University of Craiova
Faculty of Automation, Computers and Electronics

Dijkstra

Modeling and Performance Evaluation

Student
Smarandache Alexandru

ISB
Second Year

January 2025

Contents

1 Project statement

This project will analyze the implementation of the Dijkstra search algorithm. A series of implementations will be made: sequential, multithreaded or parallel. Cases of different complexity will be tested and a comprehensive analysis of the results will be performed, thus allowing a comparison both between the implementation methods and between the Java and C++ programming languages.

2 Implementations of Dijkstra's algorithm

This project will analyze different implementations of Dijkstra's algorithm, both for Java and C++. Both sequential algorithms and implementations that target parallelism will be compared. The performance of the algorithms will be given by the speed with which they provide the correct answer for the received data set. The shorter the time, the better the algorithm has performance.

The project aims to evaluate the following implementations:

2.1 C++ Sequential

2.2 Java Sequential(using priority queue)

2.3 C++ STL parallel

2.4 C++ OpenMP

2.5 C++ MPI

2.6 Java Parallel Streams

2.7 Java with Threads

3 Input data generation

To generate input data, a graph is generated based on a given density.
Input format:
noOfNodes sourceNode
c00 c01 ... c0(noOfNodes -1)
c01 c11 ... c1(noOfNodes -1)
...
c(noOfNodes -1)0 c(noOfNodes -1)1 ... c(noOfNodes -1)(noOfNodes -1)

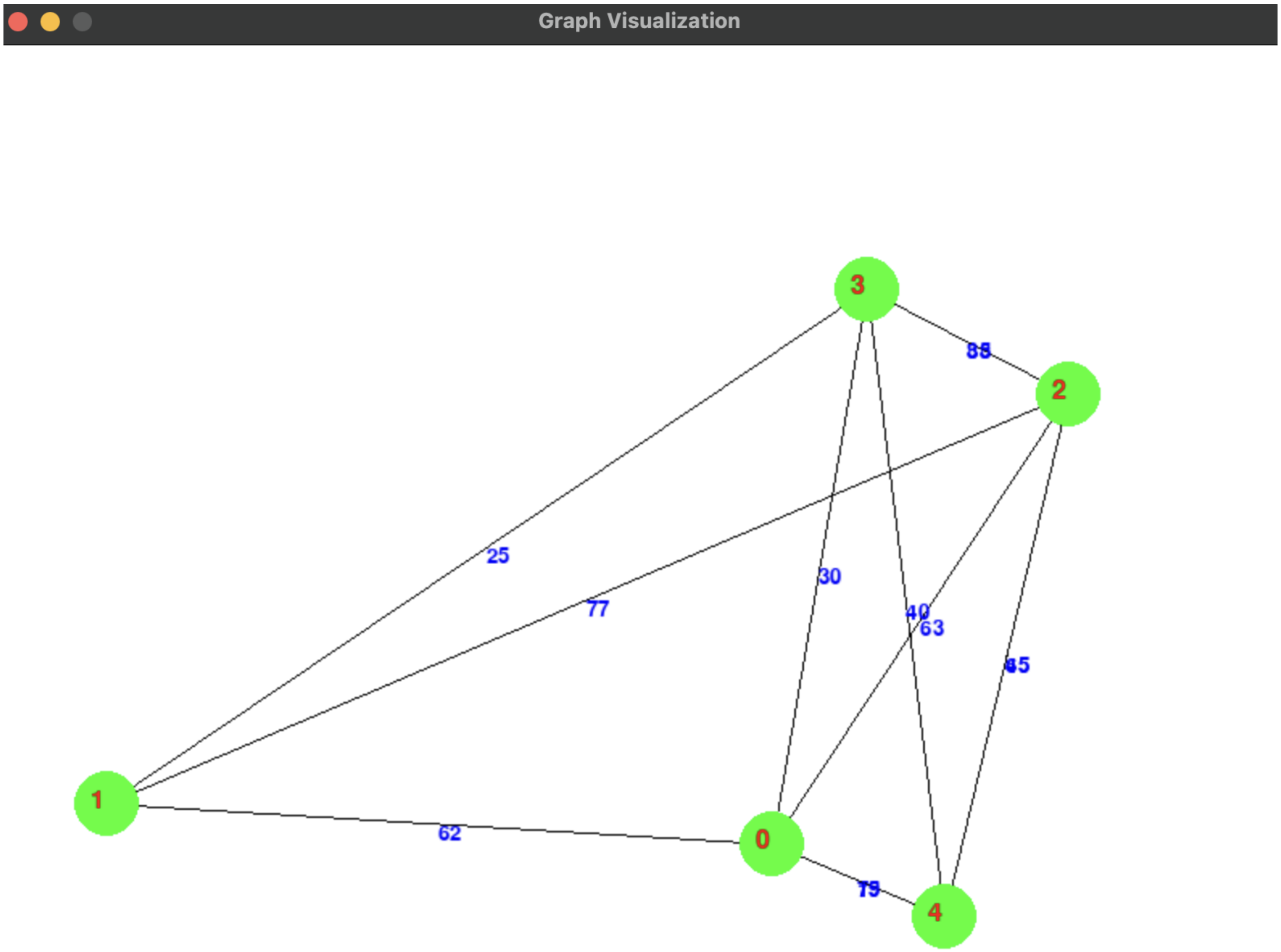
Where:

- **noOfNodes**: the number of nodes
- **sourceNode**: the source node
- **c**: the adjacency matrix

```
/**
 * Generate a random graph.
 *
 * @param noOfNodes The number of nodes.
 * @param density   The density of graph. If density is @Constants.MIN_DENSITY, no edge will be generated.
 * If density is @Constants.MAX_DENSITY, edges between all nodes will be generated.
 *
 * @return The generated graph.
 */
public static Graph generateGraph(final int noOfNodes, final int density) {
  List<List<Integer>> adjacencyMatrix = new ArrayList<>(noOfNodes);
  for (int i = 0; i < noOfNodes; i++) {
      List<Integer> row = new ArrayList<>();
      for (int j = 0; j < noOfNodes; j++) {
          final int grade = RandomUtil.generateNumber(Constants.MIN_DENSITY, Constants.MAX_DENSITY);
          if (i == j) {
              row.add(j,  element: 0);
          } else {
            row.add(j, grade > density ? Constants.INFINITE :
              RandomUtil.generateNumber(Constants.MIN_EDGE_COST, Constants.MAX_EDGE_COST)
            );
          }
      }
      adjacencyMatrix.add(i, row);
  }

  return new Graph(adjacencyMatrix);
}
```

Here are an example of the generated graph:



4 Output data

Each algorithm will have the following output format:
Vertex Distance from source(source node)
0 c0
1 c1
...
noOfNodes -1 c(noOfNodes - 1)
Where:
c[i] is the distance between node i and source node.

5 Tools

5.1 Programming Languages

The following programming languages will be used for the project:

- **Java**: A versatile object-oriented programming language, widely recognized for its portability and robust performance, making it suitable for cross-platform applications.
- **C++**: A high-performance programming language with a focus on system-level development and resource-intensive applications, offering fine-grained control over system resources.

5.2 Parallel Computing Frameworks

To enable efficient parallel computation, the project will utilize the following tools:

- **OpenMP**: A widely used API for shared-memory parallel programming in C++, providing a simple interface for multithreading by using compiler directives, library routines, and environment variables. OpenMP facilitates parallelism in loops and tasks, making it easier to scale applications on multi-core processors.
- **MPI (Message Passing Interface)**: A standard for distributed-memory parallel programming, enabling communication between processes running on different nodes in a cluster. MPI is highly efficient for large-scale parallelism, offering low-latency and high-throughput communication capabilities for resource-intensive applications.

5.3 DevOps Tools

- **Git**: A distributed version control system that tracks code changes, facilitates collaboration among developers, and enables efficient version management.

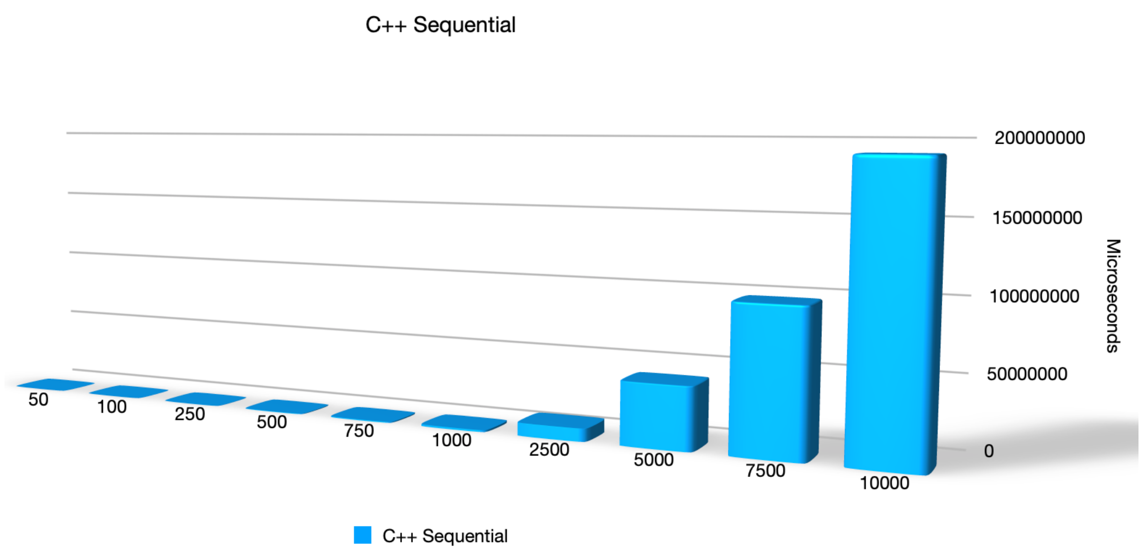
5.4 Development Environment

The project will leverage the following development environments and tools:

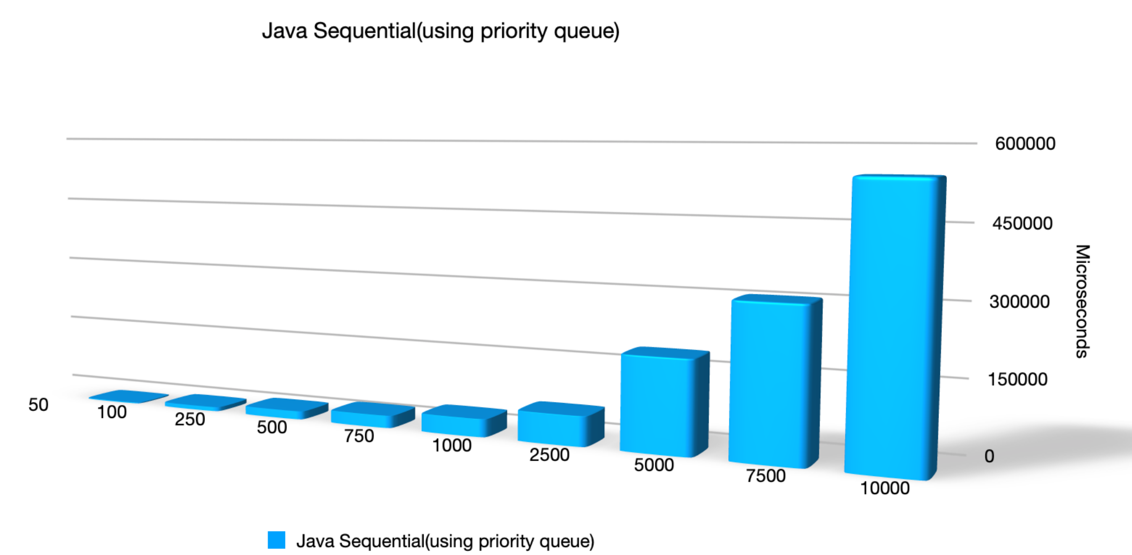
- **IntelliJ IDEA**: A feature-rich integrated development environment (IDE) designed for Java development, offering tools for code editing, debugging, and integration.
- **Visual Studio**: A powerful IDE supporting multiple programming languages, including C++, with advanced debugging and testing capabilities.
- **GitHub**: A cloud-based platform for hosting Git repositories, enabling collaborative development, issue tracking, and code reviews.

