Multicore Project Final Report

Course: Multicore Processors -- Architecture & Programming
Professor: Mohamed Zahran
Semester: Fall 2025

Project Title: Parallelization of Shortest Distance Graph Algorithms

Group Members: Gitesh Chinawalkar, Akash Belide
NetIDs: gc3410, ab11903

Abstract

Graph algorithms are a foundational component of scientific computing, networking, data mining, and optimization. As graph sizes grow, classical sequential algorithms become performance bottlenecks due to memory irregularity and limited spatial locality.

In this project, we implement and evaluate parallel versions of three classical weighted shortest path algorithms: Bellman Ford, Dijkstra, and Floyd Warshall. These algorithms represent three different computational patterns: fine-grained relaxation across edges, iterative selection with adjacency list relaxations, and dense matrix-based dynamic programming.

Using OpenMP on a shared-memory multicore system, we apply per-vertex locks for Bellman Ford, a reduction-based minimum search for Dijkstra, and row-parallel updates for Floyd Warshall. Experimental results collected on the NYU CIMS Crunchy5 server show that coarse-grained, compute-heavy algorithms achieve strong speedup, while algorithms that involve irregular memory access and heavy synchronization often do not. The results highlight the importance of selecting algorithms that expose meaningful parallelism with low overhead rather than simply increasing the number of threads.

Overall, our findings demonstrate that shared-memory parallelization is effective only when algorithmic structure aligns with regular computation and minimizes synchronization, emphasizing that scalability depends more on algorithmic behavior than thread count alone.

TABLE OF CONTENTS

1. Introduction

Graph algorithms play a central role in many modern computing tasks, including routing, network analysis, scientific simulations, and large data processing systems. As real-world datasets continue to grow, the cost of sequential graph processing becomes increasingly significant. Multicore processors provide an opportunity to reduce runtime through parallel execution, but achieving good speedup depends greatly on the structure of the algorithm and the memory access patterns it creates. Some algorithms naturally expose large independent units of work, while others contain fine-grained dependencies that limit the amount of useful parallelism.

The goal of this project is to study these differences by parallelizing and analyzing three classic weighted shortest path algorithms: Bellman Ford, Dijkstra, and Floyd Warshall. Each algorithm represents a different category of computational behavior. Bellman Ford performs repeated relaxation over the entire edge list and must coordinate frequent updates to shared distances. Dijkstra performs a combination of global minimum selection and local relaxations, which creates a different mix of sequential and parallel work. Floyd Warshall is a dense dynamic programming algorithm that applies a predictable update pattern across an entire matrix, making it a strong candidate for coarse-grained parallelism.

We implement sequential and OpenMP-based parallel versions of all three algorithms and evaluate them on large synthetically generated graphs. The experiments are designed to highlight how differences in algorithmic structure affect performance on multicore hardware. By comparing results across these three algorithms, we gain practical insight into how memory locality, synchronization, and workload distribution influence scalability. The broader objective is to understand not only how to parallelize an algorithm, but also how to choose algorithms that are well suited for parallel execution.

2. Literature Survey

Parallel graph processing has been widely studied across shared memory, distributed memory, and GPU based systems. A recurring theme in prior work is that the amount of useful parallelism in a graph algorithm depends strongly on its structure, the representation of the graph, and the regularity of its memory access patterns. Algorithms that involve dense and predictable computations often scale well on multicore processors. Algorithms that involve sparse and irregular traversal patterns are more difficult to parallelize effectively.

2.1 Dense and Compute Heavy Algorithms

Dense algorithms such as Floyd Warshall, PageRank, and matrix based all pairs shortest paths benefit from a high ratio of computation to memory access. The operations in these algorithms follow a structured pattern across rows or blocks of a matrix, which allows coarse grained parallelism without frequent synchronization. Hong et al. [1] show that dense kernels with regular access patterns can achieve strong speedup because threads operate on independent segments of data with good cache locality. Lee and Kim [2] argue that memory organization and locality play a major role in scaling such workloads.

2.2 Irregular Graph Traversal Algorithms

Traversal based algorithms such as BFS, SSSP, and Connected Components follow level by level or frontier-based expansions. Although these algorithms expose parallelism in each frontier, the amount of work per level is often unpredictable. Leiserson and Schardl [3] describe techniques for work efficient parallel BFS, but they also note that load imbalance and coordination costs remain significant challenges. Bader and Madduri [4] demonstrate that contention on shared data structures and nonuniform frontier sizes often limit the scalability of traversal algorithms on shared memory systems.

2.3 Relaxation Based SSSP Algorithms

Algorithms that repeatedly relax edges, such as Bellman Ford, contain fine grained dependencies and generate heavy write contention on distance arrays. Meyer and Sanders [5] show that traditional Bellman Ford is difficult to parallelize because threads frequently attempt to update the same vertices. Harish and Narayanan [6] observe similar behavior on

GPUs, where atomic updates and irregular memory access stall performance. Several techniques such as delta stepping and chaotic relaxation attempt to increase available parallelism, but these approaches require sophisticated data structures or distributed processing frameworks that are not suitable for basic OpenMP environments.

2.4 Why Existing Work Does Not Directly Apply to This Project

Many of the high-performance solutions in the literature rely on specialized hardware features or advanced frameworks. Several GPU based designs require warp level primitives and carefully engineered memory layouts. Distributed systems such as Pregel [7] rely on message passing and partitioned computation, which is not relevant to a single node multicore environment. Even shared memory graph frameworks such as GraphLab [9] rely on lock free data structures and asynchronous updates that exceed the scope of a course project.

For this project, the focus is on practical and portable OpenMP implementations. By studying three algorithms with very different computational properties, we show why some classical graph algorithms achieve meaningful speedup on multicore systems while others do not. The literature highlights that scalability is not determined only by the choice of a parallel programming model but also by the inherent structure of the algorithm and the organization of its data.

3. Proposed Idea

The main goal of this project is to understand how different graph algorithms behave on multicore systems and to design OpenMP based parallel versions that reflect their computational structure. We focus on three classical weighted shortest path algorithms: Bellman Ford, Dijkstra, and Floyd Warshall. These three algorithms represent fundamentally different patterns of work: sparse edge relaxations, iterative vertex selection, and dense matrix based dynamic programming. By parallelizing each algorithm in a manner that respects its structure, we can study why some algorithms scale on multicore processors while others do not.

To support consistent and fair comparison, all algorithms share the same input graph format. A unified graph generator creates directed graphs of various sizes with controlled degree distributions and weight ranges. Each algorithm then converts the input into the internal representation that best fits its computation. Bellman Ford and Dijkstra use adjacency lists and edge lists that support sparse relaxations, while Floyd Warshall uses a dense matrix that defines the complete set of pairwise distances.

The proposed idea is divided into three algorithm specific designs.

3.1 Dijkstra Parallelization

The classical array-based Dijkstra algorithm selects the unvisited vertex with the smallest tentative distance and relaxes its outgoing edges. The minimum selection step dominates the runtime for large graphs, because it requires a full scan of all vertices in every iteration. The relaxation phase must remain sequential, since relaxations depend on the selected vertex.

The parallel design accelerates the minimum selection step by using OpenMP to divide the scan across threads. Each thread computes a local minimum over its portion of the vertex set, and a critical section combines these results into a single global minimum.

Code abstract:

```
#pragma omp parallel
{
    compute local minimum over assigned range;
    #pragma omp critical
    update global minimum if needed;
}
```

Once the next vertex is selected, the relaxation of its outgoing edges is performed sequentially, preserving correctness. Although only part of the algorithm is parallel, parallelizing the dominant minimum selection step provides meaningful speedup for large values of n. This design demonstrates how partial parallelization can still be effective when the parallel region aligns with the algorithm's cost structure.

3.2 Bellman–Ford Parallelization

Bellman Ford repeatedly relaxes all edges in the graph. In the sequential version, each iteration scans the entire edge list and updates the distance array when a shorter path is found. This update operation creates write conflicts in a parallel environment, because many edges may target the same vertex. To maintain correctness while allowing parallel relaxation, our implementation assigns an OpenMP lock to each vertex. During relaxation, a thread only acquires the lock for the vertex whose distance it intends to update.

This design exposes parallel work at the level of individual edges. Each iteration is implemented as an OpenMP parallel for loop that processes different segments of the edge list concurrently.

Code abstract:

```
#pragma omp parallel for reduction(||:updated)
for (each edge e in edges) {
    long long candidate = dist[e.u] + e.w;
    if (candidate < dist[e.v]) {
        acquire lock for vertex v;
        update dist[v];
        release lock;
        updated = true;
    }
}
```

The use of per vertex locks avoids global synchronization and reduces contention, but the algorithm still performs many fine-grained updates. These characteristics make Bellman Ford a useful example of the limitations of parallelizing irregular relaxation-based algorithms on shared memory systems.

3.3 Floyd Warshall Parallelization

Floyd Warshall computes all pairs shortest paths over a dense n by n distance matrix. Its triple nested loop structure has a natural separation of parallel and sequential work. The outer loop on k represents the intermediate vertex and must remain sequential. For a fixed value of k, however, the updates to each row i are independent. This allows the middle loop on i to be parallelized without any synchronization.

The parallel version follows the structure:

```
for (int k = 0; k < n; k++) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        update row i using dist[i][k] and dist[k][j];
    }
}
```

Each thread updates an entire row of the distance matrix. Since no two threads write to the same row during a given iteration of k, no locks or atomics are required. This coarse grained parallelism, combined with the high arithmetic intensity of the algorithm, makes Floyd Warshall an ideal candidate for multicore execution.

3.4 Summary of the Approach

The three algorithms were selected because they allow us to study fundamentally different patterns of computation:

- Dijkstra demonstrates how partial parallelization focused on the dominant cost can yield substantial performance gains.
- Bellman Ford tests parallelization in the presence of frequent write conflicts and fine-grained dependency chains.
- Floyd Warshall represents dense computation with regular memory access and coarse-grained independence, which typically leads to strong speedup.

By designing parallel versions that stay faithful to the structure of each algorithm, this provides insight into how algorithmic behavior, memory locality, and synchronization requirements influence scalability on multicore systems.

4. Experimental Setup

This section describes the hardware environment, compilation settings, graph generation methodology, and evaluation procedure used to measure the performance of the sequential and parallel implementations of the three graph algorithms.

4.1 Platform

All experiments were conducted on the **NYU CIMS Crunchy5** shared-memory server, which provides a multicore environment suitable for OpenMP-based evaluation. The system specifications are:

- **CPU:** AMD Opteron™ 6272
- **Total Cores:** 64 physical cores
  - 4 sockets × 16 cores per socket
  - 1 hardware thread per core (no SMT)
- **Architecture:** x86_64
- **Operating System:** Linux (CIMS Crunchy5 environment)

All programs were compiled directly on Crunchy5 using GCC and G++ with optimization and OpenMP support enabled. Compilation commands were as follows:

```
# Graph generator (C)
gcc -O2 -std=c99 generate_graph.c -o generate_graph

# Algorithms (C++ with OpenMP)
g++ -O2 -std=c++17 -fopenmp dijkstra_seq_parallel.c -o
dijkstra_seq_parallel
g++ -O2 -std=c++17 -fopenmp bellman_ford_seq_parallel.c -o
bellman_ford_seq_parallel
g++ -O2 -std=c++17 -fopenmp floyd_warshall_seq_parallel.c -o
floyd_warshall_seq_parallel
```

The `-O2` optimization level was chosen for stable, predictable compiler behavior, and the default OpenMP runtime environment (GOMP) was used for thread scheduling.

4.2 Graph Inputs

A unified graph generator was implemented to ensure that all algorithms operate on consistent input structures. For a given number of vertices n, the generator produces a **directed graph** with the following properties:

- Vertices labeled from 0to $n-1$
- Approximately $m \approx 8n$edges (average out-degree = 8)
- Integer weights in the range $[1, 10]$, configurable to include negative weights
- No negative cycles (generator enforces DAG structure when negative weights are allowed)

The graph is generated using:

```
./generate_graph <n> 8 10 <negFlag> 45
```

where:

- `avgDeg = 8`: target average out-degree
- `maxW = 10`: maximum absolute edge weight
- `negFlag = 0`: all positive weights (used for BFS and Dijkstra)
- `negFlag = 1`: mixed positive/negative weights (used for Bellman–Ford and Floyd–Warshall)

All algorithms read from the same standardized text-based graph format:

```
n m
u v w
u v w
...
```

Graph Sizes Used

We varied the graph size $n$depending on the algorithm's theoretical complexity:

- **Dijkstra, Bellman-Ford:** n∈ {100, 500, 1000, 5000, 10000, 50000, 100000}
- **Floyd-Warshall ($O(n^3)$)** was restricted to smaller sizes: n∈ {100, 500, 1000, 2000, 5000, 10000}

For each algorithm, a fresh graph instance was generated with the appropriate negFlag to match algorithmic requirements.

4.3 Thread Counts

To evaluate the scalability of each parallel implementation, we executed every algorithm with the following OpenMP thread counts:

- **0 threads**: Sequential baseline
- **1 thread**: Parallel runtime with no concurrency (OpenMP overhead only)

- **2 threads**
- **4 threads**
- **8 threads**

Although Crunchy5 provides 64 cores, limiting experiments to **8 threads** ensures all threads remain within a single NUMA domain. This avoids cross-socket communication overhead and isolates algorithmic effects from hardware NUMA effects.

Each configuration (algorithm × graph size × thread count) was executed once, and runtime data was logged for analysis using `omp_get_wtime()` inside the core computation loop.

5. Experiments and Analysis

This section presents a detailed evaluation of the sequential and parallel implementations of Bellman Ford, Dijkstra, and Floyd Warshall on the Crunchy5 multicore system. The goal is to understand how algorithmic structure, synchronization, and memory behavior affect scalability on shared memory processors.

Each algorithm was evaluated on graphs of increasing size and executed with thread counts equal to 0, 1, 2, 4, and 8. Speedup is defined as:

$$\text{Speedup} = \frac{T_{\text{seq}}}{T_p}$$

The analysis is broken down by algorithm, followed by a cross-algorithm comparison.

5.1 Bellman Ford

Bellman Ford repeatedly relaxes all edges. The parallel implementation uses per vertex locking. This avoids global locking but still performs a large number of fine-grained synchronized writes. Because relaxations target shared distances, contention increases with the number of threads. This makes Bellman Ford particularly challenging to scale on shared memory hardware.

5.1.1 Timing Behavior

Input\Threads	Sequential	1	2	4	8
100	2.47341e-05	9.8289e-05	0.000741432	0.000999405	0.00135286
500	0.000236091	0.00052343	0.00142493	0.00157425	0.00177784
1000	0.00036516	0.000958336	0.00189423	0.00179257	0.00197767
5000	0.00359461	0.00706496	0.00927529	0.0059339	0.00479483
10000	0.00915363	0.0168808	0.0196089	0.0116515	0.00803728
50k	0.0872334	0.15421	0.12537	0.0750591	0.0469545
100k	0.194329	0.340741	0.280827	0.155172	0.100587
500K	1.52748	2.47611	1.8397	1.13097	0.611854
1M	4.10676	6.43962	4.28206	2.75311	1.50976

Table 1: Bellman Ford’s timing table

From the Bellman Ford timing table:

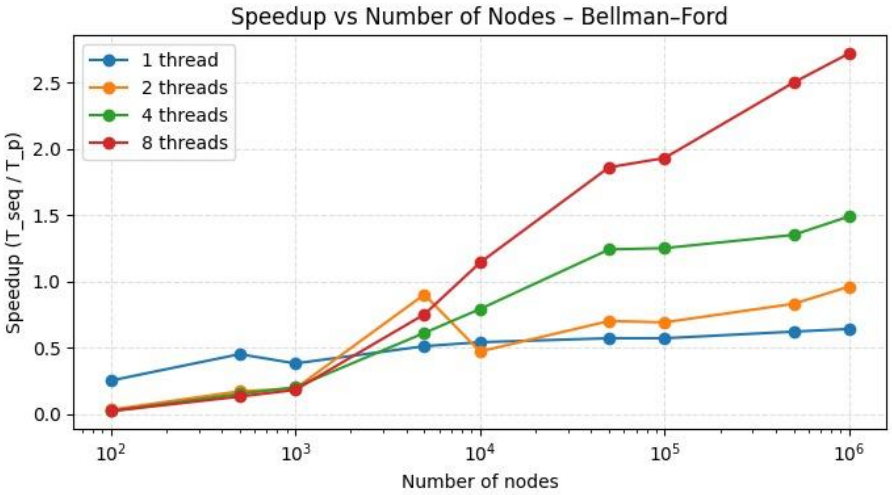
- At **n = 100**, sequential time is 2.47×10^{-5}seconds, while the 8 thread version takes 1.35×10^{-3}seconds.
  Speedup is only **0.02**.
- At **n = 5,000**, sequential time is 0.0036seconds, while the 8 thread version reaches 0.0048seconds.
  Slightly better, but still less than ideal.
- At **n = 50,000**, the 8-thread version achieves genuine parallel benefit:
  - Sequential: 0.0872 seconds
  - 8 threads: 0.0469 seconds
  - Speedup: **1.86**
- At **n = 1,000,000**, parallelization becomes clearly beneficial.
  - Sequential: 4.106 seconds
  - 8 threads: 1.509 seconds
  - Speedup: **2.72**

This trend matches the intuition that Bellman Ford requires very large graphs to amortize the overhead of lock contention.

5.1.2 Speedup Analysis

Input\Threads	1	2	4	8
100	0.25	0.03	0.02	0.02
500	0.45	0.17	0.15	0.13
1000	0.38	0.19	0.20	0.18
5000	0.51	0.9	0.61	0.75
10000	0.54	0.47	0.79	1.14
50k	0.57	0.70	1.24	1.86
100k	0.57	0.69	1.25	1.93
500K	0.62	0.83	1.35	2.50
1M	0.64	0.96	1.49	2.72

Table 2: Bellman Ford's speedup table



Plot 1: Bellman Ford's speedup plot

The performance profile can be explained by the following observations:

1. **Fine grained updates limit parallelism**

   Each edge relaxation may update a shared vertex. Although locks are per vertex, the probability of multiple updates to the same vertex increases with thread count, creating contention.

2. **Memory bound behavior**

   The algorithm streams through the entire edge list in every iteration. For small n, memory latency dominates, and thread overhead adds additional cost.

3. **Speedup only emerges at very large input sizes**

   Once the number of edges reaches millions, the amount of independent work outweighs overhead and lock contention. This produces speedups near three on 8 threads.

4. **Parallel Bellman Ford is highly input sensitive**

   It performs poorly on small graphs but improves significantly for large sparse DAGs.

Overall, Bellman Ford demonstrates the difficulty of parallelizing irregular, relaxation-based algorithms on shared memory systems.

5.2 Dijkstra

Dijkstra is implemented using the classical array-based version. The only parallelized region is the minimum selection phase, which dominates runtime for large graphs. Relaxation remains sequential. This makes Dijkstra a hybrid algorithm with a clear separation of parallel and serial phases.

5.2.1Timing Behavior

Input\Threads	Sequential	1	2	4	8
100	6.02268e-05	0.000313329	0.00120937	0.00142241	0.00233773
500	0.00136605	0.00244664	0.00369218	0.0039845	0.00714954
1000	0.00536114	0.00741885	0.00780177	0.00789014	0.0138878
5000	0.130614	0.130548	0.0832864	0.0605732	0.0716933
10000	0.522407	0.493022	0.288194	0.188075	0.171681
50k	13.0648	11.8186	6.28447	3.37074	2.03416
100k	52.1636	46.8259	24.5213	12.8591	7.24865

Table 3: Dijkstra's timing table

From the Dijkstra's timing table:

- For **n = 100**, parallel runs are slower than sequential (speedup between 0.03 and 0.19). Overhead dominates.
- At **n = 5,000**, performance starts improving:
    - Sequential: 0.130 seconds
    - 4 threads: 0.060 seconds
    - Speedup: **2.16**
- At **n = 50,000**, clear scaling appears:
    - Sequential: 13.06 seconds
    - 4 threads: 3.37 seconds
    - Speedup: **3.87**
    - 8 threads: 2.03 seconds
    - Speedup: **6.42**
- At **n = 100,000**, the highest speedup is recorded:
    - Sequential: 52.16 seconds

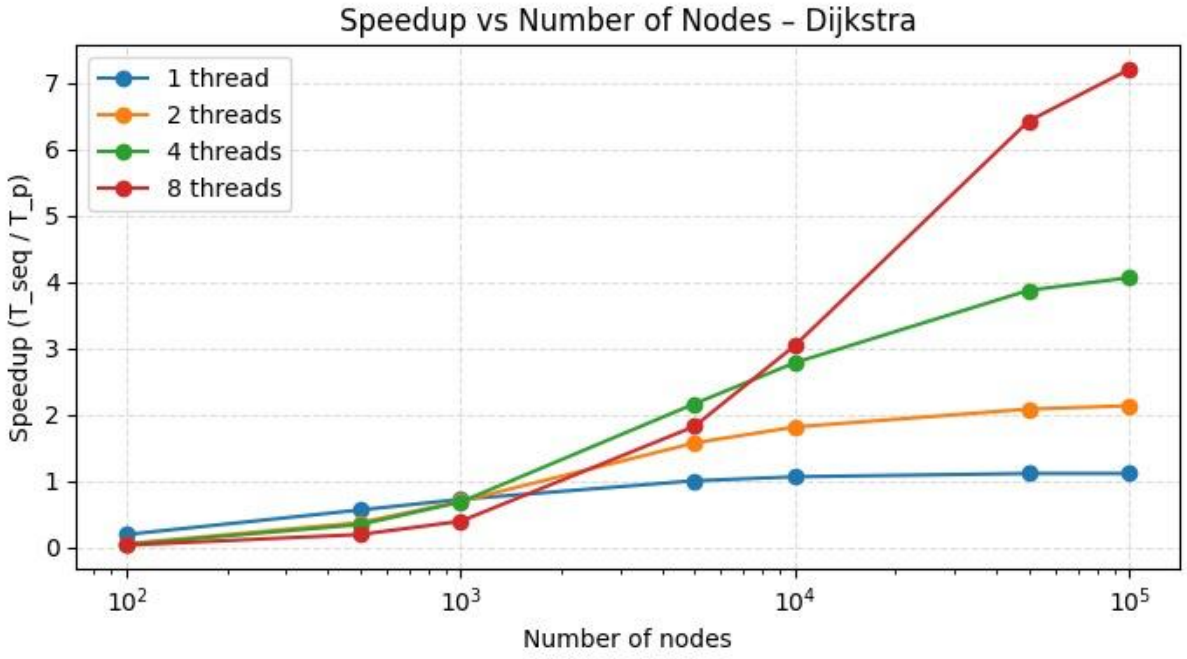
- 8 threads: 7.24 seconds
- Speedup: **7.20**

These gains come entirely from parallelizing the O(n) global minimum search.

5.2.2 Speedup Analysis

Input\Threads	1	2	4	8
100	0.19	0.05	0.04	0.03
500	0.56	0.37	0.34	0.19
1000	0.72	0.69	0.68	0.39
5000	1	1.57	2.16	1.82
10000	1.06	1.81	2.78	3.04
50k	1.11	2.08	3.87	6.42
100k	1.11	2.13	4.06	7.2

Table 4: Dijkstra's speedup table



Plot 2: Dijkstra's speedup plot

Key observations:

1. **Minimum selection dominates for large n**

The cost of scanning all n vertices per iteration becomes significant. Parallelizing this step provides substantial benefit.

2. **Relaxation remains sequential**

   Since the outgoing edge relaxations are not parallelized, Dijkstra cannot reach linear speedup. However, partial parallelization still yields up to 7x improvement.

3. **Performance stabilizes at high thread counts for small graphs**

   Overhead overshadows benefits when $n < 2000$.

4. **Excellent scaling for $n \geq 50{,}000$**

   The parallel minimum selection reaches steady and predictable speedup.

Dijkstra shows that even partial parallelization can significantly improve performance when the parallelized region aligns with the algorithm's dominant computational cost.

