitioned among processes

- Partition vertices among MPI processes
- Parallel greedy coloring with priorities
- Each process colors its assigned vertices
- Uses random priorities to break ties

## Algorithm 2: Divide-and-Conquer Strategy

Maximal Independent Sets (MIS) coloring

- Repeatedly find Maximal Independent Sets (MIS)
- Color each MIS with a new color
- Remove colored vertices from graph
- Repeat until all vertices are colored

**Both Algorithms Use:**

C + MPI | Random priorities | Communication to sync colors

# Algorithm 1: Partitioning Strategy – Intuition

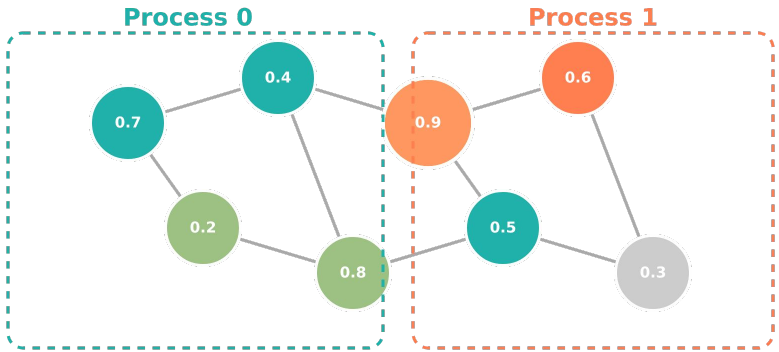
## Algorithm Intuition

Each process manages a subset of vertices in the graph, allowing for parallel processing.

### Key Steps:

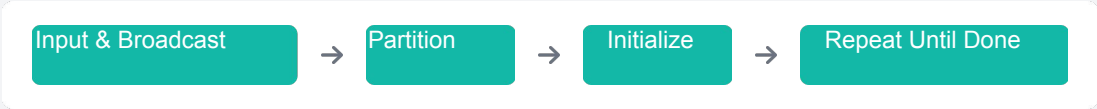
- **Local Processing:** Each process works on its assigned vertices
- **Priority Check:** Verify if vertex has highest priority among neighbors
- **Color Selection:** Choose smallest available color greedily
- **Repeat:** Continue until all vertices are colored

### Visual Intuition




**Key Insight:** Priorities prevent conflicts while allowing parallel progress.

# Algorithm 1: Partitioning Strategy – Steps




## Input & Broadcast

- Rank 0 reads n and adjacency matrix from file
- Uses MPI_Bcast to send graph to all processes

## Partition Vertices

- Compute start and end indices for each process
- Each process handles a subset of vertices

## Initialization

- For all vertices: color[v] = 0
- For all vertices: priority[v] = random(0,1)

## Repeat Until Done

- For each local uncolored vertex: Check uncolored neighbors
- If priority is highest: choose smallest unused neighbor color and assign
- Synchronize color[] across processes (e.g., MPI_Allgather) and use MPI_Reduce / MPI_Bcast

# Algorithm 1: Communication & Time Complexity

## Communication Patterns

**MPI_Bcast**
Broadcast graph size and adjacency matrix

**MPI_Allgather**
Share updated colors across all processes

**MPI_Reduce + MPI_Bcast**
Check if any vertex was colored this round

## Time Complexity Analysis

**Definitions:**

- n = vertices
- Δ = max degree
- p = number of processes
- R = number of rounds

**Local work per process:**

$$\sim (n/p) \cdot \Delta \cdot R$$

**Total time:**

$$T_1(p) \approx O((n\Delta/p) \cdot R + R \cdot comm\_cost)$$

# Algorithm 2: Divide-and-Conquer (MIS-Based) – Intuition

## Maximal Independent Set (MIS)

### Independent Set

No two vertices are adjacent (no edges between them)

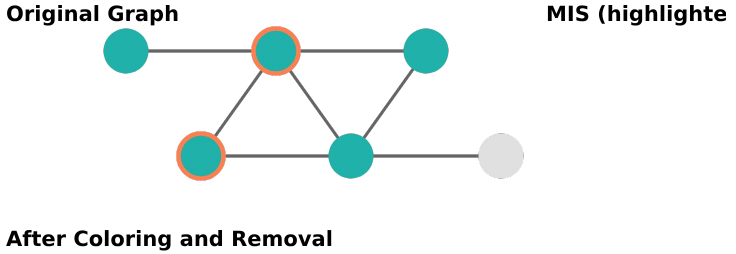
### Maximal

Cannot add more vertices without breaking independence

> In each round, we find an MIS in parallel, color its vertices, and remove them from the graph.