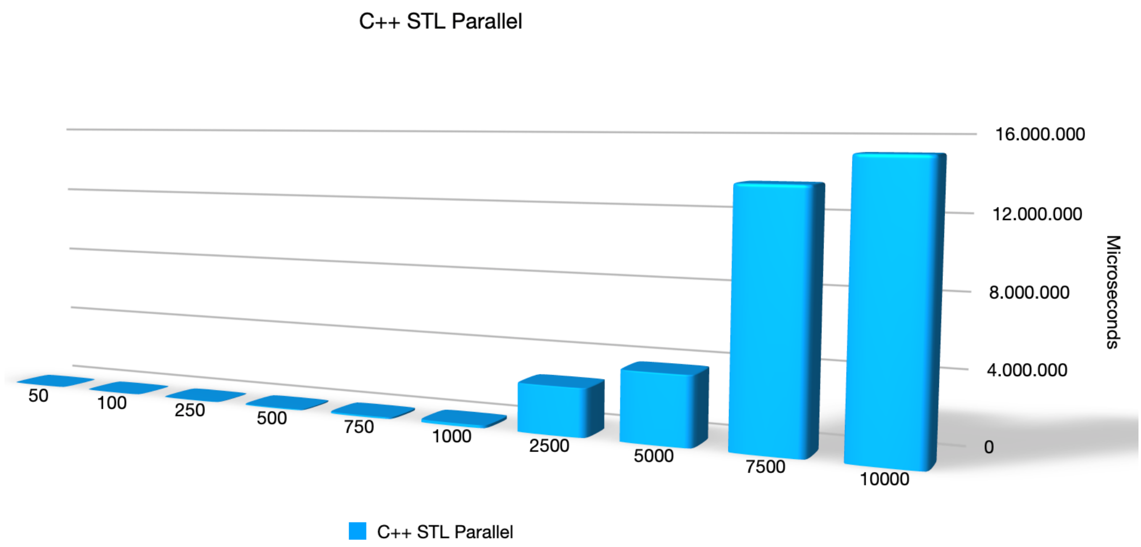
6 Results

6.1 C++ Sequential

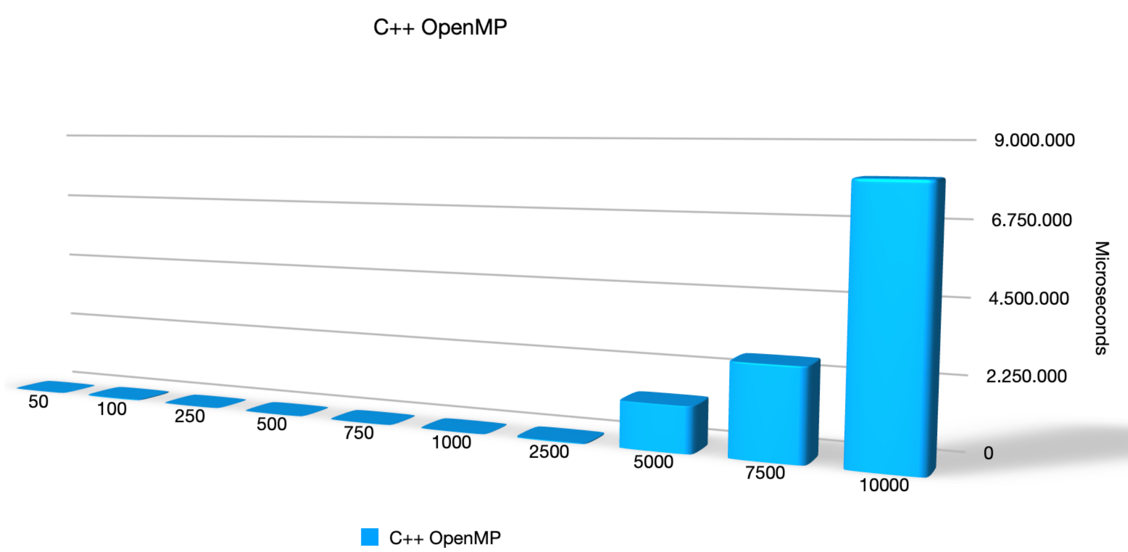


6.2 Java Sequential(using priority queue)

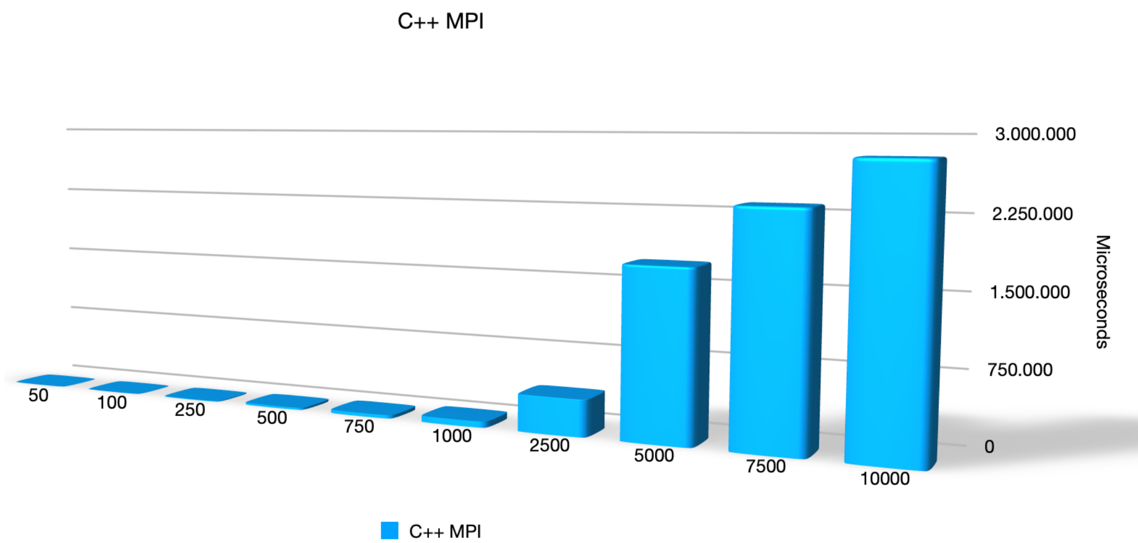



C++ STL Parallel
16.000.000
12.000.000
8.000.000
4.000.000
0
Microseconds
50
100
250
500
750
1000
2500
5000
7500
10000
C++ STL Parallel

