Parallel and Distributed Computing

CSE 467 - Project Milestone 3

Group Members:
Abdullah Iqbal (26904)
Haseeb Ahmed (26077)
Azizullah Khan (24931)

Institute of Business Administration
April 2025

Contents

1 Methodology Plan

1.1 A* algorithm

1.1.1 Steps Followed

Step 1: Problem Setup

- Define a grid-based or graph-based environment where shortest pathfinding is required
- Each node contains position coordinates and a cost estimate (heuristic).

Step 2: Serial Implementation

- Implement standard A* algorithm in Python using a priority queue (heapq) for the open list.
- Maintain a closed list for already explored nodes.

Step 3: Parallel Implementation

i. Identify parallelizable parts of the A* algorithm:

- Neighbor generation and heuristic computation.
- Path cost calculations for multiple neighboring nodes.

ii. Apply multithreading or multiprocessing:

- Multiprocessing will be preferred (due to Python GIL limitations with threading).
- Use multiprocessing.Pool to evaluate multiple neighboring nodes in parallel.
- Optimize shared data access using Manager objects or queues

Step 4: Performance Measurement

- Measure execution time, memory usage, and node expansion count for both implementations.
- Run experiments on different grid sizes and obstacle densities

Step 5: Result Analysis

- Compare serial and parallel versions based on recorded metrics.
- Analyze the scalability of the parallel version with the number of processes.

1.1.2 Technique used

Multithreading / Multiprocessing in Python:

- Specifically, they used shared-memory multithreading on multi-core CPUs (NOT multiprocessing or distributed MPI).
- Threads share access to the open list (priority queue) and closed list.
- Locking mechanisms (mutex locks) are carefully used to avoid race conditions when multiple threads access shared structures.

1.1.3 The Why

- The A* algorithm has parts that can be parallelized naturally (neighbor evaluation, heuristic calculations).
- Python's multiprocessing allows real parallelism by spawning independent processes (bypassing GIL).
- It is simple enough to set up without needing distributed systems (e.g., MPI clusters) yet powerful enough to show significant performance differences.

1.2 Dijkstra's algorithm

1.2.1 Steps Followed

Step 1: Problem Setup

- Define a graph-based structure with weighted edges (non-negative).
- Each node keeps track of the current known shortest distance from the source node.

Step 2: Serial Implementation

- Use a priority queue (min-heap) to always process the node with the smallest distance.
- For each node, update the distances to its neighbors if a shorter path is found.

Step 3: Parallel Implementation

i. Identify parallelizable parts of the algorithm:

- Updating distances to neighbors can happen in parallel since each neighbor's computation is independent

ii. Implement using Python's multiprocessing.Pool

- When a node is extracted from the queue, spawn parallel processes to relax its neighbors.

iii. Carefully manage shared state (distance dictionary) using multiprocessing-safe structures (like Manager dict).

Step 4: Performance Measurement

- Measure execution time and memory usage.
- Count the number of node relaxations.
- Run experiments across graphs of different sizes and densities.

Step 5: Result Analysis

- Compare the execution time and node processing count between serial and parallel versions.
- Analyze efficiency gains, overheads, and scalability.

1.2.2 Technique used

Multithreading / Multiprocessing in Python: For the parallel implementation of Dijkstra's algorithm, we will use Python's multiprocessing module to achieve parallelism across multiple CPU cores. First, we will implement the serial version of Dijkstra's algorithm in Python to serve as the baseline for comparison. In the parallel version, after selecting the current minimum-distance node, we will use multiple processes to simultaneously update the tentative distances of its neighboring nodes. To manage shared data structures such as the distance array and the visited set safely, we will apply synchronization techniques like multiprocessing Manager lists, Locks, or Queues to avoid race conditions. Execution time for both the serial and parallel versions will be measured across graphs of varying sizes and densities. We will then calculate the speedup achieved through parallelization and plot execution time versus the number of nodes to visually demonstrate the performance improvements.

1.2.3 The Why

- The neighbor relaxation phase is independent for each adjacent node, making it naturally parallelizable.
- Python's multiprocessing allows utilization of multiple CPU cores, bypassing the GIL.
- Full distributed computing (like MPI) is not required for this project's scale.

1.3 Bellman-Ford

1.3.1 Steps Followed

Step 1: Implement the serial version of the Bellman-Ford algorithm in Python.
Step 2: Implement the parallel version using Python's multiprocessing module.
Step 3: Design both implementations to handle the same graph input structure (e.g., adjacency list or edge list) for fair comparison.

Step 4: Measure and record execution times for both implementations across multiple graph sizes.
Step 5: Analyze the speedup, efficiency, and overheads observed.
Step 6: Compare experimental results with selected research paper benchmarks.

1.3.2 Technique used

Multithreading / Multiprocessing in Python:
Specifically, we will use multiprocessing pool to parallelize the relaxation of edges across multiple processes.

Each iteration of Bellman-Ford relaxes all edges. In parallel implementation, edges will be distributed among processes for simultaneous relaxation.

1.3.3 The Why

Bellman-Ford inherently relaxes all edges independently during each iteration (except for global distance updates), which fits the data-parallel model perfectly.

Python's multiprocessing bypasses the Global Interpreter Lock (GIL) limitation (unlike multithreading) and fully utilizes multiple cores for CPU-bound tasks like Bellman-Ford.

This approach is simple, portable (runs without needing GPUs), and aligns with our course objective of demonstrating parallelism practically.

2 Literature Comparison

2.1 A* Algorithm

The paper chosen as the base paper for implementing A* in parallel is ***Parallel A* Search on a Shared-Memory Multi-Core System", by Robert Holte et al., IJCAI 2015.*** and can be accessed at Link to the paper
The paper describes how to run A* search with multiple threads exploring nodes in parallel.

Each thread repeatedly picks a node, expands its neighbors, and updates the open list.

They carefully manage access to shared structures to avoid deadlocks and minimize lock contention.

It was shown that speedup is achievable on multi-core systems, especially for large search spaces.

2.1.1 Similarity/Difference

Similarities:
1. Both the paper and our project aim to parallelize the node expansion and neighbor evaluation steps of A*.
2. Both use a shared structure (open list and closed list).

Differences:
1. The paper uses multithreading with careful locking on shared-memory multi-core systems.
2. Our project uses multiprocessing in Python because of Python's Global Interpreter Lock (GIL) multiprocessing achieves true parallelism across cores in Python, while threading is limited.
3. The paper focuses on minimizing lock contention in a highly optimized C++ environment, whereas we focus on simple multiprocessing for educational purposes.

2.1.2 Performance Metrics

1. Execution time (speedup compared to serial A*).
2. Number of nodes expanded.
3. Efficiency of lock management (lock contention time).

2.1.3 Approach Adopted

We will compare Execution time trends for increasingly large graphs and the Speedup achieved by parallel vs serial implementations between the Base Paper and our implementation.

2.2 Dijkstra's Algorithm

The paper chosen as the base paper for implementing A* in parallel is ***"Parallel Dijkstra's Algorithm on a Graph Using Multiple Cores and CUDA", by Tirtha S. et al 2017***

- This paper explores how Dijkstra's algorithm can be accelerated using both CPU
- multicore parallelism and GPU parallelism (CUDA).
- The authors parallelized the relaxation of neighboring nodes after selecting the minimumdistance node.

2.2.1 Similarity/Difference

Similarities:
1. Both the paper and our project parallelize the relaxation step (neighbor updates).
2. Both use a priority-based structure for selecting the current node.

Differences:
1. The paper uses GPU-based CUDA programming for heavy parallelism, while our project uses CPU multiprocessing.
2. The paper handles much larger graph sizes (millions of nodes), while our experiments focus on medium-scale graphs suitable for CPU cores

2.2.2 Performance Metrics

1. Execution time.
2. Speedup compared to the serial version.
3. GPU resource utilization (only in paper, not applicable here).

2.2.3 Approach Adopted

We will compare Execution time trends for increasing graphs sizes and the Speedup achieved over the serial version between the Base Paper and our implementation.

2.3 Bellman-Ford

The paper chosen as the base paper for implementiing Bell-man Ford in parallel is ***G. Hajela and M. Pandey, "Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford," International Journal of Computer Applications, 2014.*** and can be accessed at Link to the paper

2.3.1 Similarity/Difference

Similarities:
1. Both projects aim to parallelize the edge relaxation step of Bellman-Ford.
2. Both compare performance between serial and parallel executions.

Differences:
1. Hajela and Pandey use OpenCL to implement Bellman-Ford on GPUs.
2. Our project uses Python multiprocessing for CPU-based parallelization.
3. Their approach leverages GPU memory and cores massively, while ours distributes work across CPU cores.

2.3.2 Performance Metrics

1. Execution time for solving shortest path problems on graphs of increasing sizes.
2. Speedup achieved compared to serial version.
3. Impact of synchronization overhead on performance.

2.3.3 Approach Adopted

We will use their execution time measurements for different graph sizes and speedup analysis as benchmarks to compare how much performance gain we can achieve using Python multiprocessing vs their OpenCL-based GPU implementation.

3 Practical Implementation

3.1 A* Algorithm

3.1.1 Tools and Platform used

Language: Python 3.11

Libraries:
1. multiprocessing (for parallel version)
2. time (for timing executions)
3. heapq
4. psutil (for memory usage, if needed)
5. matplotlib (for visualization of paths if needed)

3.1.2 Testing and Evaluation

Implement test cases with varying:

1. Grid sizes: (e.g., 10x10, 50x50, 100x100, 500x500)
2. Obstacle densities: (e.g., 10

Each case will be run multiple times (e.g., 10 times) to account for variability and averaged.

3.1.3 Dataset usage

1. Synthetic grids generated randomly:

- Start and goal positions randomly selected.
- Random obstacles added based on specified densities

2. Optionally use known graph datasets:

- Maze datasets (simple and complex)
- Road network graphs (small samples)

3.2 Dijkstra's Algorithm

3.2.1 Tools and Platform used

Language: Python 3.11

Libraries:
1. multiprocessing (for parallel version)
2. time (for timing executions)

3. heapq
4. psutil (for memory usage, if needed)
5. networkx (for graph generation and visualization)

3.2.2 Testing and Evaluation

Generate synthetic graphs with varying:

1. Number of nodes: (e.g., 100, 500, 1000, 5000)
2. Edge density: Sparse vs Dense graphs.

Each case will be run both on serial and parallel versions 10 times and the results will be averaged.

3.2.3 Dataset usage

1. Random Graphs:

- Using NetworkX's $\text{gnp}_{r}andom_{g}raph()methodtogenerategraphs.$

2. Optional Real Graphs:

  - Small road networks (if needed for additional experiments).

3.3 Bellman-Ford

3.3.1 Tools and Platform used

Language: Python 3.11

Libraries:
1. multiprocessing (for parallel version)
2. time (for timing executions)
3. networkx (optional, for graph generation and handling)

Platform: Windows 10 / Ubuntu (depending on availability)
Hardware: Multi-core CPU machine (e.g., Intel i5 or i7)

3.3.2 Testing and Evaluation

Test both serial and parallel Bellman-Ford implementations on:

1. Small graphs (e.g., 100–500 vertices)
2. Medium graphs (e.g., 1000–3000 vertices)
3. Large graphs (e.g., 5000–10000 vertices)

For each test:

1. Generate random directed graphs with edge weights between -10 and 10.
2. Record execution time for serial and parallel versions.
3. Calculate speedup = (Serial Time) / (Parallel Time).
4. Note memory usage and CPU utilization if possible

3.3.3 Dataset usage

1. Randomly generated graphs using networkx gnmrandomgraph().
2. Edge weights assigned randomly between -10 and 10.
3. Ensure the graphs are sparse to moderate in density to match practical scenarios.

4 Expected Output and Comparison Strategy

4.1 A* Algorithm

4.1.1 Expected Results

1. Execution times for serial and parallel versions
2. Memory usage data (optional).
3. Number of nodes expanded.
4. Path length and correctness verification (ensure the parallel version still produces valid shortest paths).

4.1.2 Comparison with Adopted Paper

1. Calculate Speedup: Speedup=TserialTparallel
2. Compare node expansion rates
3. Graph the results (Execution time vs Grid size).
4. Compare findings with the selected paper's results, focusing on:

- Magnitude of speedup
- Scalability trends
- Efficiency improvements

4.1.3 Demonstration of Efficiency or Performance Gains

1. Show clear graphs and tables comparing times and nodes expanded
2. Discuss scenarios where parallel A* performs significantly better (large grids, dense obstacles).
3. Acknowledge possible overheads for small-sized problems (due to multiprocessing overhead).

4.2 Dijkstra's Algorithm

4.2.1 Expected Results

1. Execution time for serial and parallel Dijkstra's.
2. Number of nodes relaxed.
3. Speedup ratios.
4. Memory usage statistics (optional).

4.2.2 Comparison with Adopted Paper

1. Calculate Speedup: Speedup=TserialTparallel
2. Compare execution times graphically (graph size vs time).
3. Compare findings with the paper:

- Focus on speedup trends.
- Discuss differences due to CPU vs GPU parallelism.

4.2.3 Demonstration of Efficiency or Performance Gains

1. Present tables and plots comparing execution times.
2. Highlight scenarios where parallel Dijkstra's shows major improvements (dense graphs, larger graphs).
3. Also note cases where parallel overhead cancels out gains (small graphs).

4.3 Bellman-Ford

4.3.1 Expected Results

1. The parallel version should show noticeable speedup over the serial version, especially as graph size increases.
2. However, due to Python multiprocessing overheads (process spawning, inter-process communication), the speedup may not be as high as GPU-based approaches (e.g., OpenCL).
3. Small graphs may not benefit much; larger graphs will better showcase performance gains.

4.3.2 Comparison with Adopted Paper

1. The parallel version should show noticeable speedup over the serial version, especially as graph size increases.
2. However, due to Python multiprocessing overheads (process spawning, inter-process communication), the speedup may not be as high as GPU-based approaches (e.g., OpenCL).
3. Small graphs may not benefit much; larger graphs will better showcase performance gains.

4.3.3 Demonstration of Efficiency or Performance Gains

1. Speedup: Ratio of serial to parallel time.
2. Scalability: How performance improves with increasing graph size.
3. CPU Utilization: Observe CPU core usage during parallel runs (optional using system monitors).
4. Efficiency Plot: Improvement = ((Serial Time - Parallel Time) / Serial Time) * 100.

5 References

[1] Tirtha S. et al., Parallel Dijkstra's Algorithm on a Graph Using Multiple Cores and CUDA, 2017.

[2] R. Holte et al., Parallel A* Search on a Shared-Memory Multi-Core System, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2015.

[3] Parallelizing A* Search Algorithms for Faster Shortest Path Computation.

[4] G. Hajela and M. Pandey, "Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford," International Journal of Computer Applications, 2014.