

Parallel BFS for Large-Scale Graph Processing and Smart Device Computing Frameworks

Esra'a Alhenawi^{1*}, Abdulla Aburomman², Omar Alidmat³, Hebatullah Khattab⁴

Department of Information Technology-Faculty of Prince Al-Hussein Bin Abdallah II for Information Technology,
Al al-Bayt University, Mafrqa, Jordan¹

Department of Cybersecurity and Cloud Computing-Faculty of Information Technology,
Applied Science Private University, Amman, Jordan²

Faculty of Information Technology, Zarqa University, Zarqa, Jordan^{3,4}

Abstract—Large-scale graph processing and analysis have become crucial problems in the big data era, affecting a variety of fields like social networks, road, and web networks. Traditional Breadth-First Search (BFS) algorithms frequently scale inefficiently due to memory constraints and computational bottlenecks. This paper presents a parallel version of the BFS (PBFS) algorithm using two different frameworks, Hadoop/MapReduce, and concurrency. Performance has been evaluated over graphs of different sizes based on runtime, speedup, efficiency, and cost metrics. This paper provides practical guidelines for selecting appropriate graph processing frameworks depending on graph size and system architecture. The results indicate that increasing the number of cores improves speedup but reduces efficiency due to overhead and resource contention. For small and medium sized datasets, BFS implemented with a concurrency framework on a single machine outperforms Hadoop MapReduce because communication overhead is negligible and total execution time depends mainly on computation. In addition, from a computing standpoint of a smart device, single core and dual core configurations represent a constrained environment for IoT and edge devices, which suggests that the concurrency model will be an efficient PBFS solution for this type of environment. In contrast, for large-scale datasets, Hadoop MapReduce achieves better performance by efficiently distributing computation across multiple nodes.

Keywords—Parallel; parallel breadth first search algorithm; Hadoop; concurrency; smart device computing

I. INTRODUCTION

The modern era has witnessed the emergence of the big data concept. Large-scale social networks such as Facebook, Twitter, and LinkedIn are examples of the main sources of big data. These social networks are made up of billions of users and connections around the world. The use of graph structure to analyze and understand social interaction data has been gaining great attention among researchers [1], [2], [3].

Users of social media represent the graph nodes, and their social networks represent the graph edges [4], [5], [6], [7]. These graphs are growing rapidly in terms of size and complexity, where in these applications anyone can create an account in real name or nickname, interact with others through comments, likes, shares, add photos, reports, books, statuses, and videos, which leads to a large number of nodes and edges in the representative graph [8], [9], [10]. A large number

of people worldwide use social networks. Moreover, graphs became one of the most common data structures to represent data in other applications such as transportation routes and Internet infrastructure architectures [11], [12], [13], [14], [15].

On the other hand, the demands from people to discover specific and general facts from the huge amount of information that is stored in these graphs have increased. Analysis of such huge graphs that consist of more than billions of nodes and multiple terabytes or even petabytes of edges in an acceptable time represents a great challenge that brings a lot of attention from many researchers because traditional approaches for graph searching, which are based on serial algorithms, become impractical and insufficient to handle big data anymore [16]. This situation leads to thinking about converting a sequential graph search algorithm to a parallel one and running it in parallel [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. The infrastructure to support this idea became more applicable with cloud computing, and the Hadoop MapReduce open source computational framework that emerged during the last decades [27], [28], [29], [30], [31].

Breadth-First Search (BFS) plays a fundamental role in a variety of graph analytics tasks, including shortest path computations, influence propagation, and connectivity analysis [32], [33], [34], [35]. However, traditional BFS algorithms are inherently sequential, making them unsuitable for processing massive graphs that cannot fit into the memory of a single machine. To address this limitation, parallel and distributed BFS algorithms have been developed to take advantage of modern computing infrastructures [36].

Parallel Breadth-First Search (BFS) can be conducted in different ways depending on the architecture used, such as a shared memory system by using threads or GPU warps, where all threads can directly access shared data structures, while the synchronization primitives ensure correct updates [37], [38], [39], or a distributed system. In these systems, the graph is partitioned across multiple machines (nodes) where each node processes its part of data that are exchanged over the network using HDFS writes, since each iteration involves multiple MapReduce rounds (Hadoop) [40], [41].

This paper presents an approach for efficiently implementing traversals of large-scale graphs over Hadoop MapReduce (distributed memory), and multi-threading frameworks (shared memory), based on the Breadth First Search (BFS) for visiting

*Corresponding author.

the graphs, and then compare their results based on running time, speedup, efficiency, and cost.

The contributions of this paper can be summarized in the following points.

- Implementation of PBFS using Hadoop MapReduce.
- Implementation of PBFS using multithreaded concurrency.
- Comparison of the performance of the PBFS algorithm in Hadoop/MapReduce frameworks, and concurrency in graphs of different sizes.
- Analysis using runtime, speedup, efficiency, and cost metrics.

The remaining parts of the paper are structured as follows: Section II provides background information. Section III presents the related works in the recent literature, while Section IV describes the methodology that had been used. Section V presents the experimental setup. Section VI presents and discusses the experimental results in terms of runtime, speedup, efficiency, and cost. Section VII discusses the limitations of this study and outlines directions for future work. Finally, Section VIII concludes the paper.

II. BACKGROUND

In this section an overview of the main subjects in this paper is presented, such as cloud computing and big data in Section II-A, and the Hadoop MapReduce Framework in Section II-B, while Section II-C presents the background information about the Concurrency Framework, and Section II-D introduces Smart Device Computing.

A. Overview of Cloud Computing and Big Data

Cloud computing is a solution to big data processing that provides shared resources, storage, servers, and services to achieve cost efficient and scalable services [42]. Cloud technology depends on using virtual machines on a single machine to perform parallel programming [43].

There are many applications that need better resource utilization, queries, and large data processing. Hadoop with MapReduce provides these requirements for big data companies such as Google, Yahoo, and Facebook. Hadoop software provides reliability, load balancing, scalability, and parallel computing [44].

B. An Overview of Hadoop MapReduce Framework

In 2005, Doug Cutting and Mike Cafarella established a Hadoop platform for processing big data. Hadoop is an open source implementation of the MapReduce platform from the Apache Software Foundation, and its architecture contains three layers: the Hadoop Distributed File System (HDFS), which provides data storage; MapReduce, which is a data processing layer; and Yarn, a resource management layer [45].

MapReduce is a general programming framework model for parallel computing, which was designed to mitigate the challenges associated with processing big data by splitting the job (submitted work) into a set of sub-jobs (independent tasks)

as shown in Fig. 1, where the programmer just needs to write the map and reduce code and other things are done by the framework [46].

The idea of the computations in the MapReduce platform rests on the key/value pair concept. The computations are expressed as two functions: Map and Reduce, as shown in Fig. 1. The Map function takes a set of inputs represented as key/value pairs, and then produces a set of intermediate results represented as key/value pairs that are stored on the hard disk because it is an intermediate output, not a final one [46].

After that, all intermediate values that are related to the same intermediate key are grouped together and passed to the reduce function, where the reduce function merges related values together based on the key to give the final output that is stored in HDFS. The Reducer consists of two phases, shuffling and sorting, that occur concurrently. In the shuffling phase, different mapper outputs must be shuffled to the reducer [46] as shown in Fig. 1.

There are two secondary steps in Hadoop MapReduce called combine and partition. The partitioning step occurs after the mapping step for redirecting the mapper output to the reducer based on the key to make sure that all values for each key are grouped and sent to the same reducer. The combiner, or mini reducer, is used for reducing the network congestion that occurs due to the large chunks of intermediate output size generated from the mapper and shuffled to the reducer for processing [46].

