Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Parallelized Path-finding in DuckPGQ

Author: Pingan Ren (2758021)

1st supervisor:	Prof. dr. Peter A. Boncz
daily supervisor:	Daniël ten Wolde (Centrum Wiskunde & Informatica)
2nd reader:	Prof.dr. S. Manegold

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 17, 2024

"When people are united, they can move Mount Tai."
by Mao Zedong

Abstract

Graphs, as an important model capable of describing real-world relationships, are getting more and more attention from database researchers and developers. DuckPGQ is an extension developed based on DuckDB with support for SQL/PGQ, which allows DuckDB to construct graphs from relational models and perform operations such as matching, shortest path finding, and so on, on graphs.

The existing path-finding functionality of DuckPGQ was developed based on User-Defined Functions (UDFs).UDFs are black boxes for DuckDB, and their execution relies on DuckDB's morsel-driven parallelism. Shortest path search is a time-consuming operation, and a morsel contains an excessive number of source-to-destination pairs, which makes UDF usually single-threaded with limited performance.

In this thesis, we refactor the existing Multi-Source Breadth-First Search (MS-BFS) algorithm to properly parallelize it. We locate and eliminate race conditions present in the parallel algorithm and explore performance optimization methods including work stealing and direction optimization. Parallel MS-BFS is wrapped by DuckDB's event and task mechanisms to support custom parallelization. The wrapped algorithm is embedded in operators with common interfaces, supporting an integrated process from CSR construction to results output.

We validate the performance and scalability of the new operator through experiments based on the Linked Data Benchmark Council's Social Network Benchmark (LDBC SNB) dataset. The experiments demonstrate that direction optimization has only limited performance improvement. Compared with UDF, the parallel operator has a very significant performance improvement in pathfinding, especially good performance on large graphs. However, the speedup cannot be linear and there are still bottlenecks in the algorithm that have not been eliminated.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

1.1 Context

There is a growing desire to perform more complex analyses on the increasing amounts of data being gathered. A significant value of large data sets is that they capture connections between their entities. It is intuitive to represent and think of these connections as graphs (1). Graph databases, or graph database management systems (DBMS), are optimized for working with graph-related workloads. These systems use graph structures to represent and store data, consisting of nodes (entities) and edges (relationships) (2).

Traditional relational databases struggle to handle the interconnected nature of graph data (3), leading to the rise of specialized graph databases like Neo4j (4), KÙZU (5), and the DuckPGQ extension on DuckDB (6). DuckDB is an in-process SQL OLAP (Online Analytical Processing) database management system known for its efficient columnar storage and vectorized execution engine. This makes it highly suitable for analytical workloads that require low-latency query execution (7). DuckPGQ extends its capabilities by introducing support for graph queries, leveraging its high-performance mechanisms to provide a robust platform for executing complex graph queries. DuckPGQ implements an extension of SQL, namely SQL/Property Graph Queries (SQL/PGQ) (8), a part of the official SQL:2023 standard developed by ISO, to enable users to easily perform various graph-based operations such as traversals, shortest path calculations, and pattern matching.

Path-finding is a fundamental operation in graph databases (9), essential for applications like shortest path queries, network analysis, and recommendation systems. Breadth-First Search (BFS) is a common algorithm used for exploring vertices in an unweighted graph, layer by layer, starting from a source node (10). Multi-source BFS (MS-BFS) (11) extends

this concept by allowing simultaneous traversal from multiple source nodes, which can significantly reduce the time required to explore large graphs. Previously, the path-finding functionality in DuckPGQ was based on MS-BFS implemented as a User-Defined Function (UDF) (12). While this approach was functional, it did not fully exploit the potential of parallel execution.

Parallelism in DuckDB is based on morsel-driven parallelism. Morsel-driven parallelism (13) divides the data into small chunks, or morsels, that are processed independently by multiple workers. In DuckDB, an operator has multiple parallel copies to speed up computation. This method takes advantage of DuckDB's efficient query execution engine but does not fully leverage the potential of parallel execution in graph algorithms. Operator-level parallelism, on the other hand, involves parallelizing the algorithm itself. The acceleration of expensive algorithms can be refined from executing multiple algorithms in parallel to executing the algorithms themselves in parallel.

Several studies have demonstrated the effectiveness of parallel BFS implementations. For instance, Harish and Narayanan (14) introduced a GPU-based parallel BFS algorithm that demonstrated significant speedup compared to CPU-based implementations. Similarly, Hong et al. (15) developed a parallel BFS algorithm for multicore processors, highlighting the potential of parallelism in improving graph traversal performance. These advancements underscore the potential benefits of applying parallelism to graph traversal algorithms in graph databases.

The implications of efficient graph traversal extend beyond academic research into practical applications in various domains. In social network analysis, efficient path-finding algorithms are crucial for identifying influential nodes and discovering communities (16). Similarly, in bioinformatics, analyzing protein-protein interaction networks requires robust graph traversal techniques to uncover meaningful biological insights (17). Efficient path-finding also plays a critical role in transportation networks, where optimizing routes and managing traffic flow depend heavily on the ability to quickly process and analyze large graph datasets (18). The implementation of parallel MS-BFS in DuckPGQ thus holds promise for enhancing these applications by providing faster and more efficient graph query capabilities.

In the context of DuckPGQ, we pursue here the opportunity to enhance path-finding performance by integrating an improved parallel MS-BFS. Our approach leverages operator-level parallelism to distribute the computational load to multiple threads effectively, potentially improving the overall performance of graph traversal operations. By

improve the performance of MS-BFS, DuckPGQ can handle larger graphs more efficiently and provide faster query results for path-finding operations.

The primary objective of this thesis is to enhance the path-finding capabilities of DuckPGQ by implementing parallel MS-BFS and migrating the shortest path-finding implementation from UDF to an operator. The research involves investigating the limitations of morsel-driven parallelism in the context of graph traversal, designing and implementing a parallel MS-BFS algorithm within DuckPGQ, incorporating parallel MS-BFS into the operator, and evaluating the performance of the proposed approach against existing methods using various testing directions. By addressing these objectives, this thesis aims to advance the state of the art in graph database technology and provide a more efficient solution for large-scale graph traversal. This will not only improve the performance of DuckPGQ but also set a precedent for future innovations in graph database systems.

1.2 Research Questions

The following research questions have been defined for this thesis work:

1. How best to refactor path-finding in DuckPQG?
    - How to correctly parallelize the MS-BFS algorithm?
    - What techniques can be used to optimize parallelized MS-BFS to improve speed?
    - How to embed parallelized MS-BFS into the path-finding operator instead of the UDF?
2. What is the performance of parallelized MS-BFS?
3. What are the bottlenecks of parallelized MS-BFS?

1.3 Thesis Structure

The paper is structured as follows. Chapter 2 will introduce the data model for representing graphs, DuckDB, the DuckPGQ extension, and shortest path-finding algorithms that currently exist. Chapter 3 presents related work on shortest path finding including various shortest path finding methods and parallelization approaches. Chapter 4 will describe how we design and implement operator based parallelized shortest path finding, details of the algorithms as well as alternative implementation techniques will be stated. Chapter 5

presents the performance results of the new operators in various dimensions, based on which we will discuss the bottlenecks. Finally, Chapter 6 will look ahead to the work considered but not implemented in this work, as well as optimizations that may be feasible in the future. Chapter 7 summarizes the thesis.

2

Background

This chapter will introduce important concepts and background knowledge that will be used in the thesis to help the reader understand the work.

2.1 Graph

Graphs are a fundamental data structure used to model relationships between objects, consisting of vertices (or nodes) and edges (or arcs) that connect pairs of vertices. In a graph $G = (V, E)$, V represents the set of vertices, and E represents the set of edges. Graphs can be directed or undirected, weighted or unweighted, and vary in complexity from simple linear structures to intricate networks with millions of vertices and edges (19).

Directed graphs have edges with a direction, indicating a one-way relationship between vertices, represented as ordered pairs (u, v), where u is the source vertex and v is the destination vertex. Undirected graphs have edges without a direction, indicating a mutual relationship between vertices, represented as unordered pairs (u, v). Weighted graphs have edges with associated weights or costs, representing the strength or capacity of the connection between vertices, crucial for applications like network routing and shortest path algorithms. The graphs treated in this thesis are directed and unweighted.

Graph representation methods include adjacency matrices, adjacency lists, and edge lists. An adjacency matrix is a $|V| \times |V|$ matrix where the entry at row i and column j is non-zero if there is an edge from vertex i to vertex j. This allows fast access to check if an edge exists but is memory-intensive for sparse graphs. An adjacency list uses an array of lists, where each list at index i contains the vertices adjacent to vertex i, making it more memory-efficient for sparse graphs and widely used in practical applications. An edge list is a simple representation useful for graph algorithms that process edges one at

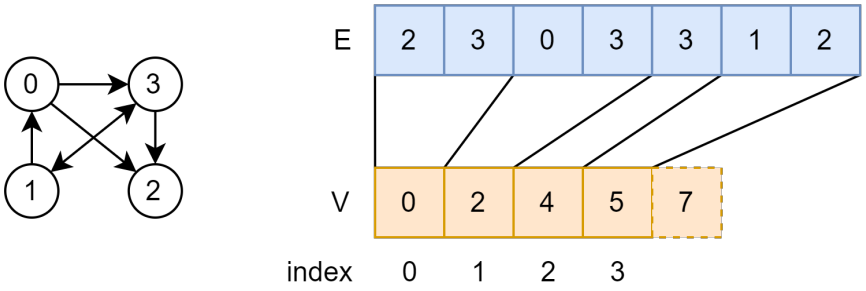

Figure 2.1: A directed unweighted graph and its corresponding CSR

a time, listing all edges in the graph with each edge represented as a pair of vertices. In the next subsection, we will introduce the graph data structure used in this thesis, which is essentially an adjacency list.

2.2 Compressed Sparse Row

Compressed Sparse Row (CSR) is an efficient data structure used to represent sparse matrices, which is particularly useful for handling large and sparse adjacency matrices in graph processing. This representation optimizes storage and allows for efficient execution of graph algorithms such as breadth-first search (BFS) and shortest path computations.

In the CSR format for a graph, we use two main arrays: V to store vertex information and E to store edges. This format is designed to make graph traversal and algorithm execution efficient and straightforward.

The V array contains information about the starting index of each vertex's adjacency list in the E array. Essentially, $V[i]$ indicates the index in E where the edges of vertex i begin. The last element of V points to the end of the E array. The last element is there to help determine the range of the previous element's edges; it is not an existing vertex itself.

The E array holds the actual edges of the graph. Specifically, E contains the destination nodes of the edges. For each vertex, its edges are stored consecutively in E.

For example, consider a simple graph shown in the left part of Figure 2.1. In CSR format, this graph would be represented as the right part.

To explain how this works, let's break down the arrays:

- $V[0] = 0$: The edges for vertex 0 start at index 0 in E. The edges are $[2, 3]$.
- $V[1] = 2$: The edges for vertex 1 start at index 2 in E. The edges are $[0, 3]$.

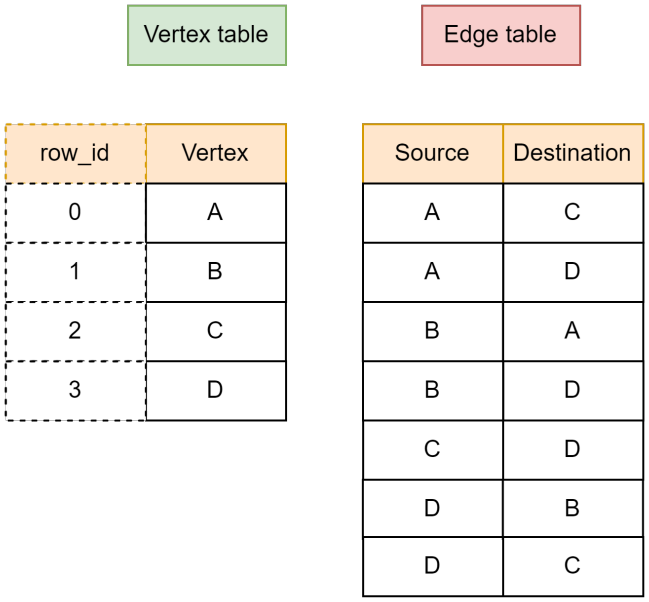

Figure 2.2: Vertex and Edge table for example graph

- $V[2] = 4$: The edges for vertex 2 start at index 4 in E. The edge is [3].
- $V[3] = 5$: The edges for vertex 3 start at index 5 in E. The edges are $[1, 2]$.
- $V[4] = 7$: This marks the end of the E array.

This format makes it easy to access the neighbors of any vertex. For example, to get the neighbors of vertex 1, we look from $E[V[1]]$ to $E[V[2]]$, which gives us $[0, 3]$.

The CSR format offers several advantages. First, by storing only the non-zero elements (edges) and their positions, CSR significantly reduces memory usage compared to a full adjacency matrix. Second, accessing the neighbors of a vertex is efficient, as we can quickly locate the start and end of the vertex's edges using the V array. Third, CSR's structure facilitates parallel processing since different vertices' adjacency lists can be accessed and processed independently.

In DuckDB, we compute on relational tables, which are on-the-fly converted to CSRs whenever we need to perform graph operations. The table corresponding to the CSR mentioned before is shown in Figure 2.2. We get vertex information through the vertex table and edge information through the edge table. In the implementation, each vertex on the CSR is represented by a hidden row row_id of the vertex table, which is unique and dense to facilitate the algorithm to be able to compute on a unified data structure, also the primary key is not relied upon when performing operations on the graphs.

2.3 Parallelization

Parallelization is a technique used to enhance the performance of a program or algorithm by dividing its tasks into smaller sub-tasks that can be executed concurrently. This approach leverages multiple processing units, such as multi-core CPUs or GPUs, to achieve faster computation times. The principles of parallelization involve task decomposition, synchronization, and efficient communication between processing units:

1. **Data Partitioning**: Divide the data into independent chunks that can be processed concurrently. Ensure that the data partitioning minimizes dependencies and maximizes parallel work.

2. **Load Balancing**: Distribute the workload evenly among all processing units to prevent some units from being idle while others are overloaded. Aim for an equal distribution of tasks to maximize resource utilization.

3. **Minimize Communication Overhead**: Reduce the amount of data exchanged between processing units to lower the communication overhead. Efficiently manage the data transfer and synchronization to prevent bottlenecks.

4. **Avoid Race Conditions**: Ensure that concurrent processes do not interfere with each other. Use synchronization mechanisms such as locks, semaphores, or atomic operations to manage shared resources safely.

5. **Locality of Reference**: Optimize for data locality to take advantage of cache memory and reduce access times. Ensure that each processing unit works on data that is close to it in memory to improve performance.

6. **Granularity**: Balance the size of the parallel tasks (granularity). Fine-grained tasks can lead to high overhead due to frequent synchronization, while coarse-grained tasks may not fully utilize parallelism. Find an optimal granularity that maximizes parallel efficiency.

To evaluate the effectiveness of parallelization, several metrics are used to compare the performance between serial and parallel implementations:

1. **Speedup**: Measures the ratio of the execution time of the serial version to the execution time of the parallel version. It is defined as:

$$Speedup = \frac{T_{serial}}{T_{parallel}} \tag{2.1}$$

A speedup greater than 1 indicates a performance improvement due to parallelization.

2. **Efficiency**: Measures the utilization of processing resources in the parallel system. It is defined as:

$$Efficiency = \frac{Speedup}{N} \tag{2.2}$$

where N is the number of parallel processing units. Efficiency values range from 0 to 1, with higher values indicating better utilization.

3. **Scalability**: Evaluates how the performance of a parallel algorithm improves as the number of processing units increases. A scalable algorithm will show near-linear speedup with an increasing number of processors.

4. **Amdahl's Law**: Provides a theoretical limit on the speedup achievable by parallelizing a portion of a program. It states that if P is the fraction of the program that can be parallelized, the maximum speedup S is given by:

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{2.3}$$

This law highlights the diminishing returns of adding more processors if a significant portion of the program remains serial.

2.4 DuckDB

DuckDB (7) is an in-process SQL OLAP (Online Analytical Processing) database management system designed to deliver high-performance analytical query processing on large-scale data. Developed to provide robust analytical capabilities directly within applications, DuckDB is characterized by its lightweight footprint, ease of integration, and efficient execution engine. It leverages a columnar storage layout and vectorized execution model, which allow for high throughput and low-latency query performance. Unlike traditional databases that often require a separate server process, DuckDB operates entirely within the host application's process, making it ideal for embedded analytics, interactive data analysis, and complex data transformations directly within the application environment. This design choice also simplifies deployment and enhances data locality, resulting in faster query performance.

2.4.1 Query Execution Pipeline

The query execution pipeline in DuckDB involves several key components that transform SQL queries into executable instructions through multiple stages. These stages include the Parser, Binder, Logical Planner, Optimizer, Physical Planner, and Execution Engine (20).

The Parser converts SQL queries into tokens such as SQLStatement, QueryNode, TableRef, and ParsedExpression, without resolving catalog details or data types. The Binder resolves these references using the catalog, assigns types, and extracts aggregate and window functions, converting nodes into their bound equivalents. The Logical Planner then creates a logical query tree from these bound nodes. The Optimizer refines this tree with techniques like expression simplification, filter pushdown, join reordering, and common subexpression extraction, generating an optimized logical plan. The Physical Planner translates this logical plan into a physical operator tree, specifying the actual algorithms and data structures for execution. Finally, the Execution Engine executes this physical plan using a push-based vectorized model, where data chunks are processed efficiently through the operator tree to produce the query results.

DuckDB splits the query into operators such as table scans, joins, filters, and aggregations, each responsible for a specific part of the query execution. During the planning stages, the query is broken down into these discrete operations, allowing for detailed optimization and parallel execution. Operators with data dependencies will form a pipeline with a starting point called the source and an endpoint called the sink, where operators from the source to the sink push chunks of data upwards as they execute. A sink operator needs to get all the data before it can be executed, then act as a source for the next pipeline. The shortest path-finding operator we designed contains two sinks: it has to get the complete CSR and all source-destination pairs to find paths before the MS-BFS operator can be executed. An operator may depend on more than one source, so the pipeline can form an operator tree with branches from the bottom up that can be executed in parallel.

On the pipeline, data is processed in parallel with Morsel-Driven Parallelism. Data is processed in vectors, typically $60 \times 2048 = 122880$ tuples in size, which allows DuckDB to make better use of CPU caches and SIMD (Single Instruction, Multiple Data) instructions, processing multiple data points simultaneously. As data flows through the operators, intermediate results are passed along until the final result set is produced and returned to the user.

2.4.2 Push-Based Execution

In pipeline query execution, there are two execution models: push-based and pull-based execution. The pull-based model involves operators pulling data from their child operators as needed. This means each operator requests data from its predecessors only when it is ready to process it. This model is simple and intuitive but can lead to inefficiencies due to frequent function calls and context switching. More importantly, the model is difficult to parallelize.

Conversely, the push-based execution model, which DuckDB employs, involves data being pushed from one operator to the next in a pipeline without explicit requests. In this model, once an operator produces data, it immediately pushes it downstream to the next operator. While the data is generated by operators, the flow is managed externally so that the model is able to handle very complex data flows such as full outer join.

In DuckDB, the push-based data flow model is designed to enhance execution efficiency by separating data flow management from the operators, centrally scheduled by a scheduler. Operations are divided into two primary types: operators, which process data (e.g., filtering and projections), and sources, which emit data (e.g., table scans).

Operators utilize the Execute interface to process input data chunks and produce output results dynamically as data becomes available. For operations requiring complete data before processing, such as certain aggregates, the Finalize interface is used, which waits for all data to be materialized before execution. This distinction allows DuckDB to handle immediate and deferred processing tasks effectively.

Sources use the Sink interface to push data into the execution pipeline. This push-based mechanism is complemented by morsel-driven parallelism, where each processing thread operates on small data chunks (morsels). Each thread maintains a local state, which is later merged into a global state so the Finalize interface can use it. Data emission from sources is managed through the GetData interface, which supplies data to operators' Sink interfaces for processing.

DuckDB parallelizes the push-based pipeline through events. A pipeline can be divided into three events, namely pipeline execution, pipeline finish which needs to wait for all the data on the pipeline to materialize before calling the Finalize interface, and pipeline completion. These three events have a front-back correlation and need to be executed serially. However, events that are not correlated can be executed in parallel. Parallel methods can also be customized within each event to increase parallelism. In our shortest path finding operator, the execution of the algorithm is located in the pipeline finish event.

2.4.3 Morsel-driven Parallelism

Morsel-driven parallelism (13) in DuckDB is a technique for efficiently distributing query execution workloads across multiple CPU cores. The key idea is to divide the workload into small, manageable chunks called "morsels," which are then processed independently. Each morsel typically contains a fixed number of tuples, currently 122880, which allows for fine-grained parallelism and dynamic load balancing if millions or even more tuples are processed.

In DuckDB, when a query is executed, the workload is split into these morsels. Each morsel is assigned to a separate thread, and multiple threads can process different morsels simultaneously. This approach leverages the full computational power of modern multi-core processors, ensuring that each core is utilized effectively.

Technically, the process begins with the pipeline source, which generates the data morsels. These morsels are then queued for execution by worker threads. The scheduling system ensures that each thread receives a new morsel as soon as it completes its current one, maintaining a constant flow of work and minimizing idle time.

Given that DuckDB's vectorized query engine treats 2048 tuples as a data chunk and processes it once, while morsel has a size of 60×2048 tuples, it follows that a thread will be allocated up to 60 data chunks at a time. Normal database operations such as join perform well on data of this magnitude, however pathfinding does not. Just a single tuple of source-to-destination vertices can take tens of seconds to find a path when the graph is large enough. Even with SIMD acceleration with a width of 512 bits, finding 122880 paths will remain time-consuming. So morsel-driven parallelism is not applicable to time-consuming operators like shortest path finding. We need to customize the parallelism within the operator to speed up the computation.

2.4.4 Extension and User-Defined Functions

DuckDB supports extension and scalar User-Defined Function (UDF) (12) to enhance its functionality and adapt to specific application needs. Scalar functions can receive a number of scalar parameters (e.g. string, integer) and produce one scalar result. Extensions in DuckDB are modular components that can be dynamically loaded into the database engine, providing additional capabilities such as new data types, functions, or even storage formats. These extensions can be written in C++ and integrated seamlessly into the DuckDB ecosystem. DuckPGQ is developed in the form of a graph extension that does not require modifications to the internals of DuckDB but still achieves good performance.

User-defined function allows users to define their own functions using SQL or external programming languages like Python or R. This flexibility enables users to perform custom computations directly within the database. For instance, a Python UDF can be created to apply complex data transformations that are not natively supported by DuckDB. These UDFs are integrated into the SQL query execution pipeline, allowing for powerful, customized data processing workflows. UDFs in DuckDB exist mainly as expressions in SQL, which are a part of the pipeline and subject to morsel-driven parallelism. However, the internal logic of UDF is a black box for DuckDB and cannot be used by the optimizer to estimate the workload.

2.5 DuckPGQ

DuckPGQ (6) is an extension of DuckDB designed to support advanced graph querying capabilities. It leverages the efficient, in-memory processing strengths of DuckDB to handle complex graph queries typically found in graph databases. DuckPGQ introduces specialized functions and algorithms optimized for graph traversal and analysis, enabling users to perform operations like shortest path finding and cheapest path finding directly within the DuckDB environment. This integration allows for seamless analytical querying and graph data management within a unified SQL-based system.

2.5.1 Multi-source Breadth-first Search

The underlying algorithm for implementing shortest path finding in DuckPGQ comes from the multi-source breadth-first search (MS-BFS) proposed by Then et al (21). The algorithm is an improvement on the single-source breadth-first search that allows multiple BFS instances to run on a graph simultaneously. The pseudo-code is shown by Algorithm 1.

Concurrency of multiple BFSs is achieved by set operations, otherwise, the algorithm is very similar to the single-source version. The algorithm takes a graph G, a collection of BFS instances $\mathcal{B}$, and a collection of source vertices s. An element (v, B') in the set *seen* indicates that vertex v has been seen on a subset of BFS instances $\mathcal{B}'$. An element $(v, \mathcal{B}')$ in the set *visit* indicates that on the subset $\mathcal{B}'$ of BFS instances, the vertex v is on the frontier of the current iteration. The set $visitNext$ is the frontier of the next iteration. In each iteration, we first find on which BFS instances $\mathcal{B}'_v$ the vertex v on the frontier is active. Then, in the process of exploring the neighbors of v, vertices that have already been visited are excluded through $\mathcal{B}'_v \setminus seen_n$, leaving the unvisited vertices and the corresponding BFS

instances (n, D) to be added to the next iteration's frontier. As the frontier expands over the graph, MS-BFS implements breadth-first searches.

Algorithm 1 Multi-Source Breadth-First Search (MS-BFS)

Input: $G, \mathcal{B}, s$
$seen_{s_i} \leftarrow \{b_i\}$ for all $b_i \in \mathcal{B}$
$visit \leftarrow \bigcup_{b_i \in \mathcal{B}} \{(s_i, \{b_i\})\}$
$visitNext \leftarrow \varnothing$
while $visit \neq \varnothing$ **do**
    for each v in $visit$ **do**
        $\mathcal{B}'_v \leftarrow \varnothing$
        for each $(v', \mathcal{B}')$ in $visit$ where $v' = v$ **do**
            $\mathcal{B}'_v \leftarrow \mathcal{B}'_v \cup \mathcal{B}'$
        end for
        for each n in $neighbours_v$ **do**
            $\mathcal{D} \leftarrow \mathcal{B}'_v \setminus seen_n$
            if $\mathcal{D} \neq \varnothing$ **then**
                $visitNext \leftarrow visitNext \cup \{(n, \mathcal{D})\}$
                $seen_n \leftarrow seen_n \cup \mathcal{D}$
                Do BFS computation on n
            end if
        end for
    end for
    $visit \leftarrow visitNext$
    $visitNext \leftarrow \varnothing$
end while

2.5.2 Shortest Path-finding UDF

DuckPGQ wraps the above MS-BFS in a UDF. This UDF receives two parameters, a (src, dst) pairs vector and the CSR. The (src, dst) vector contains the source-to-destination pairs to be searched, and the CSR generated on the fly before calling the UDF. Due to DuckDB's global vector size setting, the pairs vector has up to 2048 pairs. The number of parallel BFS instances of MS-BFS is set to 512 in this thesis, which is the widest SIMD instruction width and allows 512 pairs to be searched at the same time, this is called a batch. So one UDF call searches at most four batches. Based on the global morsel size, it is possible for more than 60 UDF calls to result in multithreaded parallelization, before that MS-BFS actually runs in single-threaded mode and does not take advantage of multicore

acceleration at all. So custom parallelization that is not based on morsel-driven is highly desirable.

2.6 Single Instruction Multiple Data

Single Instruction Multiple Data (SIMD) is a parallel computing architecture designed to improve performance by processing multiple data points simultaneously using a single instruction. SIMD works by utilizing vector registers, which can hold multiple data elements, and vector instructions that apply the same operation to all elements within these registers in parallel. This architecture leverages data-level parallelism to achieve significant performance improvements, particularly in applications involving large datasets and repetitive calculations.

In SIMD, each vector register can hold multiple data elements, such as integers or floating-point numbers. The size of the vector register and the number of elements it can hold vary depending on the hardware architecture. For instance, in a typical SIMD setup, a single instruction can add corresponding elements of two vector registers and store the result in a third vector register. This simultaneous processing of multiple elements in a single instruction cycle significantly speeds up computations compared to scalar processing, where each operation handles only a single data element.

On x86 platforms, prominent SIMD instruction sets include AVX2 (22) and AVX-512 (23). AVX2 supports 256-bit registers, allowing the processing of eight 32-bit integers or four 64-bit integers simultaneously. It includes features such as gathering instructions for efficient loading of non-contiguous data from memory and fused multiply-add (FMA) operations for combining multiplication and addition in a single instruction. AVX-512 extends these capabilities with 512-bit registers, doubling the data parallelism and introducing masked operations for conditional execution, along with a broader set of instructions for complex arithmetic and data manipulation.

In the ARM ecosystem, SIMD capabilities are provided by the NEON architecture (24), which features 128-bit registers and supports a variety of data types, including 8, 16, 32, and 64-bit integers, as well as single-precision floating-point numbers. NEON is designed to accelerate multimedia processing, digital signal processing, and other computationally intensive tasks common in mobile and embedded applications.

SIMD code can be written in two main ways: explicit SIMD (using intrinsics) and implicit SIMD (using compiler auto-vectorization).

Explicit SIMD involves using intrinsics, which are special functions provided by the compiler that map directly to SIMD instructions. Intrinsics offer fine-grained control over SIMD operations, allowing developers to optimize performance-critical sections of code manually. Here is an example of using AVX-512 intrinsics in C++ to add two arrays of floats:

```
#include <immintrin.h>

for (int i = 0; i < 512; i += 8) {
    __m512i vec_a = _mm512_loadu_epi64(&a[i]);
    __m512i vec_b = _mm512_loadu_epi64(&b[i]);
    __m512i vec_result = _mm512_add_epi64(vec_a, vec_b);
    _mm512_storeu_epi64(&result[i], vec_result);
}
```

In this example, `_mm512_loadu_epi64` loads data into 512-bit registers, `_mm512_add_epi64` performs the addition, and `_mm512_storeu_epi64` stores the result back to memory. This explicit use of AVX-512 intrinsics allows fine-tuning of performance-critical sections of code. The assembly code of these C++ codes compiled by GCC 14.1 is:

```
.L6:
        vmovdqu64       zmm0, ZMMWORD PTR [rsi+rax]
        vpaddq          zmm0, zmm0, ZMMWORD PTR [rcx+rax]
        vmovdqu64       ZMMWORD PTR [rdx+rax], zmm0
        add             rax, 64
        cmp             rax, 4096
        jne             .L6
        xor             eax, eax
        vzeroupper
        ret
```

It uses `vmovdqu64` to load 512-bit chunks (8 long integers) from source arrays into the `zmm0` register. The `vpaddq` instruction then adds two sets of values within the `zmm0` register. The result is stored back into the destination array using `vmovdqu64`.

Implicit SIMD relies on the compiler's ability to automatically vectorize code. This approach requires less effort from the developer but depends on the compiler's ability to recognize opportunities for vectorization. The new version of the GCC compiler has the ability to compile out SIMD code without any explicit instructions, for the following C++ code:

```
for (int i = 0; i < 512; ++i) {
    result[i] = a[i] + b[i];
}
```

The corresponding assembly code is:

```
.L7:
        vmovdqu64       zmm0, ZMMWORD PTR [rdi+rax]
        vpaddq          zmm0, zmm0, ZMMWORD PTR [rsi+rax]
        vmovdqu64       ZMMWORD PTR [rcx+rax], zmm0
        add             rax, 64
        cmp             rax, 4096
        jne             .L7
        vzeroupper
```

In subsequent implementations, we do not use explicit SIMD instructions instead rely on the C++ compiler's auto-vectorization capabilities. Because DuckDB is designed to be cross-platform, each platform has its own unique SIMD instructions and widths. If we used explicit SIMD, we would need to use macros to determine the platform at compile time, or dynamically call the correct code at runtime. Both increase code complexity and reduce maintainability. Although the compiler does not recognize some code that can be accelerated, or outputs SIMD code that is less performant than the manual one, it has very good portability, while at worst it can output serial code to complete the function.

2.7 Cache Latency

Cache latency is a critical factor in CPU performance, affecting how quickly data can be accessed and processed. Modern CPUs use a multi-level cache hierarchy to balance speed and capacity, including Level 1 (L1), Level 2 (L2), and Level 3 (L3) caches, with each level progressively larger and slower.

CPU caches are designed to store frequently accessed data close to the CPU cores, reducing the need to access the slower main memory (RAM). The cache hierarchy typically consists of L1, L2, and L3 caches:

1. **L1 Cache**: Smallest and fastest, usually split into instruction (L1i) and data (L1d) caches, located directly within the CPU core. Typical size is 32 KB to 64 KB per core.

Cache Level	Access Cycles	Access Time (ns)
L1	1 - 3 cycles	0.4167 - 1.25 ns
L2	10 - 20 cycles	4.167 - 8.333 ns
L3	20 - 50 cycles	8.333 - 20.835 ns
RAM	100+ cycles	41.67+ ns

Table 2.1: Access cycles and estimated time for different caches under 2.4Ghz

2. **L2 Cache**: Larger and slower than L1, serves as an intermediary, with a unified cache for both instructions and data. Typical size is 256 KB to 512 KB per core.

3. **L3 Cache**: Largest and slowest, shared among multiple cores, reducing latency for data shared across cores. Typical size is a few megabytes to several tens of megabytes.

Access times for these caches, measured in CPU cycles, are as shown in Table 2.1. We assume the CPU frequency is 2.4GHz, which is the same as our experiment machine.

Lower cache levels (L1 and L2) provide faster access times, while higher levels (L3) and RAM have progressively longer access times. Suppose a core in the CPU needs to process 100GB of data in two executions, the first execution has 50GB of data in L1 hits and the rest of the data needs to be fetched from RAM. In the second execution, all data needs to be fetched from RAM. We take the L1 access cycles to be 2 and the RAM cycles to be 100. then the first execution takes about 2125.17 seconds to read the data and the second execution takes about 4167 seconds. Cache misses can greatly affect the program's performance.

To minimize cache latency and improve performance, several techniques are employed. Prefetching involves predicting which data will be needed next and loading it into the cache before it is requested, significantly reducing cache miss rates and improving access times. Organizing data in memory to align with cache line boundaries can optimize cache usage, as modern CPUs fetch data in blocks called cache lines (typically 64 bytes), ensuring that frequently accessed data is aligned with these boundaries can reduce access times. In multi-core systems, ensuring that all cores have a consistent view of memory is crucial. Cache coherency protocols like MESI (Modified, Exclusive, Shared, Invalid) help maintain consistency across caches, but they increase latency, because each write needs to be communicated (if the line is shared) and dropped. Cache associativity (direct-mapped, set-associative, fully associative) and replacement policies (LRU, FIFO, etc.) affect how data is stored and replaced in the cache. Higher associativity can reduce conflict misses,

while effective replacement policies ensure that the most useful data stays in the cache longer.

It is clear that cache misses significantly affect the time it takes for the core to access data and hence influence performance, as does the protocol for maintaining cache consistency. This requires us to design algorithms that take into account the reduction of random Read/Write and use global synchronization as little as possible.

3

Related Work

3.1 Parallel Path-finding Algorithms

Some high-performance path-finding algorithms and their optimizations are demonstrated below.

3.1.1 Parallel Breadth-First Search

Since the shortest path finding function already available in DuckPGQ is implemented based on multi-source breadth-first search (MS-BFS), in this thesis, we mainly focus on improving the performance of BFS. We first show the optimization methods for normal single-source BFS and then show what methods are available to speed up the computation in the case of multiple sources. We mainly focus on the literature on performance improvement through parallelization.

3.1.1.1 Single-source BFS

The single-source shortest path is the basis of all optimization efforts. Given a graph $G = (V, E)$ and a source vertex $s \in V$, the BFS explores the vertices on the graph that can be accessed by edges, starting from s. The BFS explores the graph in a layer-wise order, and each iteration will look for the neighboring vertices NS of the current frontier FS. Exploration from the frontier can be in any order, and so can be parallelized. The basic serial single-source BFS algorithm is shown in Algorithm 2.

Basic parallelism. The basic parallelization of a single-source BFS is intuitive; the FS is divided into multiple blocks, and each worker computes the block to which it belongs. Note that synchronization is required when exploring neighbors since two workers are likely to visit the same vertex. This simple parallelization is used by many applications,

Algorithm 2 Serial single-source BFS algorithm

```
Input: G(V, E), source vertex s.
Output: d[1..n], where d[v] gives the length of the shortest path from s to v ∈ V.
for all v ∈ V do
    d[v] ← ∞
end for
d[s] ← 0, level ← 1, FS ← φ, NS ← φ
push s → FS
while FS ≠ φ do
    for each u in FS do
        for each neighbor v of u do
            if d[v] = ∞ then
                push v → NS
                d[v] ← level
            end if
        end for
    end for
    FS ← NS, NS ← φ, level ← level + 1
end while
```

e.g. Barnat et al. (25) uses parallel BFS for checking loops in graphs. Their parallel BFS runs on a distributed system, and workers communicate with each other using MPI. The graph is divided into blocks to be distributed to the workers. To avoid synchronization, the update of a vertex only occurs in the worker it belongs to. If a worker updates a vertex that does not belong to it, it notifies the correct worker to perform the update. Iterations are strictly synchronized to avoid conflicts between them.

The parallel BFS proposed by Scarpazza et al. (26) clarifies the details of this parallelization method. They divide each iteration into four parts. The first part initializes the variables that are reused in the iteration. The second part Gather and Dispatch explores the neighbors, and the vertices on the frontier of the next iteration will be divided into different queues temporarily stored in the local memory of the workers according to the attribution of the vertices. Part three All-to-All workers swap queues to globally clarify the attribution of vertices. The fourth part Bitmap workers exclude vertices that have already been accessed. The parts cannot be executed asynchronously with each other to avoid race conditions. Since a worker's local memory is likely to be limited in a distributed system, they also propose an optimized parallel BFS: Gather and Dispatch is split into two

parts, where the workers in Gather fetch a specified number of pieces of vertices at a time instead of dividing the graph equally.

Workload imbalance overhead. Obviously, parallel BFS has a lot of overhead in the computation process. The first overhead comes from workload imbalance. This can easily happen in the real world considering that the number of edges is different for each vertex. For example in social networks famous people can have millions of times the number of followers of ordinary people. It is clearly unfair for two workers to handle the same number of famous and ordinary people separately. Globally this would cause all threads to wait for the slowest thread. Many researchers have proposed ways to make the workload more fair.

One way to mitigate the workload imbalance is to statically assign vertex blocks with the same number of edges to each worker in advance. Yasui et al. (27) proposed to divide the adjacency list equally to get blocks that will have the same workload. Zhang et al. (28) proposed round-robin vertex shuffling, where vertices with large and small degrees are placed evenly in each chunk, which avoids having all large degree nodes or all small degree vertices in one chunk.

Scarpazza et al. (26) proposes that the number of edges contained in the vertices within a block is dynamic whenever a worker extracts the block it wants to process. The worker traverses the vertices so that once the traversed vertices contain the specified number of edges, these vertices are used as blocks to be processed.

Communication overhead. The second most significant overhead comes from synchronization communication during neighbor exploration. On single-machine systems, the communication overhead usually comes from workers waiting for the main memory to lock, when one worker atomically updates the memory, other workers must wait for the update to complete. In distributed systems, the communication overhead comes from nodes propagating modifications to vertices that do not belong to them between nodes to complete synchronization. In a single-machine system, the cores take only a few tens of nanoseconds to update the memory, but due to physical distances and complex communication protocols, the propagation of updates from distributed nodes can take several seconds. This overhead cannot be eliminated but can only be minimized as much as possible.

In a non-distributed scenario, the communication overhead between the processor's cores and memory is small, so it is feasible to make changes made by each worker immediately visible to all other workers. Such updates to contending data must be mutually exclusive.

Typically we use locks to accomplish this operation, but Hassaan et al. (29) propose that compare-and-swap (CAS) can be used to avoid the use of locks.

Communication can be reduced by changing the way the graph is divided. Andy et al. (30) proposes a 2D partitioning of graphs. Their parallel BFS is computed on the adjacency matrix rather than on the adjacency list. The adjacency matrix is divided equally into $R \cdot C$ rows and C columns, and the worker (i, j) owns the block $(j-1) \cdot R + i$. Thus, whenever a worker participates in vertex swapping, it only needs to communicate with other R or C workers instead of the full $R \cdot C$ workers, thus reducing the synchronization overhead. Yasui et al. (27) proposed column-wise partitioning but it is essentially similar to 2D partitioning, which is achieved by horizontally partitioning the adjacency list of each vertex. Their approach is shown to be suitable for NUMA architectures to provide data locality as much as possible. Buluç et al. (31) adapts 2D partitioning to distributed computing. Since a node within a current distributed system usually has multiple cores, they designed intra-node parallelization in addition to inter-node parallelization.

Further, the graph may be modified to accommodate parallel computing. Obviously, the more edges a vertex has, the more likely it is to be accessed simultaneously as a neighbor of multiple vertices in a single iteration, introducing synchronization overhead. So vertices can be pre-ordered by the number of edges such that vertices with the most edges are clustered together (32). This makes synchronization to these hotspot vertices more likely to occur within workers rather than between workers, thus reducing communication costs.

In addition to reducing the number of communications, it is also possible to reduce the amount of data per communication, which is particularly important in distributed computing. Lv et al. (33) proposes that messages can be compressed because workers pass bitsets between each other, which usually have consecutive 0s or 1s, and are well suited for compression. Also, useless parts of the message can be eliminated, e.g. we can know in advance which part of the vertices on the frontier the workers need to get. During the task assignment phase, each worker is usually given a piece of the graph that it owns. Many graphs in the literature are represented as CSR, which involves the transfer of it. Although CSR has been very efficient, there is still room for compression. Ueno et al. (34) proposed that a bitset can be used to indicate whether a vertex has out edges or not, and from the CSR these vertices without edges can be removed. Their experiments show that up to 60% of the space can be saved.

Direction optimization One optimization that is very common in the literature is direction optimization. Direction optimization has two meanings, the first refers to searching towards the center from both the source and destination vertices at the same

time (30), this method is also known as bidirectional BFS. The second refers to searching from unvisited vertices to find their neighbors that are in the frontier, usually, this method is called hybrid BFS (27, 35) shown in Algorithm 3. Both methods can improve the performance because they prevent the algorithm from having too large a frontier in the late phase of execution.

More optimization. Some researchers have argued that algorithmic bottlenecks come not only from explicit workload imbalances and communication overhead etc., but also from bandwidth bottlenecks between the physical cores and memory. In particular, random accesses during neighbor exploration in BFS introduce an extremely high cache miss rate. To address this situation, Attia et al.(37) propose that the data structure can be modified to reduce the amount of data transfer. They customized the vertex list in the CSR so that each element in it is a 64-bit value, and the last bit of the element represents whether the vertex has been visited or not. When the vertex has not been visited, the first 32 of the remaining 63 bits are the start of the edge's offset and the last 31 bits are the end of the edge's offset. When the vertex has been visited, the other 63 bits are used to store the vertex's level. This method compresses four variables (*start offset*, *end offset*, *level*, *visited*) into one.

Additionally unnecessary data reading can be avoided to reduce bandwidth pressure. Zhang et al. (28) proposed that bit operations can be utilized to speed up the checking of whether a vertex has been visited or not. This is done by loading the bitmap block containing the target vertex each time the check is performed, if the block is all 1's, all the vertices in the block must have been visited so the algorithms could skip them, otherwise the offset of the vertex in the block can be used to check whether it has been visited or not. This optimization helps the algorithm to skip a large number of vertices that have already been visited in the intermediate and late iterations.

There are some unconventional optimizations. Träff (38) points out that once a vertex has been visited, all its incoming edges are useless. Imagine that a vertex is visited in level 3, but in level 4 a thousand vertices will revisit it along the edges, generating a lot of invalid computations. By removing the incoming edges of this vertex, it will not be revisited. However, this approach requires the support of an easily changeable data structure such as a pointer adjacency list. John et al. (39) changed the memory model for BFS programs. They used a tree-structured memory model called the Fresh Breeze program execution model (PXM). They modified BFS to be an expansion of the tree, which combined with the properties of the model can eliminate synchronization and achieve lock-free parallel BFS.

Algorithm 3 Hybrid BFS algorithm of Beamer et al. (36)

Input: $G = (V, A^F, A^B)$: unweighted directed graph.
s : source vertex.
Variables: Q^F : *frontier* queue.
Q^N : *neighbor* queue.
visited : vertices already visited.
$state \in \{\text{top-down}, \text{bottom-up}\}$: traversal policy.
Output: $\pi(v)$: predecessor map of BFS tree.

1: $\pi(v) \leftarrow -1, \forall v \in V$
2: $visited \leftarrow \{s\}$
3: $Q^F \leftarrow \{s\}$
4: $Q^N \leftarrow \varnothing$
5: $state \leftarrow$ 'top-down'
6: **while** $Q^F \neq \varnothing$ **do**
7: **if** $state =$ 'top-down' then **then**
8: **for all** $v \in Q^F$ **in parallel do**
9: **for all** $w \in A^F(v)$ **do**
10: **if** $w \notin visited$ **atomic then**
11: $\pi(w) \leftarrow v$
12: $visited \leftarrow visited \cup \{w\}$
13: $Q^N \leftarrow Q^N \cup \{w\}$
14: **end if**
15: **end for**
16: **end for**
17: **else**
18: **for all** $w \in V \setminus visited$ **in parallel do**
19: **for all** $v \in A^B(w)$ **do**
20: **if** $v \in Q^F$ **then**
21: $\pi(w) \leftarrow v$
22: $visited \leftarrow visited \cup \{w\}$
23: $Q^N \leftarrow Q^N \cup \{w\}$
24: **break**
25: **end if**
26: **end for**
27: **end for**
28: **end if**
29: $state \leftarrow \text{traversal\_policy}(Q^F, Q^N, visited, state)$
30: $Q^F \leftarrow Q^N$
31: $Q^N \leftarrow \varnothing$
32: **end while**

Running on GPU. The application of GPUs to parallelize Breadth-First Search (BFS) has been explored extensively in various research studies, each contributing unique techniques and optimizations to improve performance. One notable approach is presented by Harish and Narayanan (14), who utilized CUDA to implement BFS on large graphs. Their method employs a level-synchronous approach, where each level of the BFS tree is processed in parallel. In this implementation, vertices at the current level are processed concurrently by CUDA threads, which update the frontier and mark vertices for the subsequent level. Significant optimization is achieved by using shared memory for fast access to node data and employing coalesced memory accesses to optimize global memory usage.

Hong, Oguntebi, and Olukotun (15) introduced a hybrid method that combines multi-core CPUs and GPUs to enhance the efficiency of BFS. Their algorithm dynamically partitions the graph and assigns tasks to either the CPU or GPU based on workload characteristics and device capabilities. This dynamic partitioning is complemented by a work-stealing mechanism between the CPU and GPU, which helps balance the load, thereby reducing idle times and improving overall performance. The combination of these techniques results in a more efficient utilization of available computational resources.

Zhang et al. (40) introduces several key optimizations for Breadth-First Search (BFS) on GPUs, focusing on data representation, graph partitioning, and communication among GPUs to tackle challenges like irregular memory access, workload imbalance, and high communication overhead. They propose a GPU-friendly Compressed Sparse Row (CSR) structure, which includes bitmap adaptive CSR and warp-aligned adjacency lists to enhance memory coalescing and access efficiency. Additionally, the bidirectional 1D partitioning scheme distributes both vertices and edges more evenly across GPUs, reducing inter-GPU communication and balancing workloads. Furthermore, a Unified Memory (UM)-aware communication method integrates CUDA-based MPI with UM to enable asynchronous data transfers, minimizing the limitations imposed by PCIe bandwidth. Their experiments on NVIDIA Tesla T4 and V100 GPUs demonstrate significant performance improvements, with BFS on four T4 GPUs achieving an average of 132.67 GTEPS and on four V100 GPUs reaching 392.35 GTEPS, outperforming existing CPU-based clusters. These advancements highlight the effectiveness of combining optimized data structures, balanced partitioning, and advanced communication techniques for scalable and efficient BFS on GPU architectures.

3.1.1.2 Multi-source BFS

The main work of this thesis is based on multi-source BFS Algorithm 1, which has been described in the background section. Many parallelization and optimization methods for single-source BFS can be directly applied to multi-source BFS, such as 2D partitioning of graphs and message compression. Related work on parallelized multi-source BFS is relatively rare at the moment, with the main work originating from (11).

They addresses the challenge of parallelizing Multi-Source BFS (MS-BFS) on multi-core systems. They extend the serial MS-BFS algorithm by replacing queues with fixed-sized arrays to eliminate contention points typical in BFS algorithms. The approach to parallelization is essentially the same as for single-source BFS, where synchronization is maintained between iterations, while the iterations are internally divided into several main parts: initialization, neighbor search, and visited vertex removal. And they significantly mitigates overhead through the use of a novel vertex labeling scheme and a low-overhead work-stealing scheduling scheme. The vertex labeling is both cache-friendly and skew-avoiding, ensuring even distribution of work among threads and improving cache hit rates. The work-stealing mechanism dynamically balances the load by allowing idle threads to "steal" tasks from busy threads, thereby maintaining high utilization of all cores. Furthermore, they optimize for Non-Uniform Memory Access (NUMA) architectures by pinning threads to specific CPU cores and strategically placing memory pages to maintain NUMA locality, which minimizes the expensive cross-NUMA node data accesses. Their evaluation on large graphs demonstrates that these techniques enable their parallel BFS algorithms to scale well and outperform existing state-of-the-art algorithms, providing excellent CPU utilization and significantly better performance.

3.1.2 Other Parallel Path-finding algorithms

The BFS is not the only algorithm that implements parallel shortest path finding; many of the classical algorithms in graph theory can be parallelized to improve performance. Many of the current state-of-the-art graph databases are still built on these classical algorithms. Our thesis is mainly aimed at correctly parallelizing MS-BFS, and the other algorithms presented in this section are used for inspiration and reference.

3.1.2.1 Dijkstra's Algorithm

Dijkstra's algorithm is a well-known shortest path algorithm for graphs with non-negative weights. It efficiently finds the shortest path from a single source to all other nodes in the

graph. The algorithm maintains a priority queue to keep track of the next most promising node to explore based on the shortest known distance. Its primary feature is that it handles weighted graphs and ensures the shortest path by repeatedly selecting the node with the smallest tentative distance.

The parallelization of Dijkstra's algorithm focuses on distributing the workload of exploring nodes and updating distances across multiple processors. One approach, as described by Harish and Narayanan (14), involves using a distributed priority queue where each processor maintains its local priority queue and periodically merges results. Another method, proposed by Crauser et al. (41), uses a hierarchical organization of priority queues, with a global queue for coarse-grained tasks and local queues for fine-grained processing. Additionally, the -stepping algorithm, introduced by Meyer and Sanders (42), partitions the edge relaxations into buckets of different ranges, allowing parallel processing within each bucket. These strategies enable the algorithm to efficiently handle large graphs by parallelizing node exploration and distance updates, minimizing inter-processor communication.

3.1.2.2 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm computes shortest paths between all pairs of nodes in a graph, making it suitable for dense graphs where every node is potentially reachable from every other node. The algorithm operates by iteratively refining the shortest paths through intermediate nodes, updating a distance matrix that stores the shortest known distances between every pair of nodes.

Parallelizing the Floyd-Warshall algorithm involves distributing the iterations of the outer loop across multiple processors. Czech (43) suggested that each processor handle a portion of the rows or columns of the distance matrix. In a shared memory model, OpenMP can be used to parallelize the loops directly, with synchronization mechanisms to handle updates to the distance matrix. In a distributed memory model, the distance matrix is partitioned across processors, and message passing (MPI) is used to synchronize updates, ensuring consistency. Tiling techniques can further enhance performance by improving cache utilization and reducing communication overhead (44). These approaches leverage the algorithm's inherent parallelism to efficiently compute shortest paths in large, dense graphs.

3.1.2.3 Bellman-Ford Algorithm

The Bellman-Ford algorithm computes shortest paths from a single source node to all other nodes and can handle graphs with negative weights. It operates by iteratively relaxing the edges of the graph, updating the shortest known distances until no further improvements can be made.

Parallelizing the Bellman-Ford algorithm involves distributing the relaxation steps across multiple processors. Bader et al. (45) proposed that each processor handle a subset of edges during each iteration. Another significant contribution is the work by S. Sharma et al. (46), which presents an efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures, leveraging the massive parallelism available in modern GPUs. Meanwhile ideas from -stepping algorithm can still be used.

3.1.2.4 A* Algorithm

The A* algorithm is a heuristic-based path-finding algorithm commonly used in AI and game development. It combines the advantages of Dijkstra's algorithm and best-first search, using heuristics to guide the search towards the goal node more efficiently. The algorithm maintains a priority queue to explore the most promising nodes first, making it suitable for both weighted and unweighted graphs.

Parallelizing the A* algorithm involves distributing the search space across multiple processors. Kishimoto et al. (47) proposed a method where each processor explores different regions of the graph. Synchronization mechanisms ensure consistency, especially when processors find paths that need to be shared or merged. Parallel fringe search and dynamic load balancing techniques are used to improve efficiency, as discussed by Sturtevant and Buro (48). These approaches enable the algorithm to handle large search spaces by parallelizing node exploration and heuristic evaluation, reducing search time and improving scalability.

3.1.2.5 Johnson's Algorithm

Johnson's algorithm is used for finding shortest paths between all pairs of nodes in sparse graphs. It combines the Bellman-Ford algorithm to reweight the graph and then uses Dijkstra's algorithm for each node. This approach allows it to handle negative weights while maintaining efficiency in sparse graphs.

Pogorilyy et al. (49) developed a parallel implementation of Johnson's algorithm using General-Purpose GPU (GPGPU) technology. The Bellman-Ford component is parallelized

by distributing edge relaxations across multiple GPU threads. Each thread processes a subset of edges and updates distances in parallel. For the Dijkstra component, the graph is divided into subgraphs, each handled by different GPU threads. CUDA's efficient thread management is leveraged to minimize inter-thread communication. Atomic operations are used to update shared distances, ensuring consistency across threads. The implementation uses CUDA kernel functions to execute both Bellman-Ford and Dijkstra's components concurrently. This approach achieves significant speedups in computing shortest paths for large, sparse graphs by utilizing the parallel processing power of GPUs.

3.1.2.6 Yen's Algorithm

Yen's algorithm is used for finding the K shortest paths between a pair of nodes in a graph. It first finds the shortest path and then iteratively finds the next K-1 shortest paths by modifying the graph and applying Dijkstra's algorithm.

Singh and Singh (50) proposed a parallel implementation of Yen's algorithm using CUDA. Each CUDA thread handles a portion of the path modifications and subsequent shortest path computations. The graph is represented using an adjacency list stored in multiple arrays to facilitate efficient memory access. The parallel implementation uses CUDA's thread hierarchy to manage the computation: blocks of threads are assigned to different segments of the graph, and individual threads within each block handle specific path modifications. Shared memory is used to store intermediate results, reducing global memory accesses. The implementation achieves a 6x speedup over the serial version by concurrently computing multiple shortest paths and efficiently managing memory and computation resources on the GPU.

3.2 Query Parallelism

Query Parallelism is a critical aspect of modern database management systems (DBMSs) and big data processing aimed at optimizing performance and scalability. Various approaches have been developed to exploit parallel processing capabilities, enabling databases to handle increasingly complex and voluminous workloads efficiently. By reviewing existing acceleration methods, we can get inspired to accelerate our pathfinding operator.

3.2.1 Intra-Query Parallelism

Intra-query parallelism focuses on parallelizing the execution of a single query by dividing it into smaller tasks that can be executed concurrently. This approach can be further divided into pipeline parallelism, Partitioning parallelism, and dataflow parallelism.

3.2.1.1 Pipeline Parallelism

Pipeline parallelism exploits the natural stages of query processing, such as scanning, filtering, and aggregation, by executing these stages concurrently across different processing units. A notable example of this is the Volcano model, which introduces a pull-based execution model allowing operators to work in a pipelined fashion (51).

Modern implementations have refined this approach to include techniques like vectorized execution, which processes batches of tuples to minimize overhead and improve cache performance (52). The Apache Arrow project (53) exemplifies such advancements, offering a columnar in-memory format that enhances data locality and speeds up analytic workloads.

3.2.1.2 Partitioning Parallelism

Partitioning parallelism, also known as data partitioning or horizontal partitioning, divides the dataset into smaller, independent partitions that can be processed in parallel. This technique is widely used in shared-nothing architectures (54), where each node operates on its local partition of data. Systems like Google BigQuery (55) and Amazon Redshift (56) utilize partitioned parallelism to scale out their processