# Parallelization of the Floyd-Warshall Algorithm for All-Pairs Shortest Path

## Abstract

This project explores parallel implementations of the Floyd-Warshall algorithm for solving the all-pairs shortest path problem in weighted graphs. The classic Floyd-Warshall algorithm, while elegant in its simplicity, faces performance limitations due to its O(V³) time complexity. This project implements and analyzes both shared-memory parallelization using OpenMP and distributed-memory parallelization using MPI, with the implementation done in C++ to leverage its strong type system and performance characteristics.

The parallel implementations were tested on graphs of varying sizes (100 to 1000 vertices) and densities (0.1 to 0.5), with performance metrics collected through comprehensive unit testing and automated performance analysis. Results demonstrate significant speedup, with the OpenMP implementation achieving up to 7x speedup with 8 threads on larger graphs (N=1000), while the MPI implementation showed comparable performance improvements. Performance analysis revealed that both implementations scale effectively with graph size, though efficiency tends to decrease with higher thread counts due to communication overhead. These conclusions provide insights into the practical aspects of parallelizing dynamic algorithms and the trade-offs between different parallel programming paradigms.

## 1. Introduction

Graph algorithms play a crucial role in modern computing, from optimizing network routing to analyzing social networks. Among these, the all-pairs shortest path (APSP) problem—finding the shortest paths between every pair of vertices in a weighted graph—is a fundamental challenge. The Floyd-Warshall algorithm provides an elegant solution to this problem through dynamic programming, but its cubic time complexity O(V³) becomes a significant bottleneck as graph sizes increase.

In real-world applications, graphs routinely contain thousands of vertices, making the need for faster computation critical. For example, in network routing, social network analysis, and biological network studies, the ability to quickly compute shortest paths between all nodes can significantly impact system performance. This research addresses this challenge by exploring parallel implementations of the Floyd-Warshall algorithm.

This study implements two distinct parallel approaches: a shared-memory implementation using OpenMP and a distributed-memory implementation using MPI. The choice of C++ as the implementation language was deliberate, offering both high performance and strong type safety while maintaining compatibility with industry-standard parallel programming frameworks. The chosen research topic focuses on quantifying the performance benefits and scaling characteristics of these parallel implementations compared to the serial version.

The significance lies not only in its practical application to graph processing but also in its exploration of different parallelization strategies for algorithms with regular computation patterns. This project covers both shared and distributed memory approaches, providing insights into the trade-offs and considerations necessary when parallelizing dynamic programming algorithms. Working as a solo developer on this project was a challenge but it allowed for a deeper understand of the problem space and the data obtained.

## 2. Background

The Floyd-Warshall algorithm, introduced by Robert Floyd in 1962 (building upon work by Stephen Warshall), is a dynamic programming solution for the all-pairs shortest path problem. The algorithm iteratively improves path estimates by considering whether going through an intermediate vertex k provides a shorter path between vertices i and j. This elegant approach has a time complexity of O(V³) and space complexity of O(V²), where V is the number of vertices in the graph.

The regular structure of the Floyd-Warshall algorithm makes it a strong candidate for parallelization. Previous research has explored various parallel implementations, with early work focusing on specialized hardware and more recent studies examining both shared and distributed memory approaches. Shared memory implementations typically parallelize the innermost loop of the algorithm, while distributed approaches must carefully manage the communication of path information between processors.

Existing research has shown promising results in parallelizing the Floyd-Warshall algorithm. However, many studies focus on either shared memory or distributed approaches in isolation, with limited comparison between the two paradigms. Additionally, the impact of graph characteristics such as density on parallel performance is often underexplored. This project aims to address these gaps by providing a comprehensive comparison of OpenMP and MPI implementations across varying graph sizes and densities.

The choice of C++ for implementation allowed me to leverage both low-level optimizations and high-level abstractions. I was able to take advantage of the language's strong type system and template capabilities to help with the creation of efficient, reusable graph data structures. Its direct support for OpenMP and MPI also allowed for clean integration of parallel processing capabilities. Modern C++ features also enable easy unit testing through frameworks like Google Test, which was helpful during incremental development to ensure iterative correctness.

## 3. Methods

### 3.1 Implementation

The implementation strategy focused on creating maintainable, well-tested parallel versions of the Floyd-Warshall algorithm. The core graph data structure was implemented in C++, using a 2D adjacency matrix representation to optimize for cache locality and enable efficient parallel access patterns. The base Graph class provides essential operations such as edge manipulation and random graph generation with specified vertex counts and densities.

Three distinct implementations of the algorithm were developed:

1. A serial implementation serving as the performance baseline
2. A shared-memory parallel implementation using OpenMP
3. A distributed-memory parallel implementation using MPI

The OpenMP implementation parallelizes the algorithm's innermost loop, leveraging shared memory architecture to minimize communication overhead. The MPI implementation distributes graph partitions across multiple processes, implementing efficient communication patterns for boundary data exchange.

Unit testing was implemented using the Google Test framework, ensuring correctness across all implementations and helping to catch early bugs. Test cases covered various scenarios including:

* Basic graph operations
* Edge case handling
* Performance validation
* Correctness verification against the serial implementation

### 3.2 Experimental Setup

Performance testing was conducted using graphs of varying sizes (100, 200, 500, and 1000 vertices) and densities (0.1, 0.3, and 0.5). The OpenMP implementation was tested with 1, 2, 4, and 8 threads, while the MPI implementation utilized 4 processes. Performance metrics were automatically collected using a custom testing framework that measured:

* Execution time
* Speedup relative to serial implementation
* Parallel efficiency

Data collection was automated through a comprehensive performance testing suite, with results stored in CSV format. Visualization scripts using Python with matplotlib and seaborn were developed to analyze and present the results. This automated approach ensured consistent and reproducible performance measurements across all test configurations.

## 4. Results and Discussion

### 4.1 Performance Analysis

Performance analysis of the parallel implementations revealed significant improvements over the serial version, particularly for larger graph sizes. Let's examine the key findings through several performance metrics.

A graph with a line and a red line

Description automatically generated with medium confidenceFigure 1: Speedup comparison between OpenMP and MPI implementations for N=1000 vertices

The OpenMP implementation (pink lines) shows different speedups for different scenarios, reaching a maximum of approximately 7x speedup with 8 threads. There appears to be some variation in OpenMP performance, likely due to different graph densities. The MPI implementation (red square) with 4 processes achieves about 4.7x speedup, performing competitively with OpenMP at that process count. The black dashed line represents ideal linear speedup, and we can see both implementations fall somewhat below this theoretical maximum, which is expected due to communication overhead and memory access patterns.

A graph showing the number of threads

Description automatically generated Figure 2: Strong scaling efficiency for OpenMP implementation across different problem sizes

The strong scaling efficiency results (Figure 2) demonstrate how parallel efficiency varies with problem size and thread count. For N=1000 (green line), the implementation maintains high efficiency (>90%) up to 4 threads, after which there's a more pronounced decline. Medium-sized problems (N=500, brown line) show a more gradual efficiency decrease, maintaining around 85% efficiency with 8 threads. Smaller problems (N=200, pink line) exhibit the steepest efficiency decline, dropping to approximately 50-55% at 8 threads. This behavior is typical of parallel implementations, where larger problem sizes better amortize the overhead of parallel execution.

This efficiency pattern aligns well with what we observed in the speedup analysis (Figure 1), providing additional insight into why the speedup curves deviate from linear scaling, particularly at higher thread counts.

A graph showing a number of verifies

Description automatically generated Figure 3: Execution time comparison across implementations and problem sizes

The execution time comparison (Figure 3) clearly demonstrates the impact of parallelization as problem size increases. The y-axis uses a logarithmic scale, showing execution times from 1 to 10,000 milliseconds. The serial implementation (pink line) shows the steepest increase in execution time as the number of vertices grows, reaching nearly 7000ms for N=1000. Both parallel implementations show significantly better scaling: the OpenMP implementation with 8 threads (brown line) and MPI with 4 processes (green line) maintain lower execution times across all problem sizes, with the difference becoming more pronounced at larger scales. For N=1000, both parallel implementations reduce execution time to approximately 1000-1500ms, representing a substantial improvement over the serial version.

A graph with a green line

Description automatically generated

Figure 4: Impact of graph density on OpenMP performance (N=1000)

The impact of graph density on OpenMP performance (Figure 4) shows execution times for a 1000-vertex graph across different density values. The near-overlapping lines for densities 0.1, 0.3, and 0.5 demonstrate that graph density has minimal impact on execution time. All three density configurations show similar scaling patterns, with execution time decreasing from approximately 7000ms with one thread to around 1000-1500ms with 8 threads. The slight divergence at higher thread counts suggests marginally better performance for denser graphs when using 8 threads.

### 4.2 Discussion

The performance results reveal several interesting patterns. First, the OpenMP implementation's efficiency begins to plateau at higher thread counts, particularly for smaller graphs. This behavior likely results from increased synchronization overhead and memory bandwidth limitations. The MPI implementation shows strong performance with 4 processes, though the communication overhead becomes more noticeable with denser graphs.

Graph density has a measurable but relatively minor impact on parallel performance (Figure 4). This suggests that the regular structure of the Floyd-Warshall algorithm helps maintain consistent performance across different graph characteristics.

### 5. Conclusion

This research project explored the parallelization of the Floyd-Warshall algorithm through both shared-memory (OpenMP) and distributed-memory (MPI) implementations. The performance analysis demonstrates that both parallel approaches achieve significant speedup over the serial implementation, particularly for larger graph sizes. The OpenMP implementation achieved up to 7x speedup with 8 threads on graphs with 1000 vertices, while the MPI implementation showed comparable performance improvements with 4 processes.

The strong scaling efficiency analysis revealed that larger problem sizes maintain better parallel efficiency, with the N=1000 case maintaining over 90% efficiency up to 4 threads. The investigation of graph density's impact showed that the Floyd-Warshall algorithm's regular structure leads to consistent performance across different graph densities, suggesting that the parallel implementations can be effectively applied to a wide range of graph types.

Several key insights emerged from this study:

* The parallel implementations provide substantial performance benefits for larger graphs
* Synchronization overhead becomes significant at higher thread counts
* Graph density has minimal impact on parallel performance
* The regular structure of the Floyd-Warshall algorithm makes it well-suited for parallelization

Future work could explore hybrid OpenMP/MPI implementations and investigate more sophisticated load balancing strategies. Additionally, the development of cache-optimized implementations could potentially further improve performance for larger graph sizes.