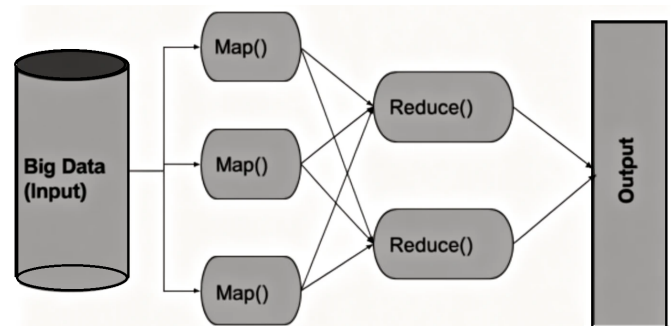


Fig. 1. Hadoop MapReduce work strategy.

C. An Overview of Concurrency Framework

Concurrency is the process of handling multiple tasks at the same time by the system to boost the efficiency and responsiveness of programs. Concurrency can be achieved through multi-threading, multiprocessing, and asynchronous programming. This technique supports better utilization of the system resources when it is carefully designed to prevent problems like race conditions, deadlocks, and data inconsistency. Proper synchronization techniques such as locks or semaphores are often used to ensure correct execution in concurrent programs [47].

Recently, a multi-core processor architecture paradigm has been developed to process threads simultaneously, to achieve higher performance than a single core processor. In the last decades, some research opened the door to designing algorithms for graph problems on multi-core processors [48], [49].

Among various graph algorithms, BFS is one of the common algorithms that is widely used for a range of graph applications such as graph traversing, finding the shortest path between two vertices, and testing a graph for bi-partitions [50].

D. Smart Device Computing

Computing for smart devices is a matter of running tasks on systems with few resources, be they IoT sensors, mobile edge nodes, or embedded processors. Given the constraints on processing power, memory, and energy that are typical of such hardware, one must be judicious in how computational and memory assets are used. This is especially true when it comes to graph processing and the application of BFS algorithms, where a lightweight approach is called for.

The proposed PBFS framework is designed to run on a single machine, which has the advantage of sidestepping the communication overhead one would encounter with distributed multi-machine solutions. While the low-core experiments have yielded some initial evidence as to the viability of PBFS in constrained settings, they fall short of proving its worth for edge-computing platforms or smart devices proper; after all, the work was not done on actual IoT or edge hardware. To make a proper assessment of PBFS performance in the face of real-world limits on capacity, energy use and the like, direct testing on the device itself is necessary.

III. RELATED WORK

BFS is a common graph traversal algorithm used in computer science due to its simplicity and effectiveness for various applications including network routing, social network analysis, and other practical applications. However, by growing in the number of nodes and edges in real world graphs, the traditional sequential BFS becomes a bottleneck. By using multi-core processors or distributed systems, PBFS allows for scalable, high-performance graph processing, which is essential in today's data driven world [51].

In recent years, there exists much research that has been conducted to find an efficient way for parallelizing the BFS algorithm. Some research papers focused on parallelism using shared memory like the work done by Andaloro et al. [52], where the authors proposed a bitmap-based algorithm for small diameter graphs and a graph storage strategy to enhance the cache locality for large-scale graphs. Results show that the proposed approach reduces RAM accesses by 15%. Shi et al. [53] came up with a novel BFS algorithm that handles both synchronization overhead and memory access efficiency called "ABi-BFS". This method decreased the synchronization overhead by using an asynchronous mode for node expanding and task assignment processes. Results support the effectiveness of the proposed method over other state-of-the-art PBFS algorithms.

The authors in [54] proposed a level-synchronous PBFS based on the shared-memory model by integrating optimization techniques to address overhead from implicit barriers and critical sections. For evaluating the proposed model, they used a Graph500 benchmark. Results show a scalable speedup with increased thread count. Also, Fuad et al. [55] introduced a parallel BFS based on shared memory systems. This model adapts dynamically both top-down and bottom-up traversals.

This model outperformed the serial standard BFS with an average speedup of 34.61.

Another category of research that has been conducted on the PBFS subject is implementing parallel models based on distributed-memory systems like MPI-based systems. The research done by Bandyopadhyay et al. [56] is one of the most recent works in this field, where the authors implemented a PBFS model by using the "bag" data structure. For embedding private information within the vertices of 3D mesh images, they utilized the PBFS searching strategy, along with layer synchronization. Results illustrated the efficiency of the proposed model to improve cloud data security frameworks, defense systems and covert communication, enhancing medical image processing, and other sensitive high-security domains.

Bhaskar et al. [57] developed a set of optimizations for PBFS on multi-core distributed systems. This hybrid BFS implementation outperformed other conventional PBFS implementations with an average speedup of 3–8 on the Intel platform, and comparable gains on the AMD system. Parvathy et al. [58] came up with a technique for installing BFS in Spark GraphX for exploiting the benefits of fault tolerance and parallel processing that are intrinsic to the Spark environment. This method can be useful in any application dealing with large graphs like bioinformatics, recommendation systems, and social networks. This method improves the analysis capability of big data in graph processing. Table I presents a summary of the related works that have been discussed in this section.

To the best of our knowledge, few studies have conducted a direct experimental comparison between Hadoop MapReduce and shared-memory concurrency for PBFS. Therefore, in this paper the PBFS algorithm is developed over two frameworks (concurrency-shared memory, and Hadoop/MapReduce).

IV. METHODOLOGY

This section presents the implementations of the PBFS algorithm using the Hadoop/MapReduce and concurrency-based shared-memory frameworks. Section IV-A describes the implementation of PBFS using the Hadoop/MapReduce framework, while Section IV-A1 provides further details on the iterative MapReduce approach and counter-based mechanisms. Section IV-B presents the implementation of PBFS using the concurrency-based shared-memory framework, and Section IV-C discusses the concurrency and synchronization mechanisms employed in this implementation.

A. PBFS Algorithm Using Hadoop/MapReduce Framework

The PBFS starts with the root node that will be processed first, then all its adjacent nodes will be processed in parallel (at the same time) in the second step, and so on until completing the search.

For the Mapper, each line in the input file represents a vertex that contains a key/value pair, where the key is the vertex id and the value is represented by a list of adjacent vertices, path from the source vertex, distance from the source node, and the vertex status that can be unvisited, reached, or visited, as presented in Algorithm 1.

For the Reducer, it takes the mapper output file as an input file and combines all similar vertex lines that have the same key

TABLE I. SUMMARY OF RELATED WORKS

Ref.	Title	Methodology	Results
[52]	Cache-optimized BFS on multi-core CPUs	Proposed two optimizations for BFS graph traversal, one for small diameter graphs and the other for large-scale graphs.	The proposed approach reduces RAM accesses by 15%.
[53]	ABi-BFS: A high-performance parallel breadth-first search on shared-memory systems.	A novel BFS algorithm that handles both synchronization overhead and memory access efficiency called "ABi-BFS".	The proposed method outperformed other state-of-the-art PBFS algorithms with speedups (2–4) on large-scale graphs.
[54]	A parallel breadth-first search using shared memory level-synchronization.	A level-synchronous PBFS based on shared-memory.	Results show a scalable speedup with increased thread count.
[55]	BFSBlitz: A highly parallel graph system for breadth-first search.	A parallel BFS that adapts dynamically both top-down and bottom-up traversals based on shared memory systems.	BFSBlitz outperformed the serial standard BFS with an average speedup of 34.61.
[56]	PBFS through pennant data structure with reducer hyper-object based data hiding for 3D mesh images.	Implemented a PBFS model by using the "bag" data structure.	The proposed model improves cloud data security frameworks, medical image processing, and other sensitive high-security domains.
[57]	Performance-driven optimization of parallel breadth-first search.	Proposed a set of optimizations for PBFS on multicore distributed systems.	The proposed model achieves speedups of 3–8 over other conventional PBFS implementations.
[58]	Implementing Apache Spark GraphX in big data using breadth-first search.	Proposes a technique for installing BFS in Spark GraphX.	Useful in any application dealing with large graphs like bioinformatics, recommendation systems, and social networks.

(vertex id) and then compares their values in order to set the final value for that vertex, where the value consists of vertex information: adjacent vertices, traversed path, distance from source vertex, and the vertex status, as shown in Algorithm 2.

Algorithm 1 PBFS Mapper pseudo-code

Map (id, value)

```
1. foreach Ver(id, value) do
  1.1. status  $\leftarrow$  vertex.Status
  1.2. if vertex.Status == "Reached" then
    // Reached means its color = Gray
  1.3. foreach v  $\in$  lVertex do
    // lVertex is the adjacency list
    // for vertex Ver
    1.3.1. w  $\leftarrow$  new node()
    1.3.2. w.id  $\leftarrow$  v.id
    1.3.3. w.lv  $\leftarrow$  v.lv
    1.3.4. w.Path  $\leftarrow$  ver.path + v.id
    1.3.5. w.distance  $\leftarrow$  ver.d + 1
    1.3.6. w.Status  $\leftarrow$  "Reached"
    1.3.7. w.value  $\leftarrow$  w.lv + w.Path + w.distance + w.Status
    1.3.8. Emit (w.id, w.value)
  1.4. ver.Status  $\leftarrow$  "visited"
  1.5. Emit (ver.id, ver.value)
```

1) *Details of PBFS algorithm with iterative MapReduce and counter:* First of all, in the initialization step, the distances from the source node to all nodes in the graph will be initialized to infinity except for the distance to the source node itself, which is zero.

Nodes are given as inputs to the mappers in the following format: Node ID, adjacent node IDs, Path, Distance from the source, and the node status, which must be one of the ordered list (unvisited, reached, visited).

The mapper is responsible for exploring all adjacent nodes that are connected directly to any reachable node. Therefore, it emits a key/value pair corresponding to each adjacent node; the key is the node ID, and the values consist of the neighbors of the node based on id, updated path, distance of the reachable node +1, and updated status to be reachable. After finishing,

Algorithm 2 PBFS Reducer pseudo-code

Enum Status {*unvisited, reached, visited*}

Reduce(*id, values*)

// *values* is the list of values
generated for vertex *id*

1. Initialize vertex *Ver* with default values:

```
Ver.id  $\leftarrow$  id
Ver.neighbors  $\leftarrow$   $\emptyset$ 
Ver.Path  $\leftarrow$   $\emptyset$ 
Ver.d  $\leftarrow$   $\infty$ 
Ver.Status  $\leftarrow$  unvisited
```

2. *Ver.neighbors* $\leftarrow$ *Ver.get_neighbors(id)*

3. **foreach** *v* $\in$ *values* **do**

```
3.1. if v.d < Ver.d then
  Ver.d  $\leftarrow$  v.d
  Ver.Path  $\leftarrow$  v.Path
```

```
3.2. if v.Status > Ver.Status then
  Ver.Status  $\leftarrow$  v.Status // Status order:
  unvisited < reached < visited
```

4. *Emit (id, (Ver.neighbors, Ver.Path, Ver.d, Ver.Status))*

the parent node status will be updated from "reachable" to "visited" as it has been explored. Now the reducers will get the output of the mapper as a key/value pair. The reducer is used for choosing the shortest distance corresponding to each node and updating its information. This process continues for the next iteration until traversing all the graph.

To find the shortest distance from the source to all nodes, the number of iterations will be equal to the diameter of the graph minus one. For providing more explanation about this algorithm, a simple graph that consists of 6 nodes, which needs three iterations to complete BFS is presented, as shown in Fig. 2.

- If there are two mappers and one reducer, the input file is divided into two blocks, then each block is sent

to a different mapper.

- Each Mapper does the following steps:
 - Check all lines in its input block of data (where each line represents a vertex).
 - For each reachable vertex (its status = reached) the mapper emits a new vertex in its intermediate output file for all reachable node neighbors and updates their distance to be equal to the reachable node distance from source +1 and path; also the status of all neighbors must be set as reachable, while the reachable node status will be set to visited after exploring all its adjacent vertices.
 - Intermediate outputs of all Mappers will be merged and sent to the Reducer.
 - The Reducer takes the result of all Mappers' merged intermediate outputs and combines those data tuples based on the key and modifies the value of the key.

This process is repeatedly done until all reachable nodes are explored in the graph, where the output of the current iteration is used as the input of the next one.

The steps in each iteration have been illustrated using a simple example in Appendix.

B. Parallel Breadth First Search Algorithm Using Concurrency Framework

The implementation process of the PBFS algorithm based on concurrency shared memory is performed through the following steps:

1) *First step*: The input graph is represented using a Node data structure, where each node stores the following information: Node-ID, [Adjacent nodes], [Path], Distance, and Status.

Where:

- Node-ID is a unique identifier for the node.
- Adjacent nodes is an array that stores the Node-IDs of the neighboring nodes, i.e., the nodes directly connected to the intended node.
- Path is an array that stores the Node-IDs along the path from the source node to the intended node.
- Distance represents the number of edges in the recorded path from the source node to the intended node.
- Status indicates the current state of the node as unvisited, reached, or visited.

2) *Second step*: Two shared structures are initialized to support data sharing among parallel threads:

- The first structure, called SharedQueue", is used to manage nodes that have been discovered but still have unprocessed adjacent nodes. It represents the set of nodes requiring further exploration, as shown in Algorithm 3.

- The second structure, called NodeSet", is used to avoid redundant processing of the same node. It is implemented as a concurrent map from node IDs to their corresponding Node objects.

3) *Third step*: The algorithm starts by creating the root node S" from the corresponding source-vertex record in the input graph. The root node is initialized with its complete list of adjacent nodes, such that $S.AdjacentNodes \leftarrow AdjacentNodes$, together with $S.Path = [S]$, $S.Distance = 0$, and $S.Status = visited$ ". The root node is then added to both the NodeSet and the SharedQueue. Thus, the root node is not initialized with an empty adjacency list; its adjacent nodes are obtained directly from the input graph.

4) *Fourth step*: Threads operate on the shared queue until all reachable nodes have been explored. In each iteration:

- A node is dequeued from the SharedQueue.
- The most recent version of the node is retrieved from the NodeSet.
- If the node has no remaining adjacent nodes, it is skipped because all of its neighbors have already been processed.
- If the node has one remaining adjacent node, that neighbor is processed directly.
- If the node has multiple remaining adjacent nodes, the SelectBest() procedure is applied to select the next neighbor for processing.

When multiple adjacent nodes are available, the complete set of neighboring Node objects is passed to the SelectBest() procedure. Each candidate is evaluated according to its degree and number of unvisited neighboring nodes, and the candidate with the highest score is selected. The selected neighbor is processed first, while the remaining neighboring nodes are retained in the current node's adjacency list and are not discarded. After processing the selected neighbor, it is removed from the list of pending adjacent nodes. If unprocessed neighbors remain, the current node is returned to the SharedQueue, allowing the next best remaining neighbor to be selected and processed in a subsequent iteration. Therefore, SelectBest() determines the subsequent priority among the available neighboring nodes rather than eliminating the unselected nodes.

For each selected neighbor, if the neighbor has already been discovered, the existing Node object is retrieved from the NodeSet. Otherwise, a new Node object is created with the corresponding path and distance, marked as reached", and added to both the NodeSet and the SharedQueue. Once all adjacent nodes of the current node have been processed, the node is marked as visited" and its adjacency list becomes empty, indicating that the node has been fully explored.

In the event that a node is adjacent to several others, PBFS will hand over the full complement of neighboring Node objects to the SelectBest() routine. There each candidate is put through its paces; by way of the criterion in Algorithm 4 a score is derived from the node's degree (D) and count of unvisited neighbors (U). The one with the top score is chosen and its Node-ID is sent back to Algorithm 3.

It should be noted that picking the best neighbor is not a matter of discarding the rest. SelectBest() is there to set the processing order for neighbors at the same BFS level. The highest ranked is dealt with first, then the next best, and so on until every eligible neighbor at the current level has been given its turn. In this way the other nodes are held in reserve for later exploration. Once the present level is done, the algorithm moves on to the next. In effect, SelectBest() imposes a priority without forgoing any of the neighboring nodes.

- The main thread oversees the status of all threads.

C. Concurrency and Synchronization Details

The concurrency-driven PBFS was implemented in Java, with multiple worker threads processing graph nodes at the same time. In each experiment, the number of threads matches the number of available processing cores. The workers use a shared queue (SharedQueue) to retrieve nodes for processing and a shared set (NodeSet) to keep track of discovered nodes. To avoid creating duplicate nodes, each worker checks whether a neighboring node already exists in the NodeSet before creating a new Node object. If another thread has already discovered the node, the worker reuses the existing object. Otherwise, it creates a new object, adds it to the NodeSet, and places it in the SharedQueue for later processing. This approach is designed to prevent different workers from creating separate objects for the same graph node.

During the course of a PBFS execution, it is the main thread's job to keep tabs on the status of every worker thread. The workers go about their business in parallel, pulling nodes from the SharedQueue for processing and moving on to explore adjacent ones. From time to time the main thread will make a check to see where the workers are at; this is to prevent an early termination of the run while a node is still being worked on. In fact, the computation will not come to an end until each of the worker threads has seen its task through to completion, signifying the close of the parallel graph exploration. Such coordination is what guarantees that all the parallel processing is done before the PBFS execution is allowed to terminate.

V. EXPERIMENTAL SETUP

This section devoted for presenting the experiments setup including the generated graphs in subsection V-A, testing environment in subsection V-B, and performance evaluation metrics in subsection V-C.

A. Generated Graphs

In this work, a graph generator was implemented to generate graphs of different sizes [250,500,1000 and 2000] with graph density equal to 0.5 for evaluation as shown in Table II.

TABLE II. CHARACTERISTICS OF THE GRAPHS USED IN THE EXPERIMENTAL EVALUATION

Graph	# Nodes	# Edges
G1	250	15562
G2	500	62375
G3	1000	249750
G4	2000	999500

Algorithm 3 Concurrency-Based PBFS

```
ConcurrencyPBFS(ID, AdjacentNodes, Path,
    Distance, Status) // Status ∈ {unvisited,
    reached, visited}
1. SharedQueue ← Queue() // Queue of gray
   nodes
2. NodeSet ← Map<ID, Node> // Maps IDs to
   Nodes
Initialization:
3. Create root node  $S$  from the input record of the
   source vertex:
3.1.  $S.ID \leftarrow ID$ 
3.2.  $S.AdjacentNodes \leftarrow AdjacentNodes$ 
   // Complete adjacency list of the
   source vertex
3.3.  $S.Path \leftarrow [S]$ 
3.4.  $S.Distance \leftarrow 0$ 
3.5.  $S.Status \leftarrow \text{"visited"}$ 
4. Add  $S$  to NodeSet and enqueue  $S$  in SharedQueue
5. while SharedQueue is not empty do
5.1.  $node \leftarrow \text{SharedQueue.dequeue}()$ 
5.2. if  $node.ID \in \text{NodeSet}$  then
5.2.1.  $node \leftarrow \text{NodeSet}[node.ID]$ 
5.3. if  $|node.AdjacentNodes| = 0$  then
5.3.1. continue // No remaining neighbors
   to process
5.4. if  $|node.AdjacentNodes| = 1$  then
5.4.1.  $neighbor \leftarrow node.AdjacentNodes[0]$ 
5.4.2. ProcessNeighbor( $node, neighbor$ )
5.4.3. Remove  $neighbor$  from  $node.AdjacentNodes$ 
else
5.5.1.  $bestID \leftarrow \text{SelectBest}(node.AdjacentNodes)$ 
5.5.2.  $best \leftarrow \text{GetNode}(bestID)$ 
5.5.3. ProcessNeighbor( $node, best$ )
5.5.4. Remove  $best$  from  $node.AdjacentNodes$ 
5.6. if  $|node.AdjacentNodes| > 0$  then
5.6.1.  $\text{NodeSet}[node.ID] \leftarrow node$ 
5.6.2.  $\text{SharedQueue.enqueue}(node)$ 
   // Re-enqueue for further
   exploration
else
5.7.1.  $node.Status \leftarrow \text{"visited"}$ 
5.7.2.  $\text{NodeSet}[node.ID] \leftarrow node$ 
```

B. Testing Environment

The algorithms themselves are written in Java and executed using the Eclipse IDE; JDK 7 used for both compilation and execution of the code.

The experiments were performed on a Windows 10 virtual machine hosted by Microsoft Azure, equipped with 128 GB of RAM and a 4.5 GHz processor. One to five processing

Algorithm 4 SelectBest Neighbor Selection

SelectBest(*AdjacentNodes*)

```
1. best ← null
2. bestScore ← −∞

3. foreach neighborID ∈ AdjacentNodes do
    3.1. degree ← GetDegree(neighborID)
    3.2. unvisited ← CountUnvisitedNeighbors(neighborID)
    3.3. score ←  $\frac{unvisited + 1}{degree + 1}$ 
    3.4. if score > bestScore then
        3.4.1. bestScore ← score
        3.4.2. best ← neighborID

4. return best
```

cores used for the algorithm tests. In order to arrive at a figure for the final analysis, every experiment was conducted ten times and the mean taken. For the Hadoop MapReduce experiments, Hadoop version 2.6.5 was used in a single-node virtual-machine environment.

C. Performance Metrics

This subsection presents a description of all metrics used for the evaluation in this paper [33]:

- **Runtime** represents the time required to complete the process.

$$\text{Runtime} = \text{Termination Time} - \text{Starting Time} \quad (1)$$

- **Speedup** defines as the ratio between sequential and parallel runtimes.

$$\text{Speedup} = \frac{\text{Sequential time}(T_s)}{\text{Parallel time}(T_P)} \quad (2)$$

- **Efficiency** represents the ratio between speedup, and number of processors that are used (P).

$$\text{Efficiency} = \frac{\text{Speedup}}{P} \quad (3)$$

- **Cost** is the result of multiplication the parallel runtime (T_P) with (P).

$$\text{Cost} = T_P \times P \quad (4)$$

VI. EXPERIMENTAL RESULTS AND DISCUSSION

In this paper, a graph generator is implemented, that generates graphs (vertices and edges) of different sizes [250, 500, 1000 and 2000] with medium density equal to 0.5 as inputs for evaluating the proposed PBFS algorithm in two different frameworks [Hadoop/MapReduce, and concurrency] in terms of running time, speedup, efficiency, and cost as evaluation metrics, which are presented in Section V-C. This

section contains two other subsections; one for presenting the results of implementing the PBFS algorithm using the Hadoop/MapReduce framework in Section VI-A, and the other, Section VI-B, devoted to the comparison results of implementing the PBFS algorithm over Hadoop/MapReduce and concurrency frameworks.

A. Results of PBFS Algorithm Using Hadoop MapReduce Framework

In Table III the results of the Runtime (R) measured in seconds on different graph sizes [250, 500, 1000 and 2000] with a medium density = 0.5 by increasing the number of cores from 1 to 5.

TABLE III. RUNTIME IN SECONDS FOR EXECUTING BFS ALGORITHM ON HADOOP FRAMEWORK USING DIFFERENT NUMBERS OF CORES

# Edges	# Nodes	5 cores	4 cores	3 cores	2 cores	1 core
15562	250	7.07511	6.748074	10.95121	12.92243	16.993
62375	500	13.50954	13.43522	14.75074	20.92879	34.679
249750	1000	14.14663	14.14663	15.61068	21.72114	38.012
999500	2000	14.76048	13.96414	17.11435	23.74582	44.001

Speedup results in different graph sizes [250, 500, 1000 and 2000] with medium density = 0.5 are recorded in Table IV by increasing the number of cores from 1 to 5.

TABLE IV. SPEEDUP FOR BFS ALGORITHM ON HADOOP FRAMEWORK

# Edges	# Nodes	5 cores	4 cores	3 cores	2 cores	1 core
15562	250	2.4018	2.5182	1.5517	1.315	1
62375	500	2.567	2.5812	2.351	1.657	1
249750	1000	2.597	2.687	2.435	1.75	1
999500	2000	2.981	3.151	2.571	1.853	1

In Table V the results of Efficiency (E) are presented.

TABLE V. EFFICIENCY FOR BFS ALGORITHM ON HADOOP FRAMEWORK

# Edges	# Nodes	5 cores	4 cores	3 cores	2 cores	1 core
15562	250	0.48036	0.62955	0.517233	0.6575	1
62375	500	0.5134	0.6453	0.783667	0.8285	1
249750	1000	0.5194	0.67175	0.811667	0.875	1
999500	2000	0.5962	0.78775	0.857	0.9265	1

The Cost results over different graph sizes [250, 500, 1000 and 2000] with medium density = 0.5 are presented in Table VI.

TABLE VI. COST IN SECONDS FOR BFS ALGORITHM ON HADOOP FRAMEWORK

# Edges	# Nodes	5 cores	4 cores	3 cores	2 cores	1 core
15562	250	35.376	26.992	32.854	25.845	16.993
62375	500	67.548	53.741	44.252	41.858	34.679
249750	1000	70.733	56.587	46.832	43.442	38.012
999500	2000	73.802	55.857	51.343	47.492	44.001

From Tables III–VI, the following results can be extracted:

- Runtime decreased when the number of cores increased for the same input graph until 4 cores, while with 5 cores the runtime increased for small size graphs only.
- Runtime increased when the number of nodes in the graph increased.

- Speedup increased for the same graph when the number of cores increased, and at the same number of cores when graph size increased the speedup increased.
- Efficiency decreased for the same graph size as the number of cores increased, and at the same number of cores when graph size increased the efficiency also increased.
- Cost increased for the same graph as the number of cores increased, and at the same number of cores when graph size increased cost increased.

B. Comparison Between Results of Implementing the PBFS Algorithm Over Hadoop/MapReduce and Concurrency Frameworks

Table VII presents the runtime results for executing the PBFS algorithm over Hadoop/MapReduce versus those using concurrency frameworks, where the results show that the implementation of PBFS over the Hadoop MapReduce framework outperformed concurrency for large input sizes, as the problem will be distributed among different processors that are working in parallel, which decreases the computation time. It is clear from the results in Tables VIII–X that for large graphs with sizes 1000 and 2000 nodes, the Hadoop framework outperformed the concurrency framework.

TABLE VII. RUNTIME RESULTS IN SECONDS FOR EXECUTING THE PBFS ALGORITHM OVER HADOOP/MAPREDUCE VERSUS THOSE USING CONCURRENCY FRAMEWORKS

# of nodes Framework	1 core		2 cores		4 cores	
	Hadoop	Concur.	Hadoop	Concur.	Hadoop	Concur.
250	16.993	10.187	12.92243	10.152	6.748074	5.134
500	34.679	30.616	20.92879	18.399	13.43522	10.255
1000	38.012	46.413	21.72114	32.915	14.14663	20.584
2000	44.001	52.413	23.74582	40.915	13.96414	24.911

TABLE VIII. SPEEDUP RESULTS FOR EXECUTING THE PBFS ALGORITHM OVER HADOOP/MAPREDUCE VERSUS THOSE USING CONCURRENCY FRAMEWORKS

# of nodes Framework	1 core		2 cores		4 cores	
	Hadoop	Concur.	Hadoop	Concur.	Hadoop	Concur.
250	1	1	1.315	1.003	2.518	1.984
500	1	1	1.657	1.664	2.581	2.985
1000	1	1	1.75	1.41	2.687	2.25
2000	1	1	1.853	1.281	3.151	2.1

TABLE IX. EFFICIENCY RESULTS FOR EXECUTING THE PBFS ALGORITHM OVER HADOOP/MAPREDUCE VERSUS THOSE USING CONCURRENCY FRAMEWORKS

# of nodes Framework	1 core		2 cores		4 cores	
	Hadoop	Concur.	Hadoop	Concur.	Hadoop	Concur.
250	1	1	0.66	0.50	0.63	0.50
500	1	1	0.83	0.83	0.65	0.75
1000	1	1	0.88	0.71	0.67	0.56
2000	1	1	0.93	0.64	0.79	0.53

VII. LIMITATIONS AND FUTURE WORK

The experiments for this study were conducted on synthetic graphs of 0.5 density in order to put the proposed Parallel BFS method through its paces in a controlled setting and gauge its

TABLE X. COST RESULTS IN SECONDS FOR EXECUTING THE PBFS ALGORITHM OVER HADOOP/MAPREDUCE VERSUS THOSE USING CONCURRENCY FRAMEWORKS

# of nodes Framework	1 core		2 cores		4 cores	
	Hadoop	Concur.	Hadoop	Concur.	Hadoop	Concur.
250	16.993	10.187	25.845	20.304	26.992	20.536
500	34.679	30.616	41.858	36.798	53.741	41.020
1000	38.012	46.413	43.442	65.830	56.587	82.340
2000	44.001	52.413	47.492	81.830	55.857	99.640

performance and scalability. One must recognize, however, that such graphs are not an exact stand-in for real-world networks, which tend to be sparse and exhibit quite different degree distributions. These variances can have an impact on Parallel BFS in a number of respects, from the way the workload is apportioned across processing units to memory consumption, the size of the frontier and the volume of edges explored. An uneven distribution of work might result, for instance, with hub nodes of high degree.

Consequently, the performance gains reported in this study are confined to the experimental parameters and should not be taken as representative of every real-world network. That said, the approach is by no means restricted to a 0.5 density and is applicable to other graph structures. Further tests must be run on sparse and authentic data to make a proper evaluation. Future work has us turning our attention to real-world datasets from the web, biological or social networks to see how the PBFS implementations fare against their synthetic counterparts.

There is also the matter of resource constrained edge devices, which the present experiments do not address. While the framework is built for parallel processing on a single machine, the current findings provide only preliminary insights into its potential application to its use in such environments. It would be wrong to read these results as proof of the approach's fitness for smart devices or edge-computing platforms given the absence of any direct testing on that kind of hardware. The realities of processing power, thermal conditions, energy use, and the like could well alter PBFS performance on those systems. For that reason, the proposed approach will be evaluated on actual edge and smart-device platforms in the future. Furthermore, the implementation and evaluation of Depth-First Search (DFS) and other graph traversal algorithms using both frameworks will be considered to provide a broader comparison of their performance and scalability.

VIII. CONCLUSION

This paper presents a detailed account of the PBFS graph exploration algorithm using Hadoop MapReduce and concurrency frameworks. The proposed method was evaluated using graphs of different sizes with a density of 0.5.

The results show that speedup generally increases as the number of cores increases for a given problem size, while efficiency tends to decrease. however, the improvement was not necessarily monotonic due to overheads associated with resource contention, memory access, and other factors that limit performance at higher core counts. When the number of cores remains constant, larger inputs achieve better speedup and efficiency.

For small and medium-sized inputs, the concurrency-based implementation performs better than Hadoop MapReduce. This is mainly because it operates on a single machine and avoids the communication overhead associated with distributed processing. For larger inputs, however, Hadoop MapReduce achieves better performance due to its ability to handle large-scale data. The results obtained on an edge device using one or two cores also show that PBFS can handle small- and medium-sized graphs in resource-constrained environments. Overall, the findings suggest that the PBFS implementations can scale across different computing environments, including IoT and edge-computing platforms.

REFERENCES

- [1] M. Ali, M. Hassan, K. Kifayat, J. Y. Kim, S. Hakak, and M. K. Khan, "Social media content classification and community detection using deep learning and graph analytics," *Technological Forecasting and Social Change*, vol. 188, p. 122252, 2023.
- [2] T. T. Tran and N. Q. Phan, "Analyzing social networks using graph neural networks with advanced hyperparameter optimization techniques," in *International Conference on Smart Objects and Technologies for Social Good*. Springer, 2024, pp. 173–187.
- [3] S. Casale-Brunet, P. Ribeca, P. Doyle, and M. Mattavelli, "Networks of ethereum non-fungible tokens: A graph-based analysis of the ERC-721 ecosystem," in *2021 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2021, pp. 188–195.
- [4] A. Sankar, Y. Liu, J. Yu, and N. Shah, "Graph neural networks for friend ranking in large-scale social platforms," in *Proceedings of the Web Conference 2021*, 2021, pp. 2535–2546.
- [5] S. Abuowaida, E. Elsoud, A. Al-Momani, M. Arabiat, H. Owida, N. ALSHDAIFAT, and H. Y. CHAN, "Proposed enhanced feature extraction for multi-food detection method," *Journal of Theoretical and Applied Information Technology*, vol. 101, no. 24, pp. 8140–8146, 2023.
- [6] A. Y. Alhusenat, H. A. Owida, H. A. Rababah, J. I. Al-Nabulsi, and S. Abuowaida, "A secured multi-stages authentication protocol for IoT devices," *Mathematical Modelling of Engineering Problems*, vol. 10, no. 4, p. 1352, 2023.
- [7] H. A. Owida, B. A.-h. Moh'd, N. Turab, J. I. Al-Nabulsi, and S. Abuowaida, "The evolution and reliability of machine learning techniques for oncology," *Int. J. Online Biomed. Eng.*, vol. 19, no. 8, pp. 110–129, 2023.
- [8] S. D. Pasham, "Scalable graph-based algorithms for real-time analysis of big data in social networks," *The Metascience*, vol. 2, no. 1, pp. 92–129, 2024.
- [9] R. Alazaidah, M. A. Safaqah, H. Alzoubi, H. Omar, M. Y. Al-Safarini, E. A. Elsoud, A. Elrashidi, M. Alluwaici, and S. A. Owaida, "Heart diseases prediction using machine learning techniques: A case study from Jordan," in *2024 25th International Arab Conference on Information Technology (ACIT)*. IEEE, 2024, pp. 1–6.
- [10] M. Iqtait, J. Ababneh, M. Rasmi, A. Abu-Jassar, and S. Abuowaida, "Enhancing facial feature detection: hybrid active shape and active appearance model (hasaam)," *IEEE Access*, vol. 12, pp. 189 590–189 610, 2024.
- [11] S. Rahmani, A. Baghbani, N. Bouguila, and Z. Patterson, "Graph neural networks for intelligent transportation systems: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 8, pp. 8846–8885, 2023.
- [12] A. Arooj, M. S. Farooq, A. Akram, R. Iqbal, A. Sharma, and G. Dhiman, "Big data processing and analysis in internet of vehicles: architecture, taxonomy, and open research challenges," *Archives of Computational Methods in Engineering*, vol. 29, no. 2, pp. 793–829, 2022.
- [13] S. Abuowaida, H. A. Owida, D. M. Alsekait, N. Alshdaifat, D. S. AbdElminaam, and M. Alshinwan, "Ultrasegnet: A hybrid deep learning framework for enhanced breast cancer segmentation and classification on ultrasound images," *Computers, Materials and Continua*, vol. 83, no. 2, pp. 3303–3333, 2025.
- [14] A. Al Sharah, H. A. Owida, F. Alnaimat, and S. Abuowaida, "Application of machine learning in chemical engineering: outlook and perspectives," *Int J Artif Intell*, vol. 13, no. 1, pp. 619–630, 2024.
- [15] M. Baniata, S. Abuowaida, M. Aljaidi, M. Kharabsheh, A. Alsarhan, and A. A. Alsuwailimi, "A multi-modal attention-guided network for Alzheimer's disease classification using deep learning," *Engineering, Technology & Applied Science Research*, vol. 15, no. 5, pp. 27 150–27 158, 2025.
- [16] W. Fan, "Big graphs: challenges and opportunities," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3782–3797, 2022.
- [17] H. A. Owida, A. Arabiat, and S. Abuowaida, "AI enhanced early diagnosis of knee osteoarthritis using x-ray imaging and machine learning approaches," *Discover Applied Sciences*, 2026.
- [18] S. Abuowaida, H. A. Owida, T. Hamadneh, N. Alshdaifat, H. A. Mashagba, M. Abu Alhaija, and A. B. Abd Aziz, "Trt-gla: Tri-representation transformers with global-local attention for high-fidelity multi-modal MRI super-resolution," *Algorithms*, vol. 19, no. 7, p. 603, 2026.
- [19] S. Abuowaida, H. A. Owida, H. A. Mashagba, S. I. Mohammad, A. B. A. Aziz, R. M. Batyha, A. Vasudevan, and M. B. Roslee, "Deep generative model for inverse design of terahertz metamaterial absorbers using conditional variational autoencoders," *Applied Mathematics & Information Sciences*, vol. 20, no. 4, pp. 873–882, 2026.
- [20] S. Abuowaida, H. A. Owida, H. A. Mashagba, A. B. Abd Aziz, M. Alali, and M. Mashagba, "Deep transfer learning with two-stage fine-tuning for robust concrete crack detection," *ES Energy and Environment*, vol. 32, p. 2326, 2026.
- [21] A. Almaghthawi, F. Alharbi, R. M. Batyha, and S. Abuowaida, "Attention-augmented pointnet with hybrid ant colony optimization and bayesian hyperparameter search for robust 3D point cloud classification in augmented reality applications," *International Journal of Advanced Computer Science & Applications*, vol. 17, no. 7, p. 27, 2026.
- [22] S. Abuowaida, H. A. Owida, H. Mashagba, A. Vasudevan, S. Mohammad, M. A. Alhaija, A. B. A. Aziz, and M. Alias, "Deep ANN feature fusion with multi-stage data processing and visualization for enhanced breast cancer detection in ultrasound images," *International Journal of Data and Network Science*, vol. 10, no. 4, p. 1789 – 1806, 2026.
- [23] R. M. Batyha, A. Almaghthawi, F. Alharbi, and S. Abuowaida, "Challenges of AI adoption in the Saudi Arabian healthcare sector: A comprehensive study," *International Journal of Advanced Computer Science & Applications*, vol. 17, no. 7, p. 868, 2026.
- [24] H. A. Owida, S. Ahmad Hasan, A. Arabiat, and S. Abuowaida, "In vitro liposome release profile prediction using explainable machine learning approaches," *PLoS One*, vol. 21, no. 8, p. e0354293, 2026.
- [25] S. Abuowaida, N. Alshdaifat, H. A. Mashagba, A. A. Aziz, A. Alzoubi, M. Roslee, M. Y. Alias, and A. Mahmud, "An enhanced explainable hybrid framework for modeling metaverse-based e-learning acceptance among students with disabilities: Integrating SEM, neural networks, and interpretable machine learning," *Metaverse*, vol. 7, no. 2, 2026.
- [26] Y. Alsarhan, F. A. Asba, A. Alsarhan, and S. Abuowaida, "Advanced multi-layer framework for distinguishing human-written and AI-generated Arabic text: A transformer-based approach with linguistic feature engineering," in *2026 1st International Conference on Emerging Technologies and Engineering Systems (ICETES)*. IEEE, 2026, pp. 181–187.
- [27] B. Berisha, E. Mëziu, and I. Shabani, "Big data analytics in cloud computing: an overview," *Journal of Cloud Computing*, vol. 11, no. 1, p. 24, 2022.
- [28] S. Abuowaida, H. A. Owida, S. I. S. Mohammad, N. Alshdaifat, E. A. Elsoud, R. Alazaidah, A. Vasudevan, and M. T. Alshurideh, "Evidence detection in cloud forensics: Classifying cyber-attacks in IaaS environments using machine learning," *Data and Metadata*, vol. 4, pp. 699–699, 2025.
- [29] S. Alqatan, M. Alshirah, M. B. Baker, H. Khafajeh, and S. Abuowaida, "A conceptual framework for the adoption of cloud computing in higher education institutions," *Data and Metadata*, vol. 4, pp. 431–431, 2025.
- [30] S. Abuowaida, A. Al-Arqan, and K. Al-Mhasneh, "Artificial intelligence and organizational decision-making: Transforming strategic choices in the era of digital intelligence," in *2025 International Conference on*

- Decision Aid Sciences and Applications (DASA)*. IEEE, 2025, pp. 2532–2538.
- [31] K. Hamad, B. Al-Harabsheh, M. Alabedallatif, W. Alsaid, M. A. Hamed, and S. Abuowaida, "Automated lung cancer diagnosis using transfer learning with efficientnetv2-s: A deep learning approach based on iq-othncd dataset," in *2025 1st International Conference on Computational Intelligence Approaches and Applications (ICCIAA)*. IEEE, 2025, pp. 01–06.
- [32] R. AlKahtani, A. Alhabdan, M. Alosami, W. Alshammari, A. A. Abdo, and L. Hamdi, "Comparative analysis between bfs and dfs-shortest path algorithms," in *2025 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*. IEEE, 2025, pp. 1–5.
- [33] S. F. Abuowaida and H. Y. Chan, "Improved deep learning architecture for depth estimation from single image," *Jordanian Journal of Computers and Information Technology*, vol. 6, no. 4, 2020.
- [34] O. Alidmat, H. Abu Owida, U. Kalsom Yusof, A. Almaghthawi, A. Altalidi, R. S. Alkhawaldeh, S. Abuowaida, N. Alshdaifat, and J. Alshaqsi, *IEEE Access*, vol. 13, pp. 7512–7525, 2025.
- [35] H. A. Owida, J. Al-Nabulsi, N. Al Hawamdeh, S. Abuowaida, Z. Salah, and E. A. Elsoud, "Deep learning-based kidney disease classification using resnet50: Enhancing diagnostic accuracy," in *2024 Second Jordanian International Biomedical Engineering Conference (JIBEC)*. IEEE, 2024, pp. 126–129.
- [36] C. Perera *et al.*, "Optimizing performance in parallel and distributed computing systems for large-scale applications," *Journal of Advanced Computing Systems*, vol. 4, no. 9, pp. 35–44, 2024.
- [37] Y. Ji, H. Liu, Y. Hu, and H. H. Huang, "ispan: Parallel identification of strongly connected components with spanning trees," *ACM Transactions on Parallel Computing*, vol. 9, no. 3, pp. 1–27, 2022.
- [38] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *ACM Transactions on Parallel Computing (TOPC)*, vol. 8, no. 1, pp. 1–70, 2021.
- [39] L. Dhulipala, G. E. Blelloch, Y. Gu, and Y. Sun, "Pac-trees: Supporting parallel and compressed purely-functional collections," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 108–121.
- [40] R. Hossen, M. Whaiduzzaman, M. N. Uddin, M. J. Islam, N. Faruqi, A. Barros, M. Sookhak, and M. J. N. Mahi, "Bdps: An efficient spark-based big data processing scheme for cloud fog-iot orchestration," *Information*, vol. 12, no. 12, p. 517, 2021.
- [41] K. Xu, X. Zhou, and B. Qiu, "Plgdc: A parallel density clustering algorithm based on lsh and grid density," in *2025 5th International Symposium on Computer Technology and Information Science (ISCTIS)*. IEEE, 2025, pp. 1189–1193.
- [42] A. Al-Shaikh, H. Khatib, A. Sharieh, and A. Sleit, "Resource utilization in cloud computing as an optimization problem," *Resource*, vol. 7, no. 6, 2016.
- [43] A. K. Sandhu, "Big data with cloud computing: Discussions and challenges," *Big Data Mining and Analytics*, vol. 5, no. 1, pp. 32–40, 2021.
- [44] D.-L. Chiang, S.-K. Wang, Y.-Y. Wang, Y.-N. Lin, T.-Y. Hsieh, C.-Y. Yang, V. R. Shen, and H.-W. Ho, "Modeling and analysis of hadoop mapreduce systems for big data using petri nets," *Applied Artificial Intelligence*, vol. 35, no. 1, pp. 80–104, 2021.
- [45] T. White, *Hadoop: The Definitive Guide*, 3rd ed. Sebastopol, California: O'Reilly Media, 2012.
- [46] S. A. Hannan, "An overview on big data and hadoop," *International Journal of Computer Applications*, vol. 154, no. 10, 2016.
- [47] M. Liu and Q. Wang, "A study on performance optimization of multi-threading and concurrency handling techniques in android applications," in *MATEC Web of Conferences*, vol. 395. EDP Sciences, 2024, p. 01042.
- [48] M. Gupta, L. Bhargava, and S. Indu, "Mapping techniques in multicore processors: current and future trends," *The Journal of Supercomputing*, vol. 77, no. 8, pp. 9308–9363, 2021.
- [49] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.
- [50] M. Kurant, A. Markopoulou, and P. Thiran, "On the bias of bfs (breadth first search)," in *2010 22nd International Teletraffic Congress (LTC 22)*. IEEE, 2010, pp. 1–8.
- [51] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson, "Parallel breadth first search on gpu clusters," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 110–118.
- [52] S. D. Andaloro, T. Pasquali, and F. Vella, "Cache-optimized bfs on multi-core cpus," in *Proceedings of the 1st FastCode Programming Challenge*, 2025, pp. 23–27.
- [53] J. Shi and Y. Xu, "Abi-bfs: a high-performance parallel breadth-first search on shared-memory systems," in *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2023, pp. 2785–2788.
- [54] K. Zheng, "A parallel breadth-first search using shared memory level-synchronization," *Applied and Computational Engineering*, vol. 36, pp. 180–184, 2024.
- [55] S. Fuad and A. Vedadi Gargary, "Bfsblitz: A highly parallel graph system for breadth-first search," in *Proceedings of the 1st FastCode Programming Challenge*, 2025, pp. 33–37.
- [56] S. Bandyopadhyay, S. Mukherjee, S. Mukhopadhyay, and S. Sarkar, "Parallel bfs through pennant data structure with reducer hyper-object based data hiding for 3d mesh images," *Security and Privacy*, vol. 7, no. 5, p. e390, 2024.
- [57] M. Bhaskar and R. Kanakagiri, "Performance-driven optimization of parallel breadth-first search," *arXiv preprint arXiv:2503.00430*, 2025.
- [58] P. R. Parvathy, J. Lenin, and S. Mishra, "Implementing apache spark graphx in big data using breadth-first search," *Innovations in Intelligent Systems and Advanced Engineering*, vol. 1, no. 1, pp. 19–28, 2025.

APPENDIX

The proposed PBFS algorithm using Hadoop MapReduce Framework can be illustrated using a simple example in Fig. 2, which requires three iterations to be traversed by the PBFS algorithm using Hadoop MapReduce.

As shown in Fig. 3, the first iteration started by splitting the input file into two blocks (one block contains the first four lines and the other block contains the last two lines) that had been sent to two different Mappers (in this example the input file consists of 6 lines, where each line represents a separate vertex that is partitioned into key/value pairs), where a key is represented by the vertex-id and a value which consists of four parts including:

- Set of vertex neighbors that are listed between brackets and separated by a comma.
- Set of nodes in the path from the source vertex to the current vertex that are listed between brackets and separated by a comma.
- Distance from the source vertex to the current vertex.
- Vertex status, which is one of the ordered list "unvisited, reached, visited", where:
 - Unvisited status means that a vertex is not reached.
 - Reached means that a vertex is reached, but not explored.
 - Visited means that a vertex is explored.

The first Mapper checks its input file line by line, where at the first line it finds node n0 which is a reachable vertex, so it emits all its neighbors (node n1, node n5, and node n2), modifies their paths and distances to be equal to one, changes status to "reached", at the same time the color changes to "Gray", sets node n0 status "visited", and paints this node with "Black" color as it is expanded as shown in Fig. 3, while nodes n1, n2, n3 are emitted with current key/value pairs. The other mapper continues in the same way on its input block of data.

The Reducer takes, as input, the sorted output that consists of a unique intermediate key, and the list of values associated with it. It performs a reduction operation on this set of values to produce one output value per unique key. Here, it receives one value for keys 0,1,2,3,4 and returns them, while for key 5 it receives two values and emits the value that has less distance with greater status in the output file (which is [0,3 — [0,5] — 1 — reached]) as illustrated in Fig. 4.

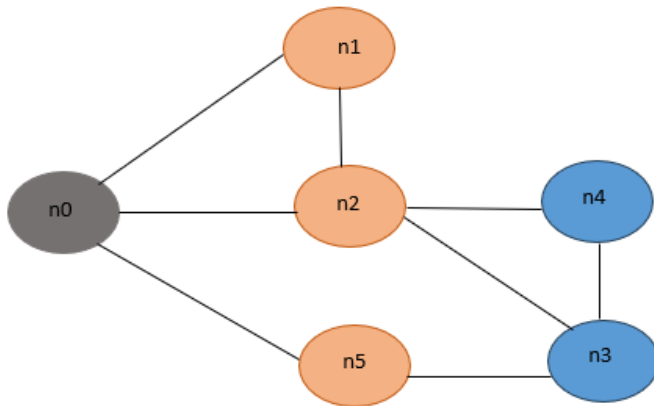


Fig. 2. The input graph / simple example.

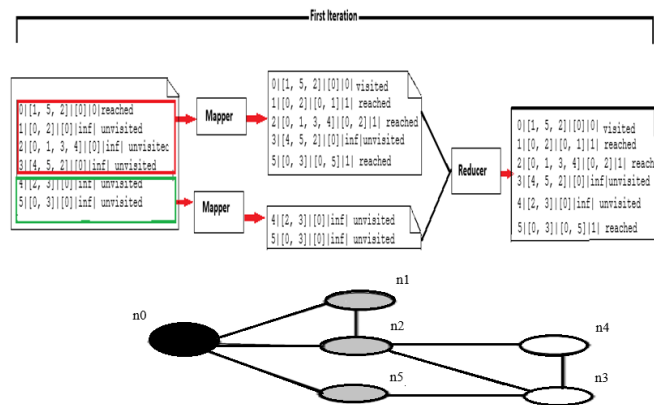


Fig. 3. First iteration / MapReduce.

The input for the second iteration (Iteration 2) is the output of the first iteration, which is split into two blocks (one block contains the first two lines, and the other block contains the last four lines), and is sent to two different Mappers (in this example the input file consists of 6 lines, where each line represents a separate vertex that is partitioned into key/value pairs).

The first Mapper checks its input file line by line, where at the second line it finds node n1 which is a reachable vertex, so it emits all its neighbors (node n0, and node n2) and modifies their path and distance to two. Also, their status changes to “reached”, and color to “Gray”. Later, it sets the status of node n1 to “visited”, and paints it with “Black” color as it has been expanded as shown in Fig. 4, while node n0 is emitted with current key/value pairs. The other mapper works in the same way on its input block of data.

The Reducer merges the set of values that are taken from the sorted output from the previous iteration to produce one output value per unique key. Here, it receives one value for keys 1 and 2 then returns these values, while for key 0 it receives two values and emits the value that has less distance with greater status in the output file (which is [1,5,2 — [0] — 0 — visited]) as illustrated in Fig. 5.

For comparing the performance of the proposed PBFS algorithm over Hadoop MapReduce and concurrency frameworks, a graph generator is implemented to generate graphs (vertices and edges) of different sizes [250, 500, 1000 and 2000] with graph density = 0.5 in terms of running time, speedup, efficiency, and cost.

In the third iteration (Iteration 3), the output from the previous step represents an input for the current one, which is split into two blocks (one block contains the first three lines, and the other block contains the last three lines) that are sent to two different Mappers (in this example the input

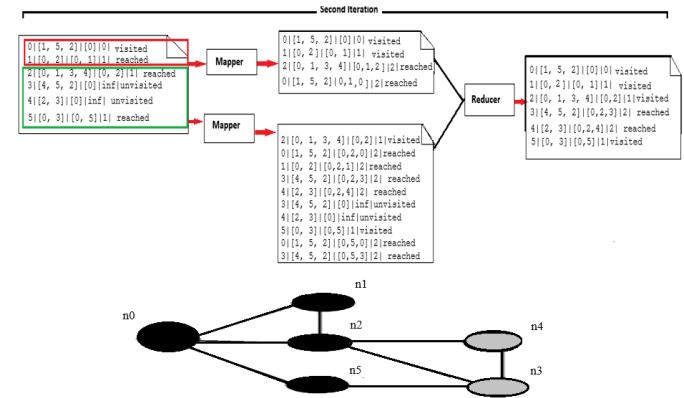


Fig. 4. Second iteration / MapReduce.

file consists of 6 lines, where each line represents a separate vertex that is partitioned into key/value pairs).

The second Mapper checks its input file line by line, where at the second line it finds node n3, which is a reachable vertex, so it emits all its neighbors (node n4, node n5 and node n2), and modifies their distances to be three. In addition, their status becomes equal to “reached” and is painted in “Gray” color. After that, it sets the status of node n3 to “visited” and paints it with “Black” color as it expands as shown in the next figure. Also, node n4 is reachable, so its neighbors (nodes n2 and node n3) will be emitted with distance = 3, and status = “reached”, then node n4 status converts to be “visited”, while node n5 is emitted with current key/value pairs. The other mapper does not find any reachable node, so it emits nodes n0, n1, n2 with their current values.

The Reducer takes the sorted output that consists of a unique intermediate key, and the list of values associated with it, as an input. It performs the reduction operation on this set of values to produce one output value per unique key. In this role, it receives one value for keys 0 and 1 then returns these values, while for key 2 it receives three values and emits the value that has less distance in the output file (which is [0,1,3,4 — [0,2] — 1 — visited]). Also, for key 3 it receives two values and selects as illustrated in Fig. 5.

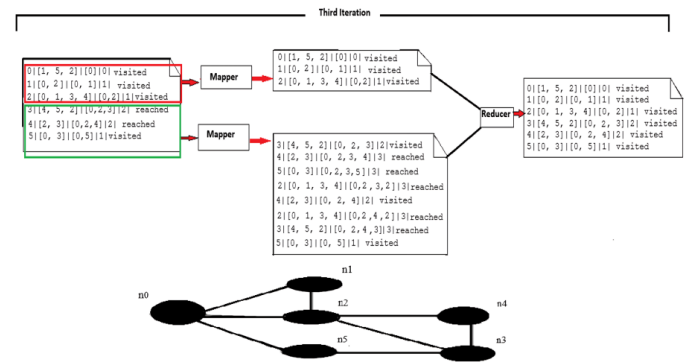


Fig. 5. Third iteration / MapReduce.