**Documentation for Graph Coloring Problem with Parallelization**

**Overview**

The code provided implements a solution to the **Graph Coloring Problem** where the objective is to assign a color to each node of a graph such that no two adjacent nodes share the same color. The solution utilizes both **backtracking** and **parallelization** to improve the performance of finding a valid coloring, especially for larger graphs. The code also employs a **lock-based synchronization** mechanism to handle concurrent updates to the global coloring solution.

**Algorithms Used**

1. **Backtracking Algorithm (Recursive)**
   * The algorithm used for coloring the graph is a **backtracking algorithm**. This algorithm explores all possible color assignments for the nodes of the graph, but it stops early if an invalid coloring is encountered, pruning the search space.
   * The process is recursive. At each step, a node is assigned a color, and the algorithm checks if the coloring is valid by ensuring that no two adjacent nodes share the same color.
   * The algorithm backtracks when it encounters an invalid coloring and tries the next color for the current node.
   * Once a valid coloring for all nodes is found, it is stored and returned as the solution.
2. **Multi-Threading for Parallelization**
   * The algorithm is parallelized using **multi-threading** to explore different colorings concurrently, which can significantly reduce the overall computation time.
   * A thread is spawned for each valid color for a node, and these threads run the getColoredGraphRecursive function in parallel to explore possible colorings.
   * The number of threads is limited by a given **threadsNumber** parameter, and once this limit is reached, the remaining color possibilities are explored sequentially.

**Synchronization Mechanism**

Since the algorithm uses multi-threading, synchronization is necessary to prevent race conditions and ensure thread safety when modifying shared data. The following synchronization techniques are used:

1. **Locking with ReentrantLock**
   * The ReentrantLock is used to ensure that only one thread can modify the global coloring solution (codes) at a time.
   * The lock is acquired before modifying the codes vector and released afterward, ensuring that no other thread can concurrently change the solution during that period.
2. **Atomic Integer for Thread Count**
   * The number of available threads is tracked using an AtomicInteger (called threadsNumber).
   * This ensures that threads can decrement the count of available threads atomically, which avoids inconsistencies in the number of threads being used concurrently.
3. **Thread Joining**

* The algorithm ensures that all spawned threads complete before proceeding by calling the join() method on each thread after they have been started.
* This guarantees that all parallel work is finished before the function proceeds with further operations, such as exploring valid colors sequentially.

**Performance Measurements**

The program includes timing code that measures the execution time for the graph coloring process. The following timing approach is used:

1. **Start Time**: The execution time is measured starting from the moment the getColoredGraph function is called.
2. **End Time**: After the graph coloring computation is complete (either a solution is found or an exception is thrown), the end time is recorded.
3. **Execution Time**: The total execution time is calculated by subtracting the start time from the end time and converting the result to milliseconds.

**Code Structure and Flow**

1. **Main.java**
   * This file contains the main function that creates different graph examples and sets the parameters (like the number of threads and colors). It then calls the getColoredGraph function to compute the solution and prints the execution time.
2. **Graph.java**
   * This class represents the graph structure, where nodes are stored in a map, and edges between nodes are represented using a Set<Integer>.
   * The class includes methods to add edges, check if an edge exists between two nodes, and generate a random graph for testing purposes.
3. **Colors.java**
   * This class represents the set of available colors. Each color is assigned a unique integer code, and the class allows adding colors and retrieving the names of colors for a given list of color codes.
4. **GraphColoring.java**
   * This class contains the core logic for graph coloring. The getColoredGraph method coordinates the graph coloring process, while the recursive getColoredGraphRecursive method performs the backtracking with multi-threading.
   * The method uses AtomicInteger to track the number of available threads and ReentrantLock to synchronize access to the shared solution (codes).

**Performance Considerations**

* The parallelization significantly reduces the runtime for larger graphs, especially when the number of available threads is high. However, if the number of threads exceeds the number of available processors, it may lead to diminishing returns due to context-switching overhead.
* The performance is also affected by the graph's structure, as dense graphs with many edges will have more constraints, making it harder to find a valid coloring quickly.

**Conclusion**

* This solution efficiently solves the graph coloring problem by combining **backtracking** and **parallelization**. The parallel approach speeds up the solution search, especially for large graphs. Synchronization using **locks** ensures thread safety when accessing shared data, and performance measurements help track the efficiency of the algorithm in real-time.