Pesquisa Operacional (2023) 43: e272130 p.1-26
doi: 10.1590/0101-7438.2023.043.00272130

Printed version ISSN 0101-7438 / Online version ISSN 1678-5142
www.scielo.br/pope
ARTICLES

PARALLELIZATION OF SHORTEST PATH CLASS ALGORITHMS: A COMPARATIVE ANALYSIS

Flávio Henrique Batista de Souza[1*], Marcos Henrique Gonçalves Abreu[2], Paulo Ricardo Fonseca Trindade[3], Gustavo Alves Fernandes[4], Luiz Melk de Carvalho[5], Braulio Roberto Gomes Marinho Couto[6] and Diva de Souza e Silva Rodrigues[7]

Received February 16, 2023 / Accepted August 28, 2023

ABSTRACT. The problem of finding the shortest path between a source and a destination node, commonly represented by graphs, has several computational algorithms as an attempt to find what is called the minimum path. Depending on the number of nodes in-between the source and destination, the process of finding the shortest path can demand a high computational cost (with polynomial complexity). A solution to reduce the computational cost is the use of the concept of parallelism, which divides the algorithm tasks between the processing cores. This article presents a comparative analysis of the main algorithms of the shortest path class: Dijkstra, Bellman-Ford, Floyd-Warshall and Johnson. The performance of each algorithm was evaluated considering different parallelization approaches and they were applied on general and open-pit mining databases present in the literature. The experimental results showed an improvement in performance of about 55% on the execution time depending on the chosen parallelization point.

Keywords: algorithm parallelization, shortest path algorithms, graphs algorithms.

*Corresponding author
[1]Centro Universitário de Belo Horizonte UNIBH, Belo Horizonte, MG, Brazil – E-mail: flabasouza@yahoo.com.br – https://orcid.org/0000-0002-7157-1535
[2]Centro Universitário de Belo Horizonte UNIBH, Belo Horizonte, MG, Brazil – E-mail: marcos111516@gmail.com – https://orcid.org/0000-0002-2629-3193
[3]Centro Universitário de Belo Horizonte UNIBH, Belo Horizonte, MG, Brazil – E-mail: paulo7trindade@gmail.com – https://orcid.org/0009-0001-2353-3040
[4]Centro Universitário de Belo Horizonte UNIBH, Belo Horizonte, MG, Brazil – E-mail: gustavo.alfer@gmail.com – https://orcid.org/0009-0008-8864-5327
[5]Centro Universitário de Belo Horizonte UNIBH, Belo Horizonte, MG, Brazil – E-mail: luizmelk22@hotmail.com – https://orcid.org/0009-0004-8652-5867
[6]Centro Universitário de Belo Horizonte UNIBH, Belo Horizonte, MG, Brazil – E-mail: coutobraulio@hotmail.com – https://orcid.org/0000-0002-5314-5161
[7]Centro Universitário de Belo Horizonte UNIBH, Belo Horizonte, MG, Brazil – E-mail: divasouz@gmail.com – https://orcid.org/0009-0007-7286-814X

1 INTRODUCTION

The graph theory is an old subject and nowadays it has become a method widely used in various fields of mathematics, computer science, engineering, chemistry, among others. Its application ranges from finding a shortest path or finding the closest object using the minimum path, to the representation of chemical structures of molecules (Shirinivas et al., 2010). It is a powerful tool, but it has a cost, as it requires computational processing power. In case of a deep or wide search being carried out and aiming to go through all the vertices of the graph, the greater the number of vertices and edges in the graph, the greater will be the processing required (Deo, 2004).

Although some problems of finding the shortest path can be solved within milliseconds, other problems can take hours, days, or even years to be solved. According to Ahuja et al. (1988), Ahuja et al. (1989) and Ahuja et al. (1993), the view on the need for studies on shortest paths in giant networks is based on the practical importance of these paths for a variety of applications, such as transport planning, communication routing, logistics and many other complex systems. These shortest paths represent the most efficient route to transport resources or information between specific points in a network, taking into account constraints such as capacity, cost, or travel time. Therefore, understanding and developing efficient algorithms to find shortest paths in giant networks is extremely important to optimize the performance of these systems and improve their overall efficiency.

The network flow models proposed by Ahuja et al. (1988) provide a mathematical framework to represent and analyze shortest path problems in giant networks. These models are based on fundamental concepts, such as graphs, edge capabilities, costs and flows, which allow a precise formulation of problems and the application of linear programming techniques to find optimal solutions. In addition, these models also allow the consideration of additional constraints, such as capacity limitations at certain points in the network or the existence of multiple resources to be transported simultaneously. Thus, the vision presented by Ahuja, Magnanti and Orlin highlights the importance of addressing the challenges related to shortest paths in giant networks, offering a solid theoretical basis and computationally efficient methods to solve these problems optimally. Ahuja et al. (1993) shows some examples of physical networks that can benefit from the shortest path studies in Table 1, with their compositions, structures and types of flows that the structures have.

One possible approach to improve the processing time is to parallelize the execution of the algorithms. As the most popular graph algorithms have a polynomial complexity (Lee & Sun, 2017), the parallelization approach can improve the efficiency and execution time.

It is possible to make the parallelization of a code by using the multiple threads of a computer. The greater the number of simultaneous threads that the computer can process, the faster is the algorithm execution, maximizing the system efficiency to achieve the final result (Iwashita et al., 2017; Dutta et al., 2017; Mullen et al., 2017).

The parallelization of a sequential algorithm is a non-trivial process, because not all parts of the code can be parallelized. Therefore, it is necessary to make a systematic study to find which part

Table 1 – Ingredients of some common physical networks.

Applications	Physical analog of nodes	Physical analog of arcs	Flow
Communication systems	Telephone exchanges, computers, transmission facilities, satellites	Cables, fiber optic links, microwave relay links	Voice messages, data, video transmissions
Hydraulic systems	Pumping stations, reservoirs, lakes	Pipelines	Water, gas, oil, hydraulic fluids
Integrated computer circuits	Gates, registers, processors	Wires	Electrical current
Mechanical systems	Joints	Rods, beams, springs	Heat, energy
Transportation systems	Intersections, airports, rail yards	Highways, railbeds, airline routes	Passengers, freight, vehicles, operators

Source: Ahuja et al., 1993.

of the algorithm can be parallelized, resulting in increased performance and avoiding the output of conflicting results (Lisboa, et al., 2019).

In 2020, it was observed that the performance comparison of different shortest path algorithms is still highly pertinent (Mukhlif & Saif, 2020; AbuSalim et al., 2020; Rachmawati & Gustin, 2020; Madduri et al., 2007; Solomonik et al., 2013; Arranz, 2015), as in the works of (Mukhlif & Saif, 2020) and (AbuSalimet al., 2020), in which such comparisons were performed focusing on the Dijkstra and Bellman-Ford algorithms. Based on that, the present paper aims on the additional analysis of two other important shortest path class algorithms addressed in the literature: Floyd-Warshall and Johnson.

The main objective of this study is to analyze the performance of four algorithms of the shortest path class in their sequential and parallel versions using real databases for the experiment of finding the shortest path. The specific objectives of this work are: to reproduce the algorithms of Dijkstra, Bellman-Ford, Floyd-Warshall and Johnson in a database to perform the analysis of experiments; parallelize the algorithms previous-mentioned at different points in its structure; and, finally, to demonstrate the difference in performance of each one based on the experiments performed.

This work contributes with the analysis of identifying the best point of parallelization in each of the four algorithms of shortest path (Dijkstra, Bellman-Ford, Floyd-Warshall and Johnson). The results obtained demonstrates a significant performance improvement considering the set of experiments used in the analysis. The approach employed in this work for the construction and representation of the algorithms can be useful for the improvement of performance of software that relies on tasks based on finding a shortest path.

2 MATERIAL AND METHODS

This section describes the related works and basic concepts that are relevant to the main research of this article.

2.1 Related works

In the literature, several authors address the parallelism of search algorithms in graph theory and how these algorithms are important in solving real problems (Fan et al., 2020; Hou et al., 2019; Dhulipala et al., 2018). The present work is focused on two main areas regarding this subject. Firstly, the class of algorithms of shortest path is analyzed with the main objective of implementing the parallelization of the algorithms to improve the performance. Then, the second aspect, is to develop a comparison study of each one of the algorithms of the class presented, in order to identify the one with the best result for the tests performed.