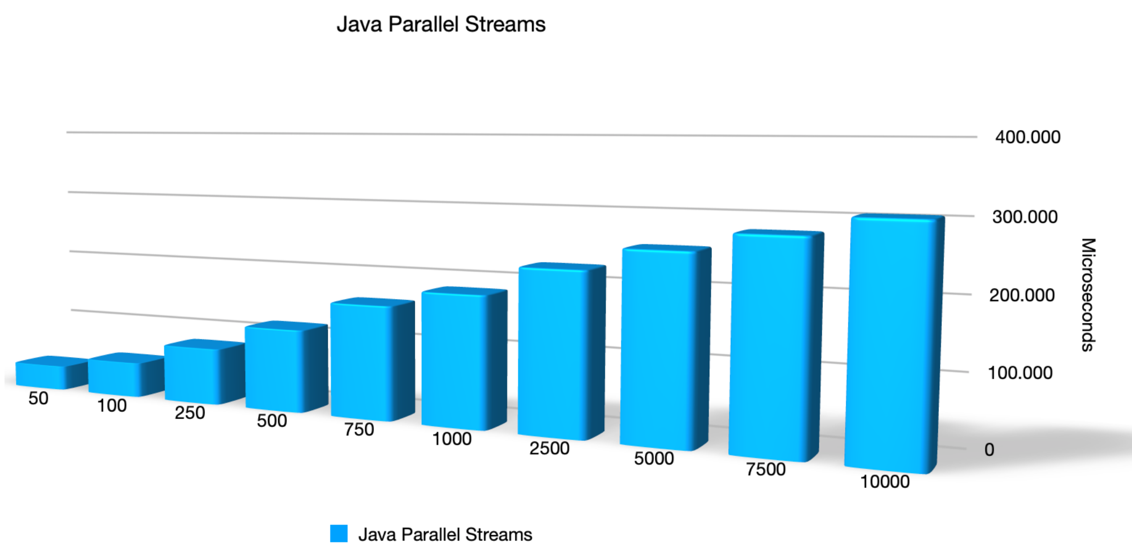
6.3 C++ OpenMP


C++ OpenMP
9.000.000
6.750.000
4.500.000
2.250.000
0
Microseconds
50
100
250
500
750
1000
2500
5000
7500
10000
C++ OpenMP