## Algorithm Intuition

1. Find an MIS in parallel
2. Color all vertices in MIS with current color
3. Remove colored vertices (active[v] = 0)
4. Repeat on remaining graph with new color



# Algorithm 2: Divide-and-Conquer – Steps

Initialization → While Active Vertices Remain → Assign Priorities → Select MIS → Color & Remove

## 1. Initialization

For all vertices:

- color[v] = 0 (uncolored)
- active[v] = 1 (active)

current_color = 1

## 2. While Active Vertices Remain

- Each process counts active vertices → local_active
- MPI_Reduce → global_active
- If global_active == 0 , stop

## 3. Assign Priorities

- For active vertices: priority[v] = random(0,1)
- Sync priorities with MPI_Allgather

## 4. Select MIS in Parallel

- For each local active vertex v :
- If no active neighbor has higher priority → v joins MIS

## 5. Color MIS & Remove

- If v in MIS: color[v] = current_color
- If v in MIS: active[v] = 0
- Sync color[] and active[] with MPI_Allgather

current_color++

# Algorithm 2: Communication & Time Complexity

## Communication Patterns

**MPI_Bcast**
Graph data at the beginning

**MPI_Allgather**
Synchronize: priority[], inMIS[], color[], active[]

**MPI_Reduce**
Sum of active vertices to detect termination

## Time Complexity Analysis

Parallel Complexity (high-level):

$$T_2(p) \approx O((n\Delta/p) \cdot R' + R' \cdot comm\_cost)$$

- Let R′ = number of color rounds (MIS selections)
- Each round: Work ~ (n/p)·Δ
- Plus communication for synchronization

**Why Algorithm 2 is typically slower:**

R′ > R → more rounds than Algorithm 1 → usually slower

# Experimental Setup

## Implementation

- Language: **C**
- Parallel library: **MPI**
- Platform: **Multi-core CPU**

## Graphs Used

Random graphs with vertices:

**n = 1000** | **n = 2000** | **n = 5000**

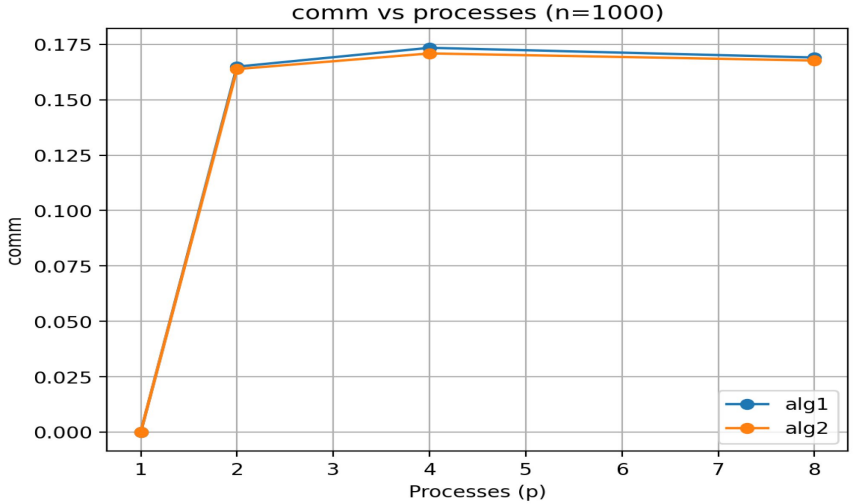
## Processes Tested

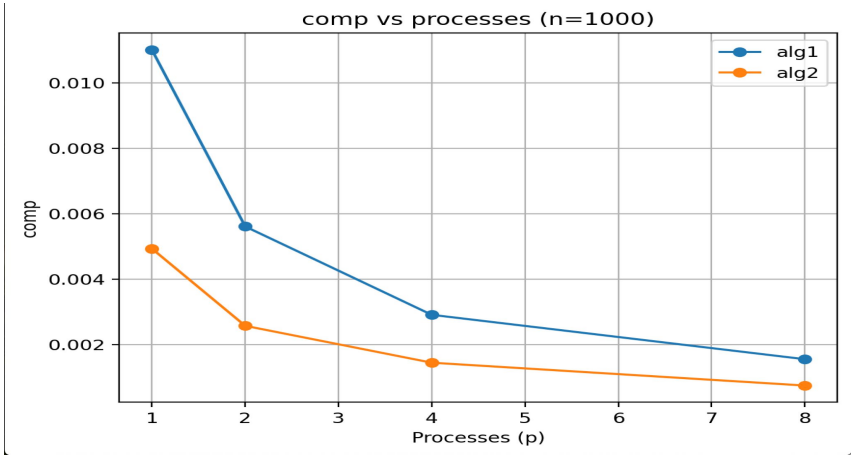
**p = 1** | **p = 2** | **p = 4** | **p = 8**

## Metrics Collected

- Execution time (seconds)
- Speedup = T(1)/T(p)
- Number of colors used

# Results – Execution Time vs Processes




## Key Observations

**Algorithm Comparison**

Partitioning Strategy shows consistently lower runtime with more processes

**Communication Overhead**

Overhead limits improvement at higher process counts

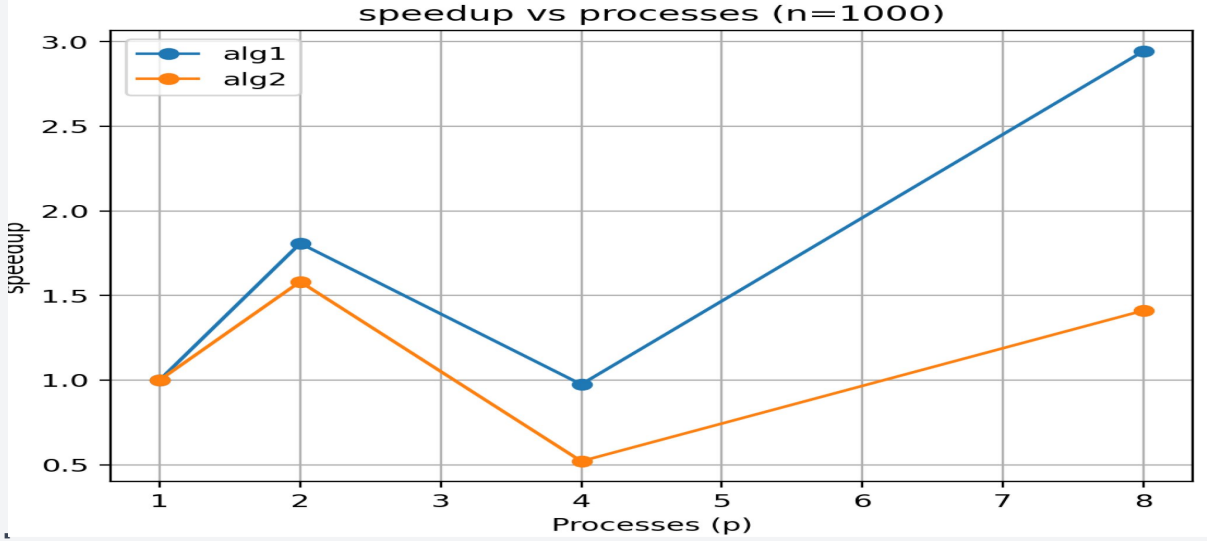
**Performance Trend**

Execution time decreases as processes increase for both algorithms

# Results – Speedup vs Processes

## Speedup Definition:

Speedup(p) = T(1)/T(p)