5.3 Floyd Warshall

Floyd Warshall is a dense $O(n^3)$ dynamic programming algorithm. The parallel version splits the middle loop across threads so that each thread updates entire rows of the distance matrix. This eliminates synchronization and exploits coarse grained parallelism

5.3.1Timing Behavior

Input\Threads	Sequential	1	2	4	8
100	0.00130524	0.00146158	0.00196311	0.00200576	0.00218008
500	0.101161	0.0877734	0.06857	0.0453313	0.0299721
1000	0.675853	0.561159	0.425847	0.271465	0.166965
2000	4.87079	3.9407	2.97431	1.90462	1.1197
5000	66.5604	54.2987	41.1021	25.3827	15.0892
10000	476.426	385.291	296.382	186.582	110.063

Table 5: Floyd Warshall's timing table

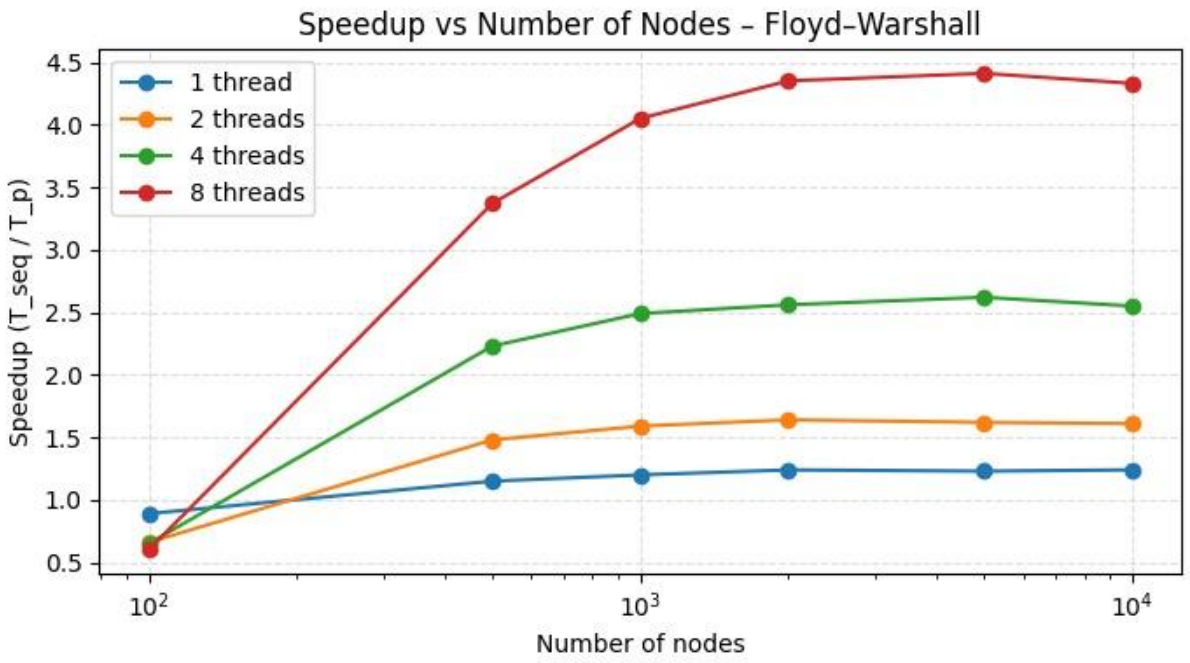
From the Floyd Warshall table:

- At **n = 100**, sequential time is 0.0013 seconds, and the 8 thread version runs in 0.0021 seconds, yielding low speedup due to tiny matrix size.
- At **n = 500**, speedups improve:
  - Sequential: 0.101 seconds
  - 8 threads: 0.0299 seconds
  - Speedup: **3.37**
- At **n = 1000**, strong scaling appears:
  - Sequential: 0.676 seconds
  - 8 threads: 0.167 seconds
  - Speedup: **4.05**
- At **n = 5000**, Floyd Warshall solves a 5000x5000 matrix in 15.08 seconds on 8 threads, compared to 66.56 seconds sequential:
  - Speedup: **4.41**
- At **n = 10000**, the speedup slightly dips due to memory bandwidth and cache limits:
  - Sequential: 476 seconds
  - 8 threads: 110 seconds
  - Speedup: **4.33**

5.3.2 Speedup Analysis

Input\Threads	1	2	4	8
100	0.89	0.66	0.65	0.60
500	1.15	1.48	2.23	3.37
1000	1.20	1.59	2.49	4.05
2000	1.24	1.64	2.56	4.35
5000	1.23	1.62	2.62	4.41
10000	1.24	1.61	2.55	4.33

Table 6: Floyd Warshall’s speedup table



Plot 3: Floyd Warshall's speedup plot

Floyd Warshall shows the most consistent and reliable parallel scaling among the three algorithms. This is mainly because of how naturally its structure maps to OpenMP.

1. **Independent row updates**

   For a fixed value of k, each row of the distance matrix can be updated independently. Threads operate on separate rows, so there are no shared writes or lock contention. This makes the parallel region almost entirely free of synchronization overhead.

2. **Stable speedup for moderate and large n**

   Once n reaches 500 or higher, the speedup curves become smooth and predictable. On 8 threads, speedup increases quickly and remains between 4.0 and 4.4 for matrix sizes up to 5000. This reflects the regular, well balanced nature of the work across threads.

3. **Memory system limits performance at the largest sizes**

   At n = 10000, the matrix contains one hundred million entries. Accessing and updating such a large structure puts heavy pressure on cache and memory bandwidth.

This causes a slight flattening of the speedup curve, although performance still remains strong.

4. **Good scaling due to high computation per memory access**

   Floyd Warshall performs a large number of arithmetic operations for each memory load, which helps hide memory latency. As a result, it maintains good parallel efficiency even when memory becomes a bottleneck.

Overall, Floyd Warshall is the best performing parallel algorithm in the study because it combines coarse grained independence, regular memory access, and high arithmetic intensity.

6. Conclusions

The structure of each algorithm is the main factor that determines whether it scales on a multicore system: Algorithms with regular and predictable work, such as Floyd Warshall, parallelize very effectively because threads can operate independently on large chunks of data. In contrast, algorithms like Bellman Ford struggle because their work involves many small updates that frequently conflict, which limits how much parallel speedup is realistically achievable.

Focusing on the dominant source of cost can produce strong speedup even when only part of the algorithm is parallelized: In Dijkstra, parallelizing just the minimum selection step leads to significant gains for large graphs, reaching more than seven times faster on eight threads. This shows that it is not always necessary to parallelize every line of an algorithm. Instead, identifying and accelerating its most expensive part can deliver most of the benefit.

Synchronization overhead can outweigh the benefits of threading in algorithms with irregular memory access: Bellman Ford, even with per vertex locking, spends a substantial amount of time waiting on locks when graphs are small or medium sized. Only at very large problem sizes does the amount of available work overcome these costs. This highlights the importance of choosing parallel strategies that minimize shared state and reduce the need for constant coordination across threads.

7. References

[1] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun, “Accelerating CUDA Graph Algorithms at Maximum Warp,” PPoPP, 2011.

[2] J. Lee and H. Kim, “TAP: A TLP Aware Cache Management Policy for CPU GPU Heterogeneous Architectures,” HPCA, 2012.

[3] C. E. Leiserson and T. Schardl, “A Work Efficient Parallel Breadth First Search Algorithm,” SPAA, 2010.

[4] D. A. Bader and K. Madduri, “Designing Multithreaded Algorithms for BFS and ST Connectivity on Shared Memory Systems,” ICPP, 2006.

[5] U. Meyer and P. Sanders, “Delta Stepping: A Parallel Single Source Shortest Path Algorithm,” Journal of Algorithms, 2003.

[6] P. Harish and P. J. Narayanan, “Accelerating Large Graph Algorithms on the GPU Using CUDA,” HiPC, 2007.

[7] G. Malewicz et al., “Pregel: A System for Large Scale Graph Processing,” SIGMOD, 2010.

[9] Y. Low et al., “GraphLab: A New Framework for Parallel Machine Learning,” arXiv:1006.4990, 2010.