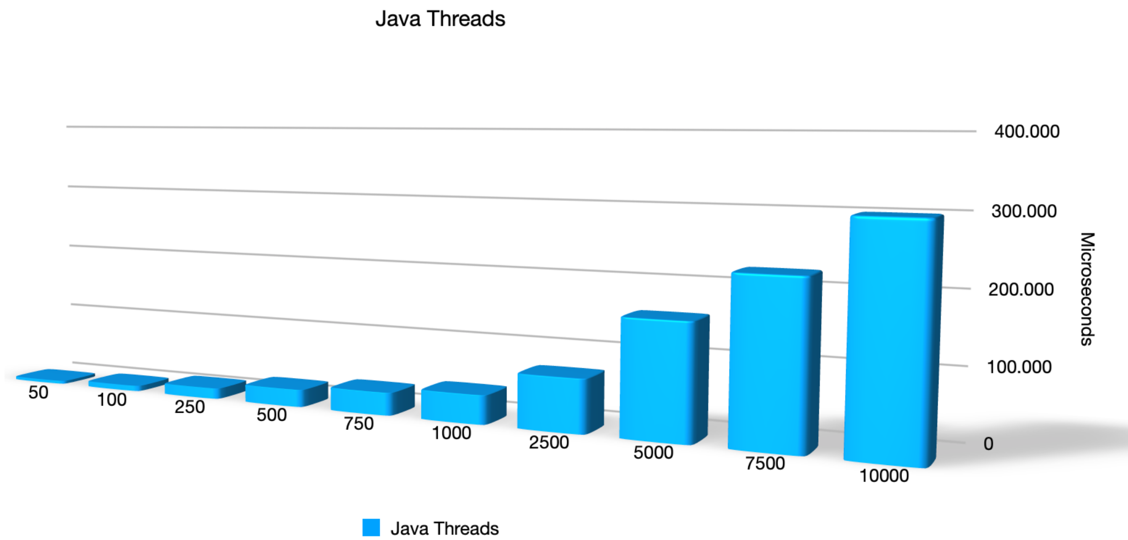
6.4 C++ MPI



6.5 Java Parallel Streams

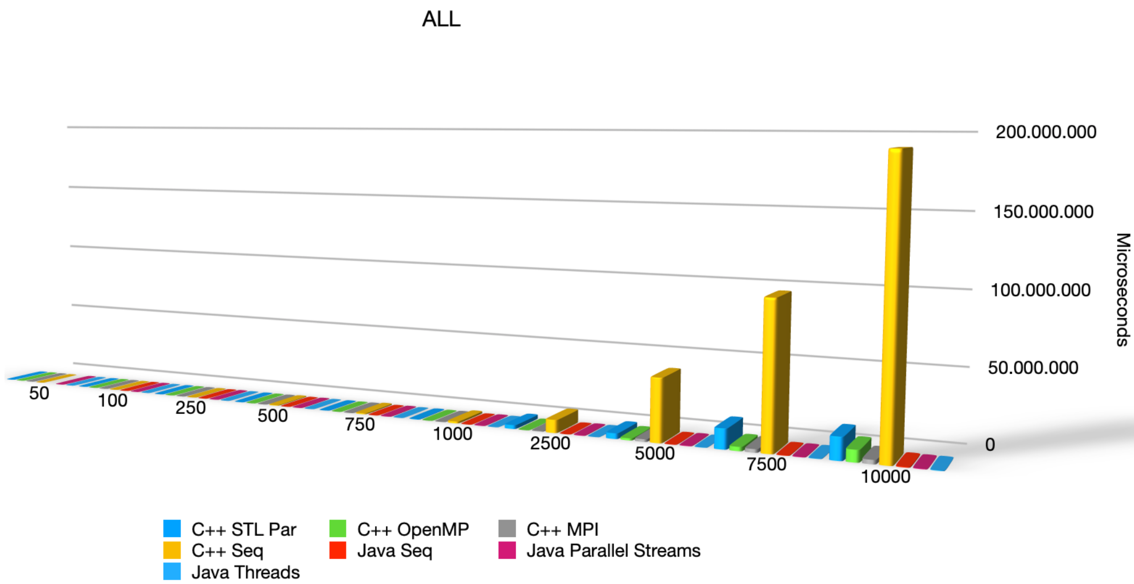


6.6 Java with Threads

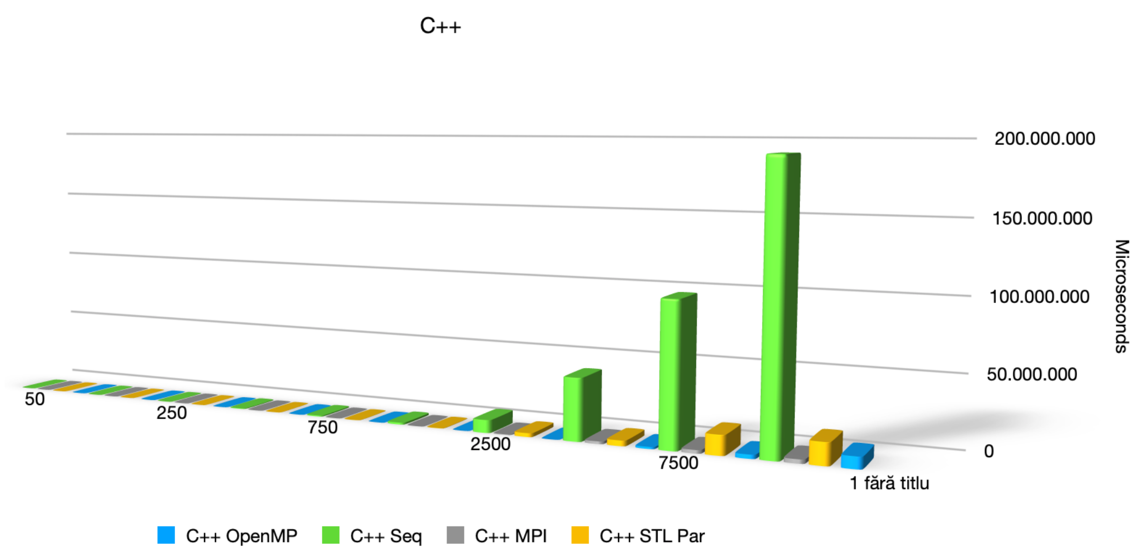


7 Comparasion - Implementation

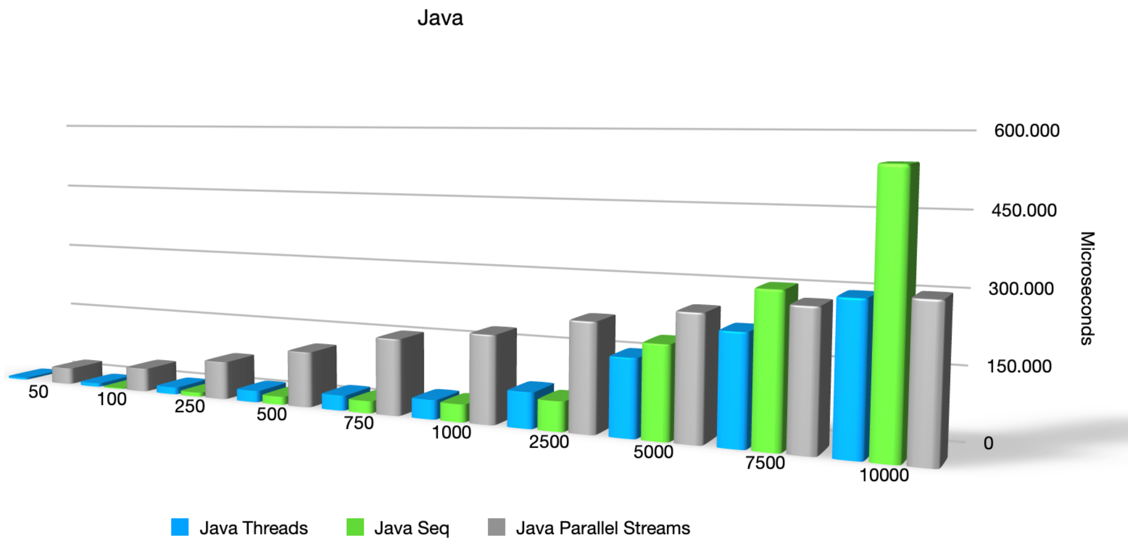
7.1 All implementations



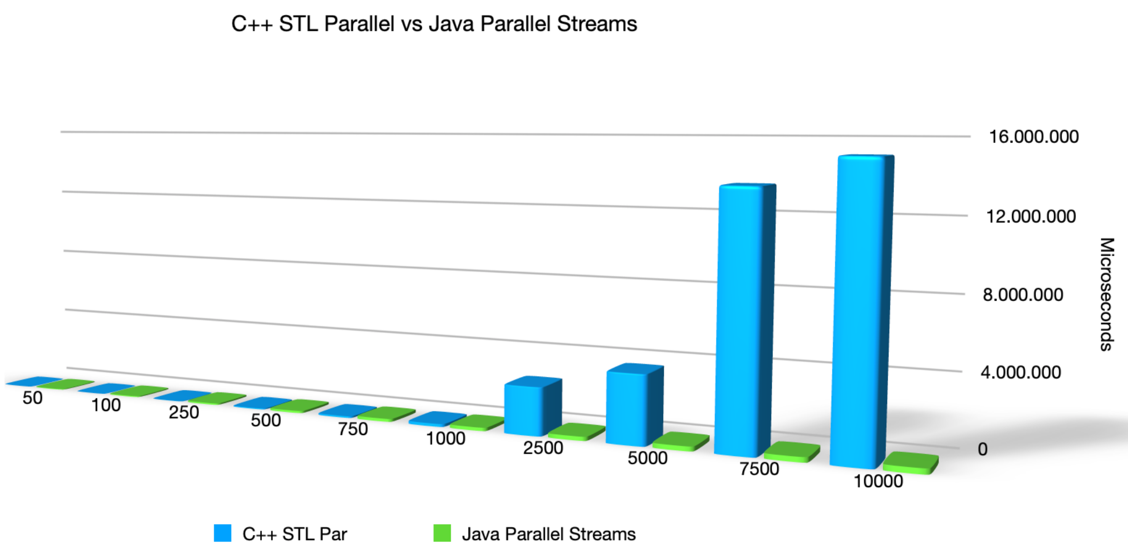
7.2 C++ implementations



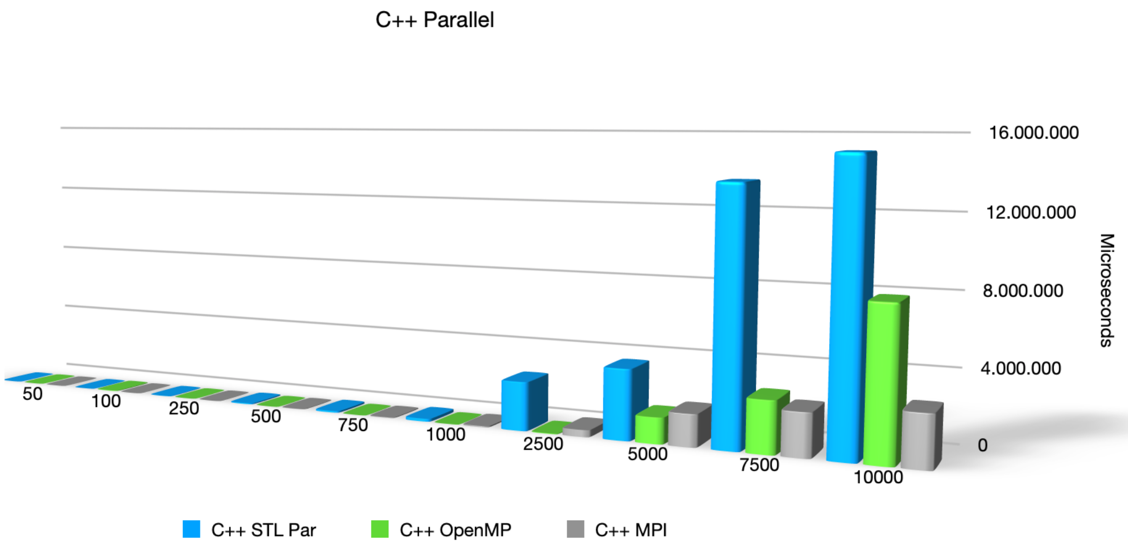
7.3 Java implementations



7.4 C++ STL Parallel vs Java Streams Parallel

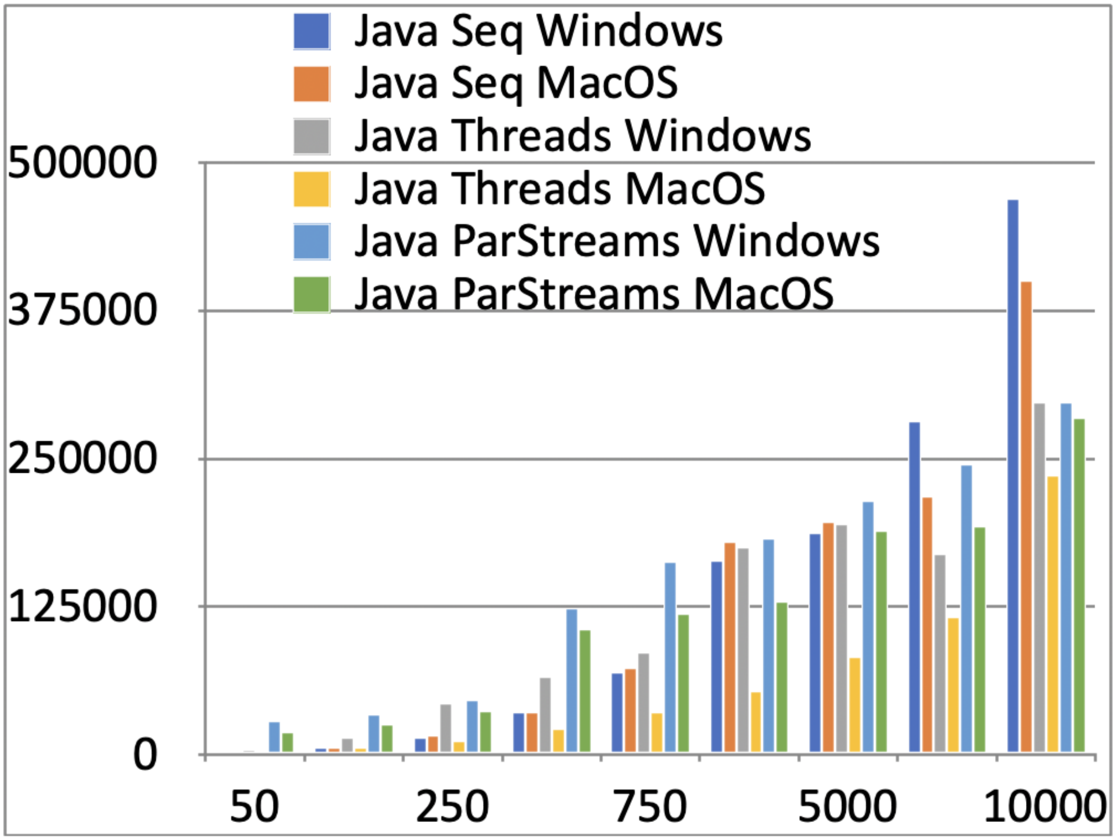


7.5 C++ Parallel



Observation
The environment on which I ran the tests is: Windows 10 Home, 16GB Ram, 512GB SSD, Eclipse for Java programs and Visual Studio 2022 for C++.

7.6 Java - MacOS vs Windows



Observation
Current version was Windows 10 Home and MacOS Sequoia 15.2.

8 Conclusion

The analysis of the Dijkstra algorithm implementations—sequential, multithreaded, and parallel—reveals the following insights:

- For small input sizes, the **sequential implementation** demonstrates faster execution times. This is attributed to the lack of overhead associated with thread creation, synchronization, and inter-thread communication.
- As the input size increases, the **multithreaded and parallel implementations** become significantly more efficient. This is due to their ability to divide the computational workload across multiple cores, effectively reducing the time spent on intensive operations such as updating distances and selecting the next minimum node.

The turning point occurs when the computational overhead of the parallel approaches is outweighed by their ability to process large datasets simultaneously. This makes the multithreaded and parallel versions more suitable for handling high-volume inputs, as they exploit modern hardware capabilities to achieve greater scalability and reduced execution times.

Although surprisingly, the Java language was faster than C++, but this factor can also be influenced by the different implementations of the time measurement functions in the two languages.

The operating systems offer marginal differences, something shown for running programs written in Java, both in Windows and on MacOS. The latter was faster, but also benefited from a newer version and higher hardware performance.

9 Bibliography

References

[1] repository.stcloudstate.ed,https://repository.stcloudstate.edu/cgi/viewcontent.cgi?article=1044&context=csit_etds, accessed in January 2025.

[2] geeksforgeeks,https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/, accessed in January 2025.