## Key Observations:

### Nearly Linear Speedup

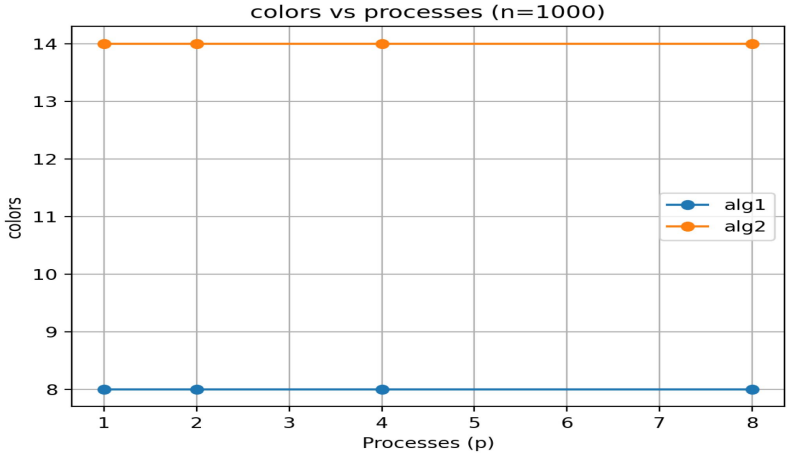
For small # of processes, both algorithms show linear speedup

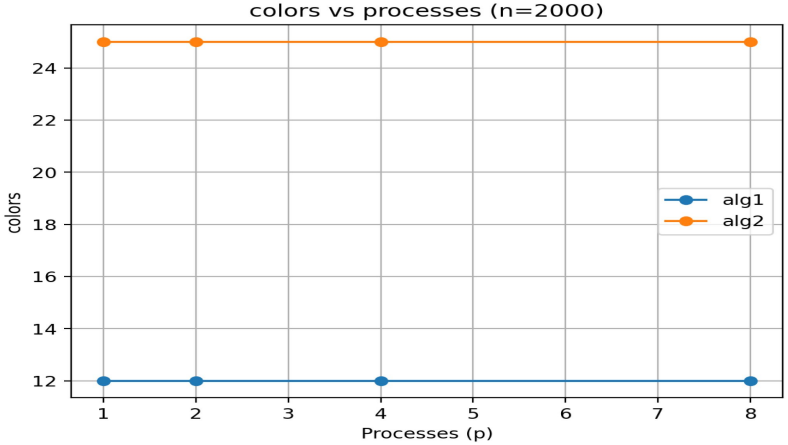
### Communication Dominance

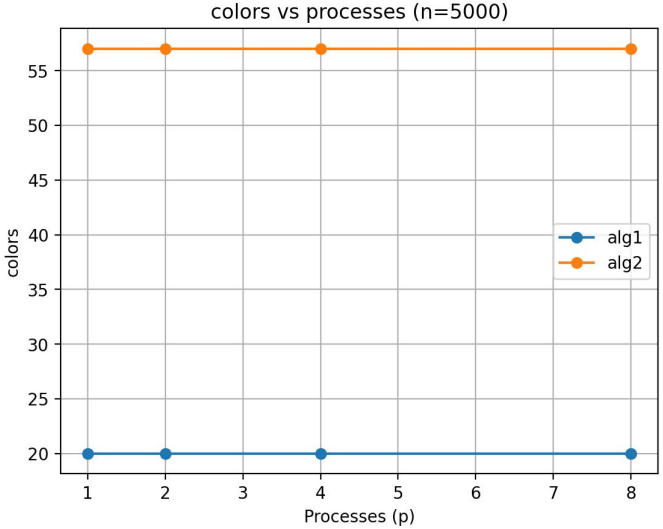
Speedup flattens as communication overhead becomes significant

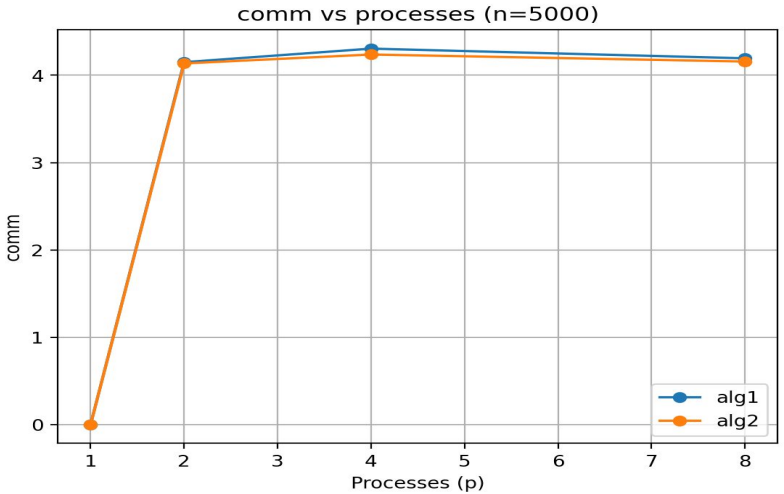
### Algorithm Comparison

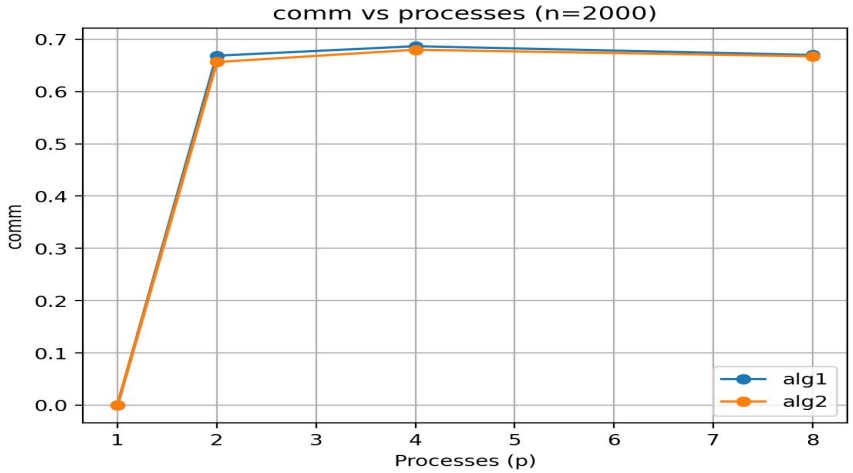
Algorithm 1 achieves better speedup than Algorithm 2

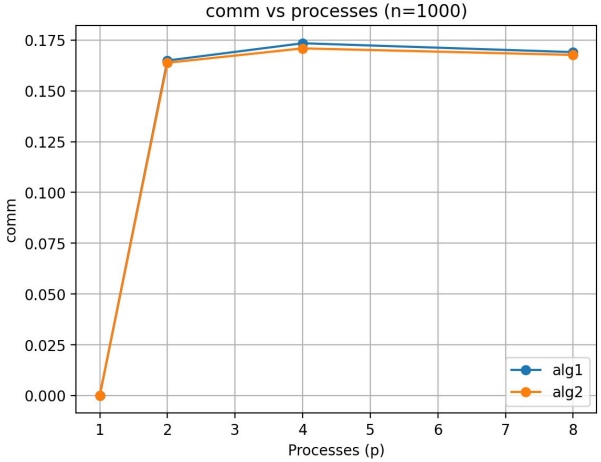

colors vs processes (n=1000)
colors
Processes (p)
alg1
alg2
14
13
12
11
10
9
8
1
2
3
4
5
6
7
8

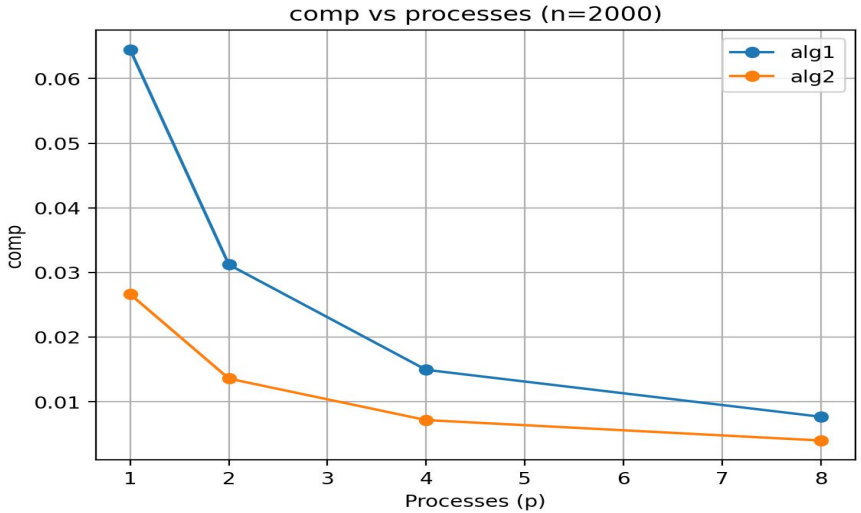

colors vs processes (n=2000)
colors
Processes (p)
alg1
alg2
24
22
20
18
16
14
12
1
2
3
4
5
6
7
8

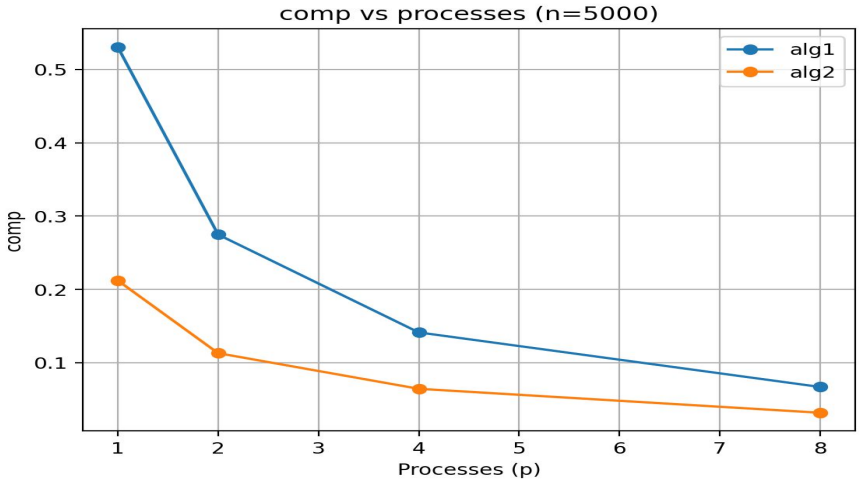

colors vs processes (n=5000)
colors
Processes (p)
alg1
alg2
55
50
45
40
35
30
25
20
1
2
3
4
5
6
7
8

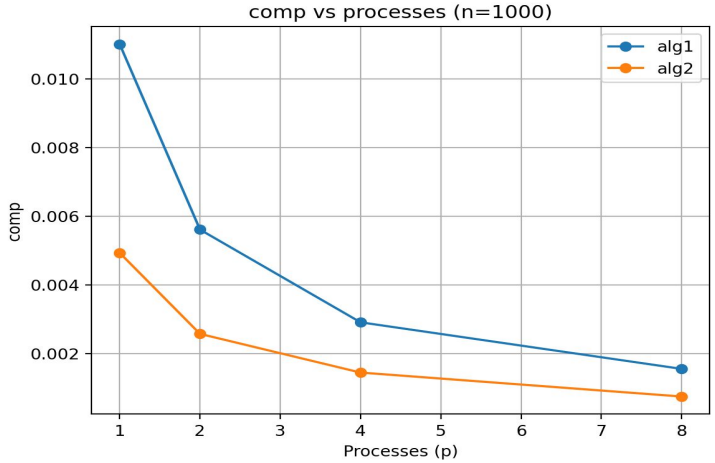

comm vs processes (n=5000)
comm
Processes (p)
alg1
alg2
0
1
2
3
4
1
2
3
4
5
6
7
8

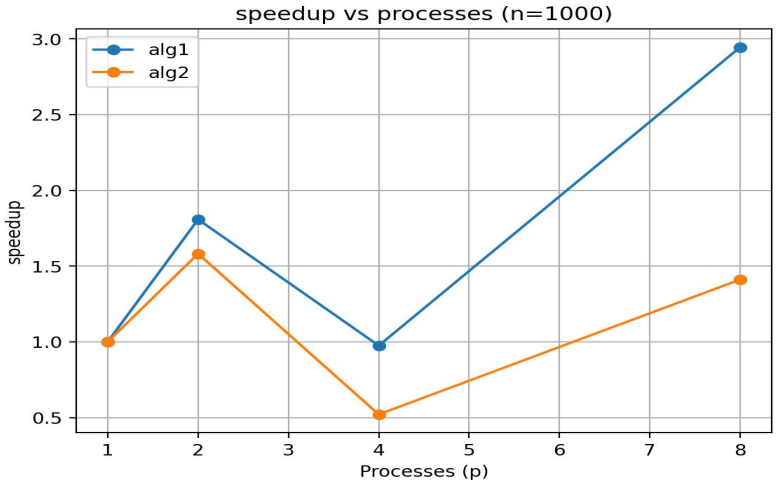

comm vs processes (n=2000)
comm
Processes (p)
alg1
alg2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
1
2
3
4
5
6
7
8