The main difference between this work and the papers present in the literature is that, in the present work, the algorithms of a shortest path class are presented in their sequential and parallel versions to demonstrate the performance improvement on this class of algorithms in the parallel versions.

In the work of Junior et al., (2019) a study is carried out comparing five efficient algorithms to solve a shortest path problem. The algorithms were: Dijkstra, Radix Heap, Heap Binomial, L-Threshold and Dijkstra with Heuristics. The main objective was to evaluate the performance of each one of the algorithms when applied to a given problem. The algorithm with best performance was the Dijkstra algorithm with Heuristics.

The research of Madduri et al. (2007) evaluates the Δ-stepping algorithm performance on solving a problem with 100 million vertices by using a high computational resource (40 processors involved). The research developed represents a contribution, considering computational resources limited to only 1 applied processor, with a maximum limit of 200 visitation points. However, the present research demonstrates a real situation for several sectors that have a high impact on the market, such as the open pit mining sector, which contemplates this reality with billions of dollars used every year.

In the work of Hajela & Pandey (2014) it is presented a parallel approach of the Bellman-Ford algorithm based on GPU for different shortest path problems, in which the weights of its edges can have positive or negative values.

Also, Solomonik et al. (2013) demonstrates a structure that optimizes the interconnection between processors during the parallelism process, using as a reference a Cray XE6 supercomputer that used 24,576 cores. From a market perspective, this work contributes with the simple application of the parallelization process that assists a wide range of companies operating in the industrial sector, that do not have access to a powerful computational resource, even using cloud computing resources.

The research in Arranz (2015) explores with broad consideration the parallelism process, including the All-Pair Shortest-Path (APSP) and Single-Source Shortest-Path (SSSP) structures. It focuses on the use of sequential and parallel execution techniques. Among the differences with the research carried out, there is the use of GPUs in their experiments, instead of CPUs, with element evaluation, even analyzing the L1 cache utilization. This work focuses on the use of CPUs that are even used by employees of the sector under analysis.

In the article of Dickow, et al. (2013), it is presented four classic algorithms that are also algorithms covered in the present work, which are: Bellman-Ford, Dijkstra, Floyd-Warshall and Johnson. The objective of the work was to analyze these routing algorithms and perform a comparison based on their applications in WirelessHART networks.

It is worth citing research with such structures with practical applications. As an example, there is the research by Hribar et al. (2001) with the research of shortest paths for parallel transport applications. We also have more recent research, such as Aridhi et al. (2015) applying map reduction to large-scale networks, Selim & Zhan (2016) also applying such perspectives to large networks. Another pertinent reference is the research by Heywood et al. (2019) with an evaluation of paths with multiple sources, with parallelism for a macroscopic analysis of a path assignment in a network.

And finally, in 2020, as shown in the works of Mukhlif & Saif (2020) and AbuSalim et al. (2020), where such comparisons were performed focusing on the Dijkstra and Bellman-Ford algorithms, confirming the relevance of the present paper which aims to add the performance comparison's analysis of two more relevant structures in the literature.

2.2 Theory background

In this subsection it is presented the main concepts related to the parallelism techniques and the shortest path algorithms. The survey on the structures used can be justified here by a brief comparative analysis. According to Gallo & Pallottino (1986), shortest path algorithms may differ in their approaches and resolution strategies. They propose a unified view of shortest path methods, which encompasses classic algorithms such as the Dijkstra algorithm, the Bellman-Ford algorithm and the Floyd-Warshall algorithm, among others. One of the main differences between shortest path algorithms is the way they select and update paths in search of the shortest path. Dijkstra's algorithm, for example, uses a greedy strategy, exploring the paths with the lowest accumulated cost so far. The Bellman-Ford algorithm, on the other hand, allows the presence of edges with negative weight and iterates over all edges in each step, updating the path costs iteratively. The Floyd-Warshall algorithm, in turn, is a dynamic programming method that calculates the shortest paths between all pairs of vertices in a single run.

In addition, other factors that can differentiate shortest path algorithms include the type of graph considered (directed, undirected, weighted, etc.), the presence of additional constraints (such as capabilities or resource limits), computational efficiency, and the ability to handle large networks or dynamic updates to the network topology. In summary, shortest path algorithms can vary in

their approaches, path selection and update strategies, ability to handle different types of networks and constraints, and computational efficiency. The choice of the most suitable algorithm depends on the specific characteristics of the network and the requirements of the problem in question.

2.2.1 Speedup metrics

As previously mentioned, a parallelism technique consists of dividing complex tasks into subtasks, to distribute them among other interconnected processors and executing instructions simultaneously in order to achieve a performance improvement. According to Rafikov et al.(2020), the speedup and efficiency of a parallel algorithm are both metrics relevant to evaluate the performance of a parallelized algorithm. The speedup (S) metric can be calculated according to equation (1).

$$S = \frac{T(1)}{T(N)} \tag{1}$$

where $T(1)$ is the execution time in a single processor (serial time) and $T(N)$ is the time of execution in N processors (parallel time). In summary, the speedup represents the relative benefit in solving a problem with multiple processors. The efficiency (E) represents the fraction of the total time that all processors are used during processing, and it is given by equations (2) and (3).

$$E = \frac{S}{N} = \frac{\frac{T_1}{T_N}}{N} = \frac{T_1}{N.T_N} \tag{2}$$

$$E\varepsilon\,[0,1] \tag{3}$$

where E is the efficiency of the parallel version in relation to the serial, S is the speedup, N is the number of processors.

2.2.2 Graph and distance calculation

A graph is represented as $G = (V,E)$ where V is the set of vertices, and E is the set of edges (Sadiq & Yousaf, 2020; Chu & Wu, 2021; Losqui & Souza, 2019). According to Arranz (2015) and Sedgewick & Wayne (2011), a direct graph (or digraph) is "*a graph* $G = \{(V,E)/(u,v) \neq (v,u) : (u,v),(v,u) \in E\}$ *where the edge (u, v), that connects node u with node v, only can be traversed from u to v, and therefore is different from edge* $(v,\ u)$". The weight of the path $p = (v_0, v_1, v_2, ..., v_k)$ is the sum of the weights of its paths, which is given by equation (4):

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{4}$$

Given two vertices u and v, the weight of shortest path between them is defined as $\delta(u,v) = \min\{wp : u,v$, if such a path exists, or $\delta(u,v) = \infty$ if the path does not exist. For each edge

$(u,v) \in E$ there is the following relationship: $\delta(s,v) \leq \delta(s,u) + w(u,v)$, where s represents the origin vertex. The shortest path estimated between u and v is represented by $d[v]$, where $d[v] \geq \delta(s,v)$ for all vertices $v \in E$, and once $d[v]$ reaches the value of $\delta(s,v)$, it never changes again.

To measure the distance between two points, the Euclidean distance calculation is commonly used, which can be proved by repeated application of the Pythagorean theorem, as shown in equation (5) (where n is the number of points, p and q are the initial and final points of each path) (Losqui & Souza, 2019; Lu et al., 2020).

$$\sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{5}$$

2.2.3 Dijkstra Algorithm

The algorithm described by Dijkstra in 1959 has the purpose of finding the shortest path in a directed or non-directed graph given by $G = (V,E)$. The edge weights must have positive values, $w\ (u,\ v) \geq 0$ for each edge $(u,\ v \in\ E$ (Dijkstra, 1959; Martelli, 1977; Cormen, 2013). According to Arranz (2015), Dijkstra can be defined as an algorithm described in four fundamental steps:

1. *(Initialization) It starts on the source node s, initializing the distance array* $D[i]\ =\ \infty$ *for all nodes i and* $D[s]\ =\ 0$. *Node s is settled and is considered as the frontier node* $f\ (f\ \leftarrow\ s)$, *the starting node for the edge relaxation.*
2. *(Edge relaxation) For every node v adjacent to f that has not been settled, a new distance from source node s is found using the path through f, with value* $D[f] +\ w(f,\ v)$. *If this distance is smaller than the previous value* $D[v]$, *then* $D[v]\ \leftarrow\ D[f]\ +\ w(f,\ v)$.
3. *(Settlement) The non-settled node b with the minimal value in D is taken as the new frontier node* $(f\ \leftarrow\ b)$, *and it is now considered as settled.*
4. *(Termination criterion) If all nodes have been settled, the algorithm finishes. Otherwise, the algorithm proceeds once more to step 2.*

According to Algorithm 1, the execution of the algorithm makes it go through the vertices of a graph G and separating each one of the vertices in two different groups: Q and S. The vertices not visited by the algorithm persist in set Q, being added to set S as the visit is carried out (Dijkstra, 1959; Martelli, 1977; Cormen, 2013).

According to Arranz (2015), there is a process of storing the process of visiting the nodes until reaching the source node.

2.2.4 Bellman-Ford Algorithm

The Bellman-Ford algorithm is the integration of the algorithm described by Bellman (in 1956) with the algorithm described by Ford (in 1956). The objective is to find a shortest path in a

Algorithm 1: Dijkstra

```
1   D (G, s)
2       for each vertex v in G
3           d[v] = ∞                          // initial distance from source to vertex v is set to ∞
4           previous[v] = undefined           // Previous node in optimal path from source
5       d[s]= 0                               // Distance from source to source
6       Q = the set of all nodes in Graph     // all nodes in the graph are unoptimized - thus are in Q
7       while Q ≠ ∅                           // main loop
8           u = node in Q with smallest d[ ]
9           remove u from Q
10          for each neighbor v of u:         // where v has not yet been removed from Q.
11              alt = d[u] + d(u, v)
12              if alt < d[v]                 // Relax (u, v)
13                              d[v] = alt
14                  previous[v] = u
15      return previous[ ]
```

Source: Dijkstra, 1959; Martelli, 1977.

graph with weighted edges . In contrast to Dijkstra method, the algorithm can present edges with negative values (AbuSalim et al., 2020; He et al., 2020). The algorithm solves the problem of shortest path in a directed or non-directed graph given by G = (V, E). During the iterations of the algorithm, a Boolean value is returned, indicating the existence, or not, of a cycle with negative weight. The pseudocode of the algorithm is presented in algorithm 2 considering s as the initial vertex.

Similar to Dijkstra, the algorithm starts by assigning the weight $\delta(s,v) = 0$, if the source vertex is the destination itself, and $\delta(s,v) = \infty$, if a real path from s to v is not yet known. In line 3 it is created a loop visiting all the vertices of the graph in $|V| - 1$ visitations. At each visit, the algorithm performs the relaxation technique on all edges present at each vertex. The relaxation technique is used to progressively decrease the estimate $d[v]$ in the weight of a shortest path from the origin s to each vertex $v \in E$, in order to find the real weight $\delta(s,v)$ of the shortest path. After visiting all the vertices of the graph G, lines 6, 7, 8 and 9 check for a negative weight cycle. The algorithm returns 'TRUE' if, and only if, the graph has no negative weight cycle that is accessible from the origin s.

2.2.5 2.2.5 Floyd-Warshall Algorithm

Published by Floyd in 1962 and based on a theorem described by Warshall in 1962, the Floyd-Warshall algorithm solves the problem of shortest paths between nodes in a directed and non-directed graph given by $G = [V,E]$. The algorithm seeks to find a shortest path for all vertices $(u,v) \in V$, with minimum weight, as well as Bellman-Ford with negative weight edges. The

Algorithm 2: Bellman-Ford

1 $\delta\ (s, v) = 0$ **IF** $s = v$

2 $\delta\ (s, v) = \infty$ **IF** $s \neq v$

3 **FOR** $i = 1$ to $i = |V\ [G]|$

4 **FOR** each edge $(u, v) \in E[G]$

5 **DO** relaxation (u, v, w)

6 **FOR** each edge $(u, v) \in E[G]$

7 **IF** $d\ [v] > d\ [u] + w(u, v)$

8 **RETURN** FALSE

9 **RETURN** TRUE

Source: Dickow, et al., 2013; Weber et al., 2020.

Floyd-Warshall algorithm uses an adjacency matrix to represent the same graph. The element in the matrix of row u and column v is the weight of a shortest path from u to v (Hougardy, 2010).

The operation of the algorithm is based on the multiplication of matrices using the repeated squared technique. It explores the relationship that the number of vertices of the directed weighted graph G is given by $V = \{1, 2, 3, \ldots, n\}$ and the intermediate vertices of the path p between u and v are considered a subset $\{1, 2, 3, \ldots, k\}$. The pseudocode of the Floyd-Warshall algorithm is presented in Algorithm 3, where $M[u, v]$ is a matrix $N \times N$, where N is the number of vertices, and its elements represent the weights of the edges between the vertices u and v (Dickow, et al., 2013).

The execution of the algorithm starts by assigning to the elements of the matrix the weight $M[u, v] = 0$, if the origin is the destination itself in the adjacency matrix. If there is not any path between u and v, the weight $M\ [u,\ v]\ =\ \infty$ is assigned. Lines 3, 4 and 5 form loops that determine the execution time of the algorithm. At each visit to a given vertex, the paths are recalculated, and it is checked whether there is a path with less weight so that the matrix $M\ [u,\ v]$ is updated.

Algorithm 3: Floyd-Warshall

1 $M[u, v]\ =\ 0$ **IF** $u\ =\ v$

2 $M[u, v]\ =\ \infty$ **IF** $u\ \neq\ v$

3 **FOR** $k\ =\ 1$ to $k\ =\ n$

4 **FOR** $u\ =\ 1$ to $u\ =\ n$

5 **FOR** $v\ =\ 1$ to $v\ =\ n$

6 **IF** $M\ [v, u]\ >\ M\ [u, k]\ +\ M[k, v]$

7 **THEN** $M\ [v, u]\ =\ M\ [u, k]\ +\ M[k, v]$

8 **RETURN** $M[u, v]$

Source: Dickow, et al., 2013.

2.2.6 Johnson Algorithm

The algorithm described by Johnson in 1977 finds a shortest path between all vertices of a graph given by $G = [V, E)]$ through an adjacency matrix. It makes use of the Dijkstra and Bellman-Ford algorithms in its subroutines and assumes that the edges E are stored in a list of adjacencies (Dickow, et al., 2013).

The algorithm, presented in Algorithm 4, produces a $V \times V$ matrix D with its elements representing the weight $\delta\ (u,\ v)$ of the path between u and v. The algorithm uses the weighting technique, which considers that if all weights w of edges of a graph G are positive, to find a shortest path between all pairs of vertices by executing the Dijkstra algorithm (Dickow, et al., 2013).

The algorithm starts by the evaluation of the graph $V\ [G\,']$ with the original weight function w, where s is the origin vertex. Initially, the edges $E\ [G\,']$ and the weights $w(s,\ v)$ are also calculated. The Bellman-Ford algorithm is executed in line 4, if the FALSE Boolean value is returned. This indicates that a negative cycle has been found and that it will be declared on line 5.

Lines 6 to 14 assume that the graph $G\,'$ has no negative weight cycle and define h(v) as the weight of the shortest path calculated by the Bellman-Ford algorithm for all $v \in V\,'$. Lines 8 and 9 perform the calculation for the new weights $w\,'$. The loop from lines 10 to 14 calculates the weight of the shortest path $\delta(u,\ v)$ for each pair of vertices $u \in V$ using the Dijkstra algorithm.

Algorithm 4: Johnson

```
1 Calculate V [G'] = V [G] U {s}
2 Calculate E [G'] = E [G] U { (s, v) : v ∈ V [G] }
3 Calculate w (s, v) = 0 para v ∈ V [G]
4 IF Bellman-Ford (G', w, s) = False
5   THEN "There is a negative cycle"
6  FOR EACH v ∈ V [G']
7     DO h(v) throught Bellman-Ford
8   FOR EACH (u, v) ∈ E [G']
9   DO w' (u, v) = w (u, v) + h (u) − h(v)
10  FOR u ∈ V [G]
11    DO Dijkstra (G', w', u)
12       calculate δ( u, v) in v ∈ V G
13    FOR EACH v ∈ V G
14       DO d(uv) ← δ(s, v) +h (v) − h(u)
15 RETURN D
```

Source: Dickow, et al., 2013.

In line 14 of the pseudocode, it is stored at the input of the matrix the estimate $d(u,\ v)$ of the correct weight of the shortest path between u and v. Finally, in line 15, it is returned the matrix D completed with the weights $\delta(u,v)$ of each vertex of the graph G.

2.3 Parallelization approach

The C / C ++ programming languages have a multi-process programming library of shared memory on multiple platforms called OpenMP, that can be used to parallelize the previous mentioned algorithms. OpenMP is a library widely used by programmers with little, if any, experience in parallelism, as it facilitates the creation of threads and requires that the programmer only establishes what needs to be parallelized (Silva & Martins, 2012).

According to Silva & Martins (2012), OpenMP is based on the shared memory programming paradigm, in which the parallelism consists of multiple threads. Thus, it can be said that OpenMP is an explicit parallel programming model, offering total control to the programmer.

The programming model adopted by OpenMP is very portable and scalable and can be used on several platforms, ranging from a personal computer to supercomputers. OpenMP is based on compilation directives, library routines and environment variables.

The first step to parallelize the proposed algorithms was to carry out an analysis of the execution time of their serial versions. The analysis was made based on databases with graphs of different sizes containing the latitude and longitude of different cities, where the Euclidean distance from all cities to all cities was calculated, using the database present in Reinhelt (2014). In this database (table 2), each dataset has a number of vertices (example: base att48 - 48 vertices, rat99 - 99 vertices), which will imply a difference in evaluation times.

Table 2 – Databases

att48	kroB100	lin105	pr144
berlin52	kroB150	pr76	pr152
kroA100	kroB200	pr107	rat99
kroA150	kroC100	pr124	rat195
kroA200	kroD100	pr136	kroE100

Source: Authors.

These instances are stored in text files with a specific structure. Each instance file contains several sections, including general instance information and point coordinate data. Here is a simplified example of how data is stored in the TSPLIB95 base:

1. General information section:
    - NAME: Instance name
    - TYPE: Type of issue (usually ”TSP”)
    - DIMENSION: Number of cities/points in the instance
    - EDGE_WEIGHT_TYPE: Distance matrix type (Euclidean, Geometric, etc.)
2. Coordinate section: Point coordinate data is listed in this section. Typically, each line represents a point/city and contains a unique identifier and the coordinates (x, y) of the point. For example, a TSP instance in base TSPLIB95 might have the following structure:

- makefile
- Copy code
- NAME: example
- TYPE: TSP
- DIMENSION: 4
- EDGE_WEIGHT_TYPE: EUC_2D
- NODE_COORD_SECTION
- 1 20 35
- 2 40 50
- 3 15 25
- 4 10 30

In this example, we have an instance with 4 cities/points, and the coordinates of these points are provided in the NODE_COORD_SECTION section. Each line in this section contains the point identifier followed by the x and y coordinates. Instances in base TSPLIB95 may have additional sections depending on the specific issue and data provided. It is important to consult the official TSPLIB95 documentation for detailed information about the structure and format of specific instance files. This database was used considering the proposed algorithms. For example, in Dijkstra' s algorithm, node "b" is chosen as the next node to be visited from the unresolved nodes with the shortest current distance. To do this, initialize the algorithm by setting the distance from the origin node to 0 and the distance to the other nodes to infinity. In each iteration, we select the unresolved node with the shortest current distance, marking it as resolved. This process is repeated until all nodes are resolved or until the target node is resolved, if applicable. In this way, the algorithm expands towards the unresolved nodes with the shortest distance, ensuring an efficient search in the graph to find the shortest path.

The experiments were carried out using the IDE Visual Studio 2019, which shows the runtime data in detail. The language used on the development of the algorithms was C ++, because it is a very efficient programming language with regards to the performance and memory usage.

With the OpenMP library, it is possible to use the "#pragma omp" directive, that makes the C / C ++ compiler generate the optimized code for the OpenMP execution environment. One of its main directives is the "#pragma omp parallel", which allows to parallelize such part of the algorithm as shown in Algorithm 5 as an example. In the example showed in algorithm 5, the basic OpenMP function "omp_get_thread_num ()" returns the identifier of the current thread, and can then take the control over it. The function omp_get_num_threads () returns the number of threads currently active. Another directive used in this work was "#pragma omp critical", which informs the compiler that a code instruction within its block can only be executed by one thread at a time. For instance, if the programming of an instruction is within a critical section, it means

Algorithm 5: Inserting Parallelism

```
3 ...
4 PRAGMA OMP PARALLEL num_threads(8)
5    n_threads ← omp_get_num_threads()
6    id ← omp_get_threads_num()
7    FOR i ← 0 to num_steps
8       local_count ← 0
9       FOR j ← 0 to ...
```

Source: Authors.

that it is only possible to program one instruction at a time in the system, otherwise it will be returned inconsistent and disconnected results.

The tests were performed on a notebook with the Intel (R) Core (TM) i7-8565U 1.8GHz processor, 8GB of RAM and 256GB PCIe NVMe SSD.

3 RESULTS

The following subsections present the main results of analysis made on the previous mentioned algorithms.

3.1 Preliminary Analysis of Performance

In order to describe the process to obtain the results related to the performance of the shortest path algorithms in their parallel execution in comparison to their serial execution, it is presented in this subsection a preliminary analysis of the performance of a code to calculate the value of π. The algorithms that represent its serial and parallel version are presented in algorithm 6 and 7, respectively.

Algorithm 6: Calculating the value of π – serial version

```
1 BEGIN
2 x, pi, sum ← 0
3 step ← 1 / num_steps
4 FOR i ← 0 to num_steps
5    n ← (i + 0.5) * step
6    sum ← sum + (4 / (1 + (x*x)))
7 END FOR
8 pi ← step * sum
9 PRINT("Result: ", pi)
10 RETURN 0
11 END
```

Source: Authors.

Algorithm 7: Calculating the value of π – parallel version

```
1 BEGIN
2 x, pi, global_sum ← 0
3 step ← 1 / num_steps
4 PRAGMA OMP PARALLEL num_threads(8)
5    n_threads ← omp_get_num_threads()
6    id ← omp_get_threads_num()
7    local_sum ← 0
8    FOR i ← 0 to num_steps
9       x ← (i + 0.5) * step
10       local_sum ← local_sum + 4 / (1 + x * x)
11   END FOR
12 END PARALLEL
13 PRAGMA OMP CRITICAL
14    global_sum ← global_sum + local_sum
15 END CRITICAL
16 pi ← step * sum
17 PRINT("Result: ", pi)
18  RETURN 0
19 END
```

Source: Authors.

The execution time spent in the serial version was 8.438 seconds. However, for the parallel version with 8 threads the execution time was 1.256 seconds, which represents an execution time reduction of more than 80%.

3.2 Analysis of Serial and Parallel Versions of Algorithms

The shortest path algorithms were parallelized in different parts of their code with focus on the parts that represents a significant impact on the computational cost such as loops, comparisons and math calculations. The algorithms were applied on the database previously mentioned in section 2. Each algorithm was executed 500 times on the database to guarantee an average analysis of their performance.

The first test was executed with the Dijkstra algorithm (see Figure 1), which had a code parallelization at three different points. Firstly, in line 10, the internal loop was parallelized, but it resulted in a poor performance, with a 50% drop in the execution time and a poor performance when compared to its serial version. This drop in performance was due to the algorithm having to redo the calculation of the distribution of the parallelization point, according to the number of vertices of the graph and because it is inside a loop that is executed V times, with V being the number of vertices.

Another parallelization alternative was implemented in line 8, where the minimum distance is calculated. This approach resulted in the worst performance on the tests considered, being the one that spent the longest execution time, with a drop of about 57% on its execution time when

compared to the serial version. Based on that, similar to the test performed in the inner loop, this one had a worse result because it is also inside an outer loop and, therefore it is a simpler part of the algorithm, in which the parallelization on a single line (line 8) brings no benefit.

The test with the best result occurred in the parallelization of line 5, which corresponds to the outermost loop of the code. This test achieved a considerable improvement in the execution time, resulting in about 65% speed up when compared to the serial version. This last test achieved the best result because the calculation to distribute the threads was performed only once, and, as it represents a large part of the algorithm, it resulted in a better use of the parallelization approach.

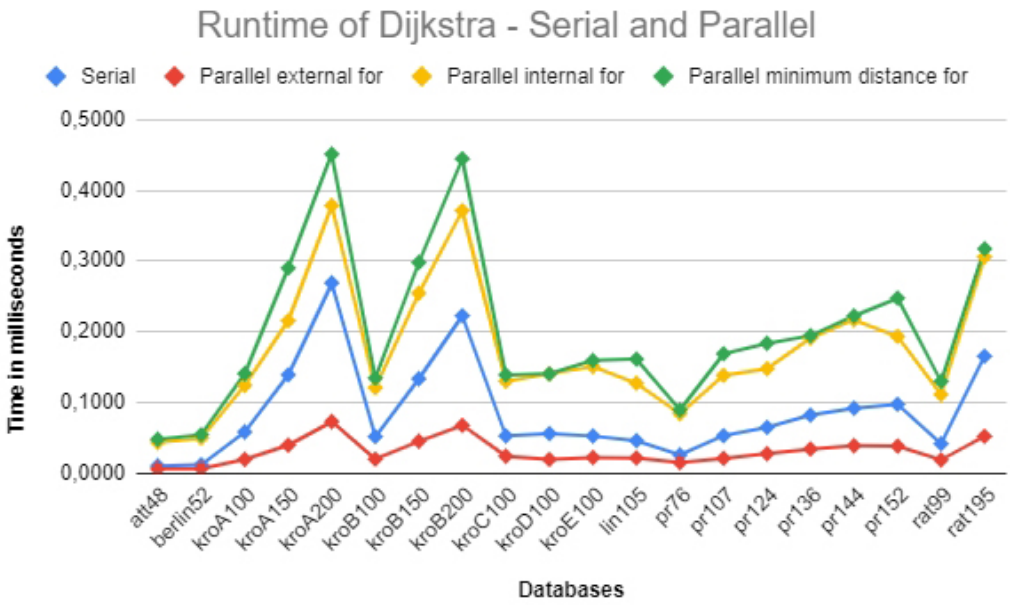

Figure 1 – Dijkstra comparison chart.

Source: Authors.

The data presented in Figure 2 shows the statistical mean of the execution time of the Dijkistra algorithm parallelized at different points on all the databases. Most of the results related to the time execution of the algorithms at different points of parallelism are concentrated between 0.05 ms and 0.2 ms, with a maximum limit of approximately 0.45 ms, which represents a better result than in the serial version.

The second test was performed on the Bellman-Ford algorithm (see Figure 3), which similarly to the approach applied on the Dijkstra algorithm, was parallelized at the same 3 parts of the code. The first parallelization, in line 6, that contains the loop responsible for checking the presence of a negative cycle, resulted in a 35% performance improvement when compared to the serial version. The complexity of this part of the algorithm is not significant and, because of that, the time spent to distribute the threads is longer than the execution of the code itself.

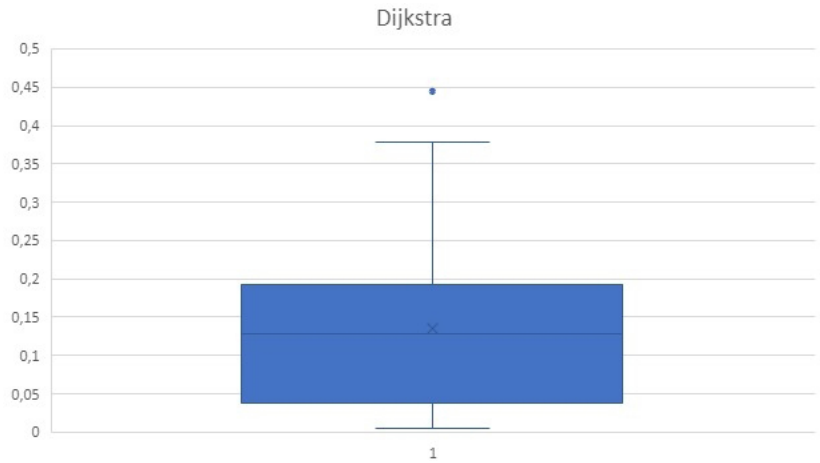

Figure 2 – Base processing times by the Dijkstra algorithm (in milliseconds) where the boxplot demonstrates a minimum processing value of 0.005ms and an approximate maximum value of 0.375, with a concentration of test times between 0.05 and 0.2 ms.

Source: Authors.

The second point of parallelism was in line 4, in the innermost loop responsible for the relaxation calculation. The parallelization at this point of the algorithm resulted in the worst performance with a drop of about 73% compared to its serial version. The drop in performance came from the multiple calculations of the distribution of the parallelization point threads, that the algorithm had to perform according to the number of edges of the graph. In addition, this calculation is inside a loop that is executed E times, being E the number of edges, which causes a greater impact on the performance.

The best result was obtained by parallelizing line 3, in the outermost loop of relaxation with a 66% improvement in performance. This last test, as well as in Dijkstra algorithm, presented the best result because the calculation to distribute the threads was performed only once, and, as it represents a significant part of the algorithm, it was advantageous to apply of the parallelization process.

In Figure 4, most of the processed data time is between 2 ms and 18 ms, which is significantly higher in relation to its serial version, which had its maximum limit below 1 ms according to the graph in Figure 1. Therefore, it is important to identify the point of parallelism that provides an improvement in performance.

The third test was performed with the Floyd-Warshall algorithm (see Figure 5), which, such as the previous ones, was parallelized in 3 different parts of the code. On line 5, in the innermost loop, was performed a parallelization and at this point of the algorithm the worst performance was obtained with a loss of about 80% in performance when compared to the serial version. The loss of performance at this point was due to the calculation of the distribution of threads, which

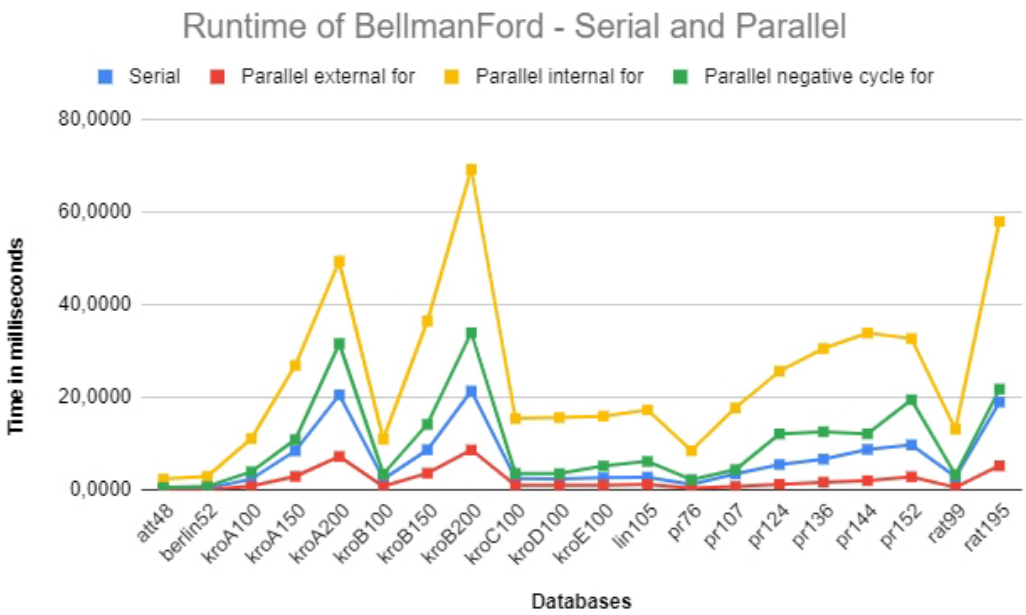

Figure 3 – Bellman-Ford comparison.

Source: Authors.

is performed V^2 times, because the inner loop is inside of two other loops that execute V times each.

On line 4, in the middle loop, the parallelization at this point also resulted in a drop of performance, but not as expressive as the previous test. The drop in performance was of about 27%, because, in this case, the distribution calculation was performed V times.

The best result obtained for this test came from the parallelization of line 3, in the outermost loop, in which was achieved a 37% gain in the execution time when compared to the serial version. Similar to the other algorithms, this point of parallelization resulted in the best result, because the calculation to distribute the threads was performed only once. In addition, as this part of the code represents most of the algorithm, it was possible to achieve a better outcome from the use of parallelization.

Figure 6 shows an analysis similar to the Bellman-Ford algorithm. In this analysis it is important to consider the part of the code that was parallelized to guarantee that there is no loss of performance, because its serial version had a maximum limit of about 17 ms. As can be seen in Figure 6, most of the data performance is concentrated between 5 ms and 18 ms, with outliers still representing the execution of the biggest bases in its worst parallelization part, reaching about 50 ms of execution.

The fourth test was performed using the Johnson algorithm (see Figure 7). As this algorithm uses the Dijkstra and Bellman-Ford algorithms in its execution, the tests were performed with these

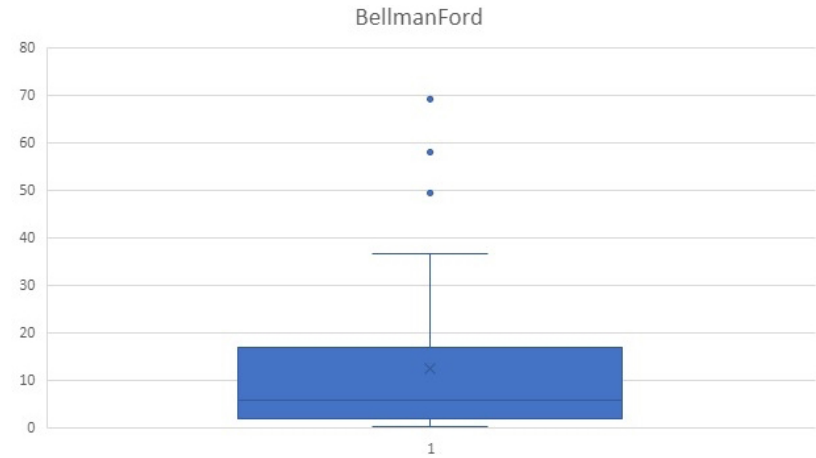

Figure 4 – Base processing times by the Bellman-Ford algorithm (in milliseconds) where the boxplot demonstrates a minimum processing value of 1ms and an approximate maximum value of 36.6ms (with a specific sample of 70ms), with a concentration of test times between 2 and 18 ms.

Source: Authors.

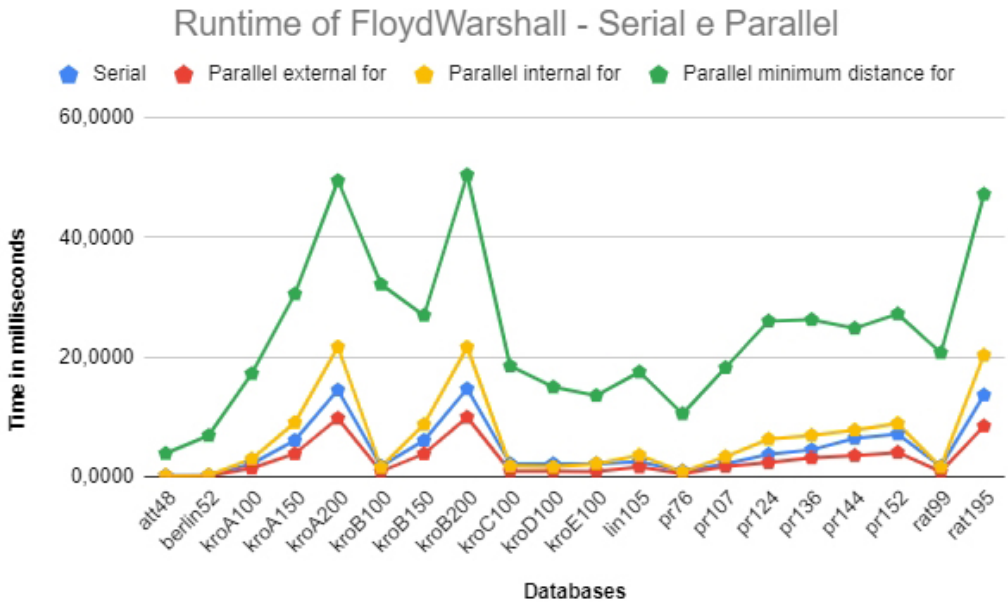

Figure 5 – Floyd-Warshall comparison chart.

Source: Authors.

algorithms in their parallel versions, that corresponded to the best performance obtained in the previous tests.

By parallelizing the Dijkstra algorithm on line 11, it was possible to obtain an improvement in performance of about 3%, while the parallelization of the Bellman-Ford algorithm resulted

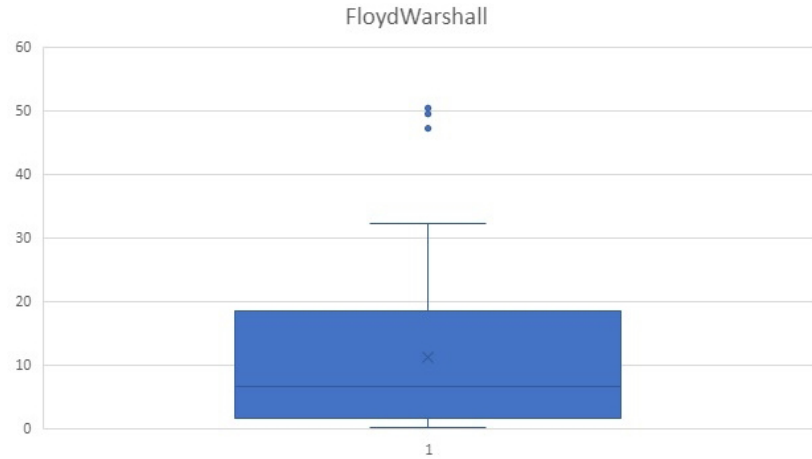

Figure 6 – Base processing times by the Floyd-Warshall algorithm (in milliseconds) where the boxplot demonstrates a minimum processing value of 0.5ms and an approximate maximum value of 33ms (with a specific sample of 50ms), with a concentration of test times between 5 and 18 ms.

Source: Authors.

in an improvement even greater, reaching a gain in performance of about 54%. In Figure 8, it is observed that, on average, the bases executed in parallel versions took about 300 ms to be executed, with a minimum limit of 0.1 ms and a maximum limit of 850 ms.

The outliers showed in the graph came from the execution of the larger databases. For these databases, the difference in performance for the parallelization of the Dijkstra algorithm compared to the parallelization of the Bellman-Ford algorithm becomes more significant, with the Bellman-Ford algorithm having a better performance. In summary, by analyzing the results of the tests performed, it is possible to notice that, if the point of parallelism in the code is not strategically selected, it can result in a loss of performance. In contrast, if the correct part of the code is parallelized, a considerable improvement (up to 55%) in performance is achieved. The results obtained by the shortest path between the initial and final position of each graph were the same for all algorithms (see Table 3), concluding that in terms of information, all algorithms presented an accurate result.

The provided situation illustrates a comparison of execution times for different graph algorithms on a specific instance (kroA150). These algorithms are used to solve a problem related to graphs, possibly finding the shortest paths between nodes. The execution times are measured in milliseconds. The significant discrepancies between the execution times are as follows:

- Dijkstra: This algorithm has the shortest execution time among the listed algorithms. It takes approximately 0.275 milliseconds in the serial case and around 0.075 milliseconds in the best parallelized case.

