PDP - Team project:
Find N-Coloring of a graph

@Dan-Nicolai Badea
@Jaclina-Iana Bulat

Introduction

Graph coloring stands as a foundational problem in graph theory, requiring the assignment of colors to vertices so that adjacent vertices possess distinct colors.

These implementations concentrate on solving the graph coloring challenge via a parallelized greedy algorithm, aiming for efficient vertex coloring with a minimized color palette.

Problem Statement

The task involves coloring the graph vertices such that adjacent vertices exhibit different colors.

The objective is to achieve this coloring through a parallelized greedy algorithm, emphasizing computational efficiency and a reduction in the overall number of colors used.

Java Approach

Greedy Coloring Approach:

Colors vertices using the smallest available color not used by neighboring vertices.

Parallelization with ExecutorService:

Implements Java's ExecutorService and Callable for parallelizing the coloring process. Ensures thread safety through synchronized blocks during color assignment.

Java Solution

The Java solution offers a practical implementation of the graph coloring algorithm.

The GraphColoring class manages the graph, adds edges, and coordinates the coloring process.

Parallel processing is achieved using Java's ExecutorService to concurrently assign colors to vertices. The algorithm is demonstrated on a sample graph, aiming to enhance performance for larger graphs.

Rust Approach

Greedy Coloring Approach:

Colors vertices using the smallest available color not used by neighboring vertices.
Prioritizes computational efficiency, without guaranteeing optimality.

Parallelization with MPI:

Uses the mpi crate for parallel processing.

Each MPI process colors a subset of graph vertices, with communication for synchronization.

Rust Solution

The Rust solution provides a practical implementation of a parallel graph coloring algorithm.

The GraphColoring struct manages the graph, handles edge additions, and oversees the coloring process.

Parallel processing is achieved through the Message Passing Interface (MPI), enabling efficient vertex coloring across multiple processes.

The program runs with multiple processes, each representing a graph vertex.

Vertices collaborate through MPI communication to determine and assign colors, ensuring adjacent vertices have distinct colors.

Conclusions

The Java implementation employs a parallelized greedy algorithm for graph coloring, using a fixed-size thread pool for concurrent execution. The algorithm assigns colors to vertices while ensuring no neighboring vertices share the same color. The implementation incorporates thread synchronization to manage shared data and prevent race conditions.

On the other hand, the Rust implementation uses the Message Passing Interface (MPI) for parallel processing. The program is designed to run on multiple processes, with each process responsible for coloring a subset of vertices. MPI communication is employed to exchange color information between processes, ensuring that neighboring vertices across different processes are properly colored without direct shared memory.

In the Java implementation, the ExecutorService with a fixed thread pool is used to parallelize the coloring process. Each vertex is assigned a callable task that independently determines its color based on neighboring vertices. Synchronization is achieved through a synchronized block to prevent conflicts in color assignment.

In the Rust implementation, the MPI environment is initialized, and each process independently colors its assigned vertices. Communication between processes is handled using MPI point-to-point communication. The root process (rank 0) coordinates the overall coloring process by initiating communication with other processes, exchanging color information, and ensuring a proper coloring of the entire graph.

In conclusion, while both implementations aim to parallelize the graph coloring process, they differ in their approaches. The Java implementation uses thread pooling and synchronization for parallelism, whereas the Rust implementation leverages MPI for distributed parallelism across separate processes.

Results:

Same 5 vertices, 5 edges and 3 colors

On average:

Rust with MPI: 3.3ms
Java with ThreadPool: 24ms