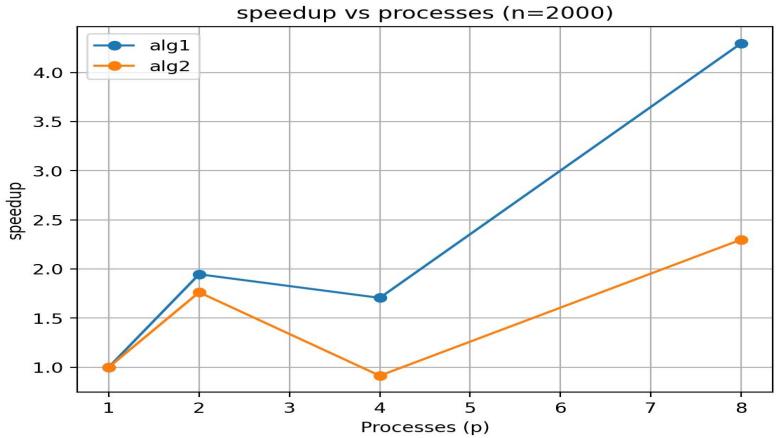

comm vs processes (n=1000)
comm
Processes (p)
alg1
alg2
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
1
2
3
4
5
6
7
8

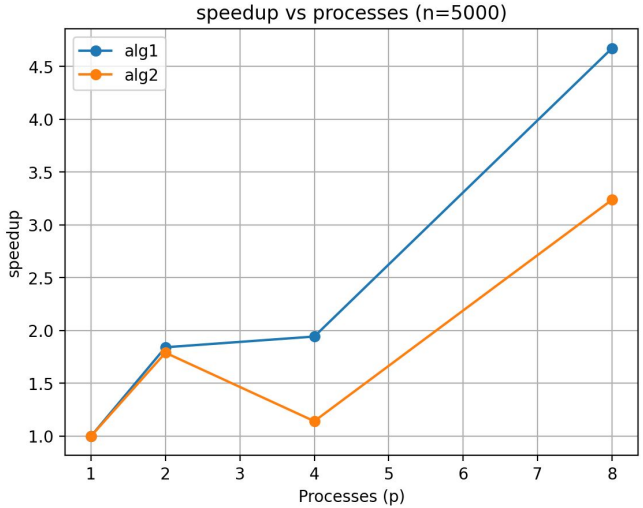

comp vs processes (n=2000)
alg1
alg2
comp
0.06
0.05
0.04
0.03
0.02
0.01
1
2
3
4
5
6
7
8
Processes (p)

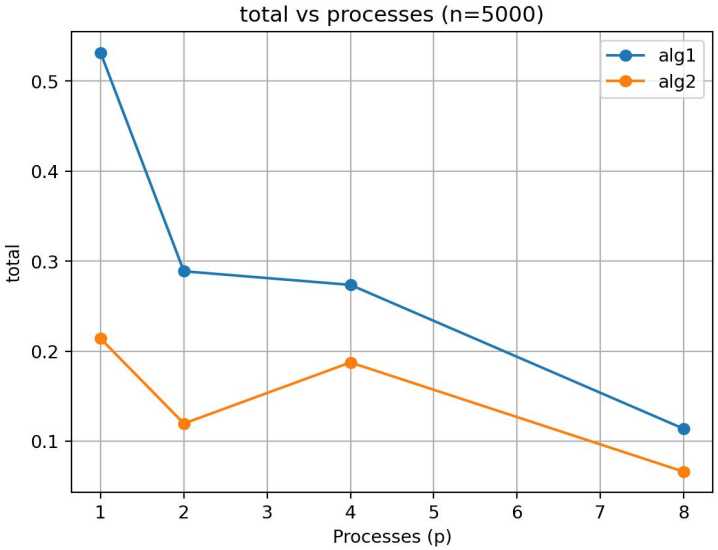

comp vs processes (n=5000)
alg1
alg2
comp
0.5
0.4
0.3
0.2
0.1
1
2
3
4
5
6
7
8
Processes (p)

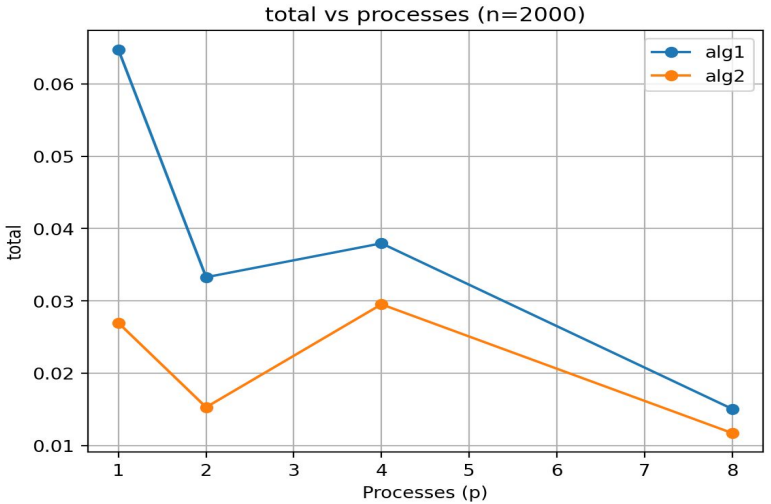

comp vs processes (n=1000)
alg1
alg2
comp
0.010
0.008
0.006
0.004
0.002
1
2
3
4
5
6
7
8
Processes (p)

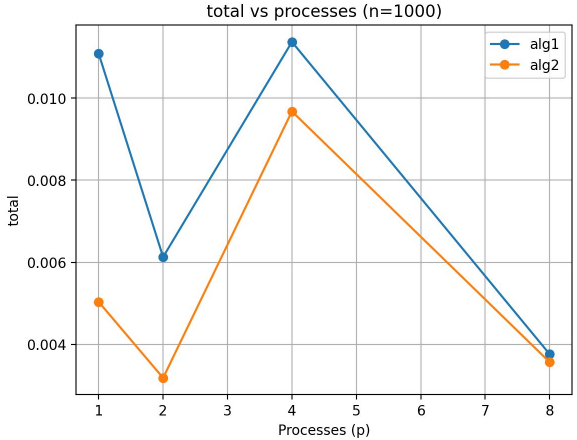

speedup vs processes (n=1000)
alg1
alg2
speedup
Processes (p)
3.0
2.5
2.0
1.5
1.0
0.5
1
2
3
4
5
6
7
8


speedup vs processes (n=2000)
alg1
alg2
speedup
Processes (p)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1
2
3
4
5
6
7
8


speedup vs processes (n=5000)
alg1
alg2
speedup
Processes (p)
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
1
2
3
4
5
6
7
8


total vs processes (n=5000)
alg1
alg2
total
Processes (p)
0.5
0.4
0.3
0.2
0.1
1
2
3
4
5
6
7
8


total vs processes (n=2000)
alg1
alg2
total
Processes (p)
0.06
0.05
0.04
0.03
0.02
0.01
1
2
3
4
5
6
7
8


total vs processes (n=1000)
alg1
alg2
total
Processes (p)
0.010
0.008
0.006
0.004
1
2
3
4
5
6
7
8

# Algorithm Comparison

## Algorithm 1: Partitioning Strategy

### Pros

- Lower number of rounds
- Usually faster in practice
- Often uses fewer colors

### Cons

- More complex local decision
- Check neighbors' colors + priorities

## Algorithm 2: Divide-and-Conquer Strategy

### Pros

- Simple concept: MIS → color → remove
- Naturally fits "divide-and-conquer" idea

### Cons

- Typically more rounds (one MIS per color)
- More colors used than greedy partitioning
- Slightly heavier communication

# Conclusions & Future Work

## Conclusions

- Both algorithms produce valid graph colorings in parallel
- Partitioning Strategy: Better execution time, Good speedup with MPI
- Divide-and-Conquer: Conceptually clean MIS-based approach, Slower but easier to reason about

## Limitations

- Use of adjacency matrix limits very large graphs
- MPI communication overhead at high process counts

## Future Work

- Use adjacency lists / CSR format for sparse graphs
- Better partitioning / load balancing
- Hybrid MPI + OpenMP model
- Test on real-world graph data

Thank you for your attention

Questions?