# Hamiltonian Cycle Search: Parallel Algorithm

## Introduction

This document provides an overview of the implementation of a parallel algorithm to find a Hamiltonian cycle in a directed graph. The algorithm uses recursive backtracking, enhanced with parallelization through the Fork/Join framework in Java.

## Algorithm Description

The algorithm starts from a fixed vertex and attempts to construct a Hamiltonian cycle by exploring all neighbors of the current vertex. At each step, the algorithm splits the work into multiple threads to explore unvisited neighbors concurrently. This parallelization is achieved using Java's RecursiveTask and ForkJoinPool classes.

Synchronization is managed through the following mechanisms:  
1. An AtomicBoolean (`foundHamiltonianCycle`) ensures that threads stop searching once a cycle is found.  
2. A ReentrantLock protects updates to the shared `output` list that stores the Hamiltonian cycle.  
3. Copies of the `visited` list are created for each thread to avoid conflicts.

## Synchronization

To ensure thread safety and correctness, the following synchronization techniques were used:  
- AtomicBoolean: Allows threads to communicate the discovery of a Hamiltonian cycle safely.  
- ReentrantLock: Ensures that updates to the shared output list are performed atomically.  
- Thread-local Copies: Each thread operates on a copy of the `currentPath` and `visited` lists, preventing interference between threads.

## Performance Measurements

The performance of the parallel algorithm was measured and compared against a sequential implementation. Key metrics include execution time and scalability with increasing graph size. Results show that the parallel approach significantly reduces execution time for larger graphs but may incur overhead for small graphs due to task management costs. Detailed measurements can be found in the appendix.

## Conclusion

This implementation demonstrates an efficient parallel approach to finding Hamiltonian cycles in directed graphs. By leveraging Java's Fork/Join framework, the algorithm achieves significant speedup while ensuring correctness through robust synchronization mechanisms.