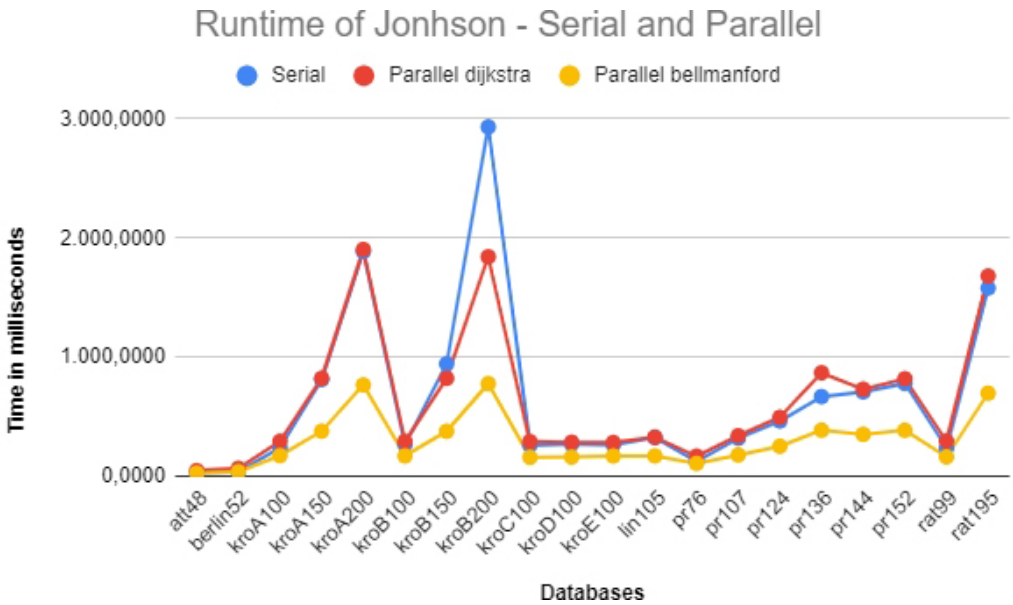

Figure 7 – Johnson comparison chart.

Source: Authors.

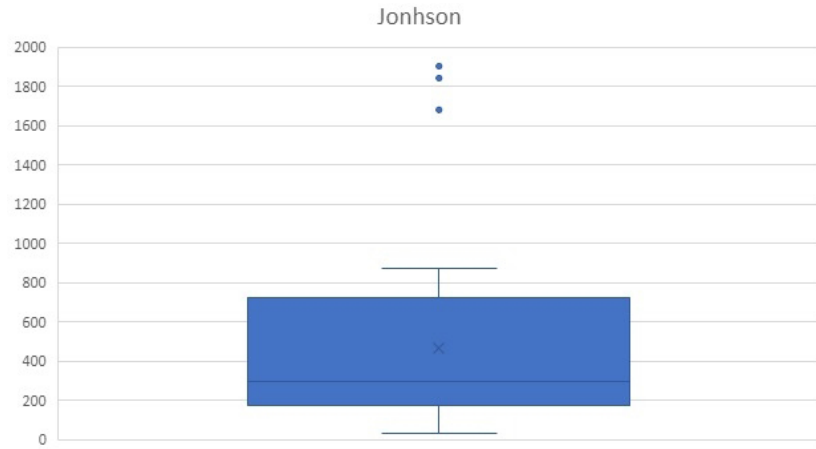

Figure 8 – Base processing times by the Johnson algorithm (in milliseconds) where the boxplot demonstrates a minimum processing value of 0.1ms and an approximate maximum value of 850ms (with a specific sample of 1900ms), with a concentration of test times between 170 and 750 ms.

Source: Authors.

Table 3 – Shortest path of Algorithms.

	DIJKSTRA	BELLMAN-FORD	FLOYD-WARSHALL	JOHNSON
att48	3,743.079102	3,743.079102	3,743.079102	3,743.079102
berlin52	1,220.460938	1,220.460938	1,220.460938	1,220.460938
kroA100	2,643.494141	2,643.494141	2,643.494141	2,643.494141
kroA150	1,382.167114	1,382.167114	1,382.167114	1,382.167114
kroA200	2,616.114990	2,616.114990	2,616.114990	2,616.114990
kroB100	1,248.565552	1,248.565552	1,248.565552	1,248.565552
kroB150	2,096.789307	2,096.789307	2,096.789307	2,096.789307
kroB200	3,117.489258	3,117.489258	3,117.489258	3,117.489258
kroC100	1,650.458374	1,650.458374	1,650.458374	1,650.458374
kroD100	2,718.802979	2,718.802979	2,718.802979	2,718.802979
kroE100	3,496.164307	3,496.164307	3,496.164307	3,496.164307
lin105	1,742.839600	1,742.839600	1,742.839600	1,742.839600
pr76	3,716.180908	3,716.180908	3,716.180908	3,716.180908
pr107	10,385.687500	10,385.687500	10,385.687500	10,385.687500
pr124	9,411.224609	9,411.224609	9,411.224609	9,411.224609
pr136	9,070.000000	9,070.000000	9,070.000000	9,070.000000
pr144	10,949.486328	10,949.486328	10,949.486328	10,949.486328
pr152	13,719.090820	13,719.090820	13,719.090820	13,719.090820
rat99	215.037201	215.037201	215.037201	215.037201
rat195	304.401062	304.401062	304.401062	304.401062

Source: Authors.

- Bellman-Ford: In the serial case, this algorithm takes approximately 20 milliseconds, which is about 72 times the execution time of Dijkstra. In the parallelized case, it takes around 7.5 milliseconds, about 100 times the execution time of Dijkstra.
- Floyd-Warshall: In the serial case, this algorithm takes approximately 15 milliseconds, roughly 54 times the execution time of Dijkstra. In the parallelized case, it takes around 10 milliseconds, approximately 133 times the execution time of Dijkstra.
- Johnson: In the serial case, this algorithm takes approximately 1900 milliseconds, which is about 7000 times the execution time of Dijkstra. In the parallelized case, it takes around 750 milliseconds, approximately 10000 times the execution time of Dijkstra.

These discrepancies in execution times suggest that the algorithms are performing differently on the given instance. While Dijkstra is the fastest algorithm in both serial and parallel cases, other algorithms like Bellman-Ford, Floyd-Warshall, and Johnson exhibit significantly longer execution times. Possible factors could include the algorithm' s design, the nature of the graph, the complexity of the problem, and the efficiency of parallelization techniques used. Certainly, let' s correlate the provided execution time results for the algorithms on the kroA150 instance with the results from the base att48 instance. The instances seem to differ, but we can still analyze the patterns between the two datasets.

On the att48 instance, the execution times are as follows:

Serial Case:

- Dijkstra: Approximately 0.0100 milliseconds
- Bellman-Ford: Approximately 0.3051 milliseconds
- Floyd-Warshall: Approximately 0.2362 milliseconds
- Johnson: Approximately 33.4489 milliseconds

Parallel Case:

- Dijkstra: Approximately 0.0057 milliseconds
- Bellman-Ford: Approximately 0.1172 milliseconds
- Floyd-Warshall: Approximately 0.1170 milliseconds
- Johnson: Approximately 31.2480 milliseconds

When comparing the results between the kroA150 instance and the att48 instance:

- In both instances, Dijkstra consistently has the shortest execution time, and it is also faster in the parallel case compared to the serial case. This suggests that Dijkstra' s algorithm is well-suited for these instances and benefits from parallelization.
- Bellman-Ford and Floyd-Warshall exhibit similar patterns in both instances. In the serial case, they have longer execution times compared to Dijkstra, and in the parallel case, they are faster but still slower than Dijkstra. These algorithms seem to be affected by the size and complexity of the graph instances.
- Johnson' s algorithm shows a significant increase in execution time in both instances compared to the other algorithms. It takes much longer than the rest in both serial and parallel cases. Additionally, it appears that the execution time of Johnson' s algorithm is consistent across instances, despite the differences in instance size.

Overall, the att48 instance results exhibit similar trends to those of the kroA150 instance, even though the absolute execution times differ due to the nature of the instances. These trends can provide insights into how the algorithms perform in general, their sensitivity to instance characteristics, and the impact of parallelization. Across the 20 tested databases, a consistent pattern emerged where the algorithms' average behaviors in terms of execution speed remained relatively constant. Despite variations in absolute execution times, the algorithms maintained their relative ranking in speed. Dijkstra consistently performed fastest, followed by Bellman-Ford and

Floyd-Warshall, while Johnson consistently took the most time. This stability suggests that the algorithms' inherent efficiencies were consistent, although their performance varied due to differences in graph structures and sizes in each database. This underlines the algorithms' sensitivity to graph characteristics and the importance of choosing the right algorithm for specific problem instances based on their performance profiles.

4 CONCLUSIONS

The algorithms of the shortest path class – Dijkstra, Bellman-Ford, Floyd-Warshall and Johnson – already have a remarkable performance, being capable of taking milliseconds to execute in considerably large graphs and solving most of the application problems in real life. However, in the study presented it was identified that there is a significant improvement in its parallelized versions.

After the parallelization analysis, the improvement observed by executing the algorithms in different databases was of an average gain of about 55% on all algorithms. In real-time applications, where it is possible to have thousands of requests made per second, the gain obtained with the parallelization can represent a decrease in the response time of the system, which can represent a positive impact on the savings of the companies.

Finally, the effort used to parallelize the algorithms would have benefits, because it was used techniques from the OpenMP library of C / C ++, which has implemented paradigms to facilitate the parallelization of serial codes.

References

AbuSalim S, Ibrahim R, Saringat M, Jamel S & Wahab J. 2020. Comparative Analysis between Dijkstra and Bellman-Ford. In: *International Conference on Technology, Engineering and Sciences*. p. 1–11.

Ahuja R, Magnanti T & Orlin J. 1988. *Network flows*.

Ahuja R, Magnanti T & Orlin J. 1989. *Handbooks in operations research and management science*.

Ahuja R, Magnanti T & Orlin J. 1993. *Network flows: theory, algorithms and applications*. Prentice Hall.

Aridhi S, Lacomme P, Ren L & Vincent B. 2015. A MapReduce-based approach for shortest path problem in large-scale networks. *Engineering Applications of Artificial Intelligence*, **41**: 151–165.

Arranz H. 2015. *Parallel approaches to shortest-path problems for multilevel heterogeneous computing*. P. lhD Thesis.

Chu D & Wu C. 2021. May. Generalizing the over operator for parallelization and order-independency. *Journal of Parallel and Distributed Computing*, p. 52–60.

CORMEN T. 2013. *Desmistificando algoritmos*. Elsevier.

DEO N. 2004. *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall of India.

DHULIPALA L, BLELLOCH G & SHUN J. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*.

DICKOW V, MULLER I, WINTER J, KUNZEL G, MACHADO T, NETTO J & PEREIRA C. 2013. *Avaliação da Aplicabilidade de Algoritimos Clássicos de Roteamento em Redes WirelessHART*. XI Simpósio Brasileiro de Automação.

DIJKSTRA E. 1959. *Communication with an Automatic Computer*. P. lhD thesis in Computer Science.

DUTTA S, CADAMBE V & GROVER P. 2017. Coded convolution for parallel and distributed computing within a deadline. In: *IEEE International Symposium on Information Theory (ISIT)*.

FAN W, HE K, LI Q & WANG Y. 2020. *Graph algorithms: parallelization and scalability*. Science China Information Sciences.

GALLO G. 1986. Shortest path methods: A unifying approach. In: ED (Ed.), *Mathematical Programming Studies*. p. 26. Springer.

HAJELA G & PANDEY M. 2014. Parallel Implementations for Solving Shortest Path. *International Journal of Computer Applications*, p. 1–6.

HE G, ZHANG X, SUN Y, LUO G & CHEN L. 2020. Contour line simplification method based on the two-level Bellman–Ford algorithm. *Transactions in GIS*, .

HEYWOOD P, MADDOCK S, BRADLEY R, SWAIN D, WRIGHT I & MAWSON M. 2019. A dataparallel many-source shortest-path algorithm to accelerate macroscopic transport network assignment. *Transportation Research Part C: Emerging Technologies*, **104**: 332–347.

HOU N, YAN X & HE F. 2019. A survey on partitioning models, solution algorithms and algorithm parallelization for hardware/software co-design. *Design Automation for Embedded Systems*, p. 57–77.

HOUGARDY S. 2010. The Floyd-Warshall algorithm on graphs with negative cycles. *Information Processing Letters*, p. 279–281.

HRIBAR M, TAYLOR V & BOYCE D. 2001. Implementing parallel shortest path for parallel transportation applications. *Parallel Computing*, **27**: 1537–1568.

IWASHITA T, IDA A, MIFUNE T & TAKAHASHI Y. 2017. Software Framework for Parallel BEM Analyses with H-matrices Using MPI and OpenMP. *Procedia Computer Science*, p. 2200–2209.

JUNIOR J, PONTES H & ALBERTIN M. 2019. Development of educational software to support location and routing teaching [Desenvolvimento de um software educacional para apoio ao ensino de localização e roteirização. *Exacta*, p. 81–99.

LEE Y & SUN H. 2017. An SDP-based algorithm for linear-sized spectral sparsification. In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. p. 678–687.

LISBOA A, SOUZA F, RIBEIRO C, MAIA C, SALDANHA R, CASTRO F & VIEIRA D. 2019. *On Modelling and Simulating Open Pit Mine Through Stochastic Timed Petri Nets*. IEEE. 112821-112835 pp.

LOSQUI H & SOUZA F. 2019-09. Analysis Of Random Points As A Strategy For Local Optimal Improvement In A Constructive Heuristic [Análise de pontos de aleatoriadade como estratégia para melhoria de ótimos locais em uma heurística construtiva. *Revista Produção Online*, p. 923–951.

LU S, ZHANG M & LI Q. 2020. Feasibility and A Fast Algorithm for Euclidean Distance Matrix Optimization with Ordinal Constraints. *Computational Optimization and Applications*, p. 535–569.

MADDURI K, BADER D, BERRY J & CROBAK J. 2007. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In: *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX).Society for Industrial and Applied Mathematics.p*. p. 23–35.

MARTELLI A. 1977. On the complexity of admissible search algorithms. *Artificial Intelligence*, p. 1–13.

MUKHLIF F & SAIF A. 2020. Comparative Study On Bellman-Ford And Dijkstra Algorithms. In: *International Conference on Communication, Electrical and Computer Networks (ICCECN 2020)*. p. 1–5.

MULLEN J, BYUN C, GADEPALLY V, SAMSI S, REUTHER A & KEPNER J. 2017. Learning by doing, High Performance Computing education in the MOOC era. *Journal of Parallel and Distributed Computing*, p. 105–115.

RACHMAWATI D & GUSTIN L. 2020. Analysis of Dijkstra's Algorithm and A* Algorithm in Shortest Path Problem. *Journal of Physics Conference Series*, p. 1–7.

RAFIKOV R, SILSBEE K & BOOTH R. 2020. A Fast O(N2) Fragmentation Algorithm. *The Astrophysical Journal Supplement Series*, .

REINHELT G. 2014. TSPLIB: a library of sample instances for the tsp (and related problems) from various sources and of various types. Available at: http://comopt.ifi.uniheidelberg.de/software/TSPLIB95.

SADIQ M & YOUSAF M. 2020. Distributed Algorithm for Parallel Edit Distance Computation. *Computing and Informatics*, p. 757–779.

SEDGEWICK R & WAYNE K. 2011. *Algorithms*. Addison-Wesley Professional.

SELIM H. 2016. Towards shortest path identification on large networks. *J Big Data*, **3**(10).

SHIRINIVAS S, VETRIVEL S & ELANGO N. 2010. Applications of graph theory in computer science an overview. *International journal of engineering science and technology*, **2**(9): 4610–4621.

SILVA H & MARTINS C. 2012. Avaliação de implementações do Algoritmo Genético Paralelo para Solução do Problema do Caixeiro Viajante usando OpenMP e Pthreads. In: *Workshop de iniciação científica evento integrante do XIII Simpósio em Sistemas Computacionais WSCAD-SSC*. pp. 1–4.

SOLOMONIK E, BULUÇ A & DEMMEL J. 2013. Minimizing Communication in All-Pairs Shortest Paths. In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. p. 548–559. IEEE Computer Society.

WEBER A, KREUZER M & KNOLL A. 2020. A generalized Bellman-Ford Algorithm for Application in Symbolic Optimal Control. In: *Proc. European Control Conference (ECC)*. p. 2007–2014.

How to cite

SOUZA FHB, ABREU MHGA, TRINDADE PRF, FERNANDES GA, CARVALHO LM, COUTO BRGM & RODRIGUES DSS. 2023. Parallelization Of Shortest Path Class Algorithms: A Comparative Analysis. *Pesquisa Operacional*, **43**: e272130. doi: 10.1590/0101-7438.2023.043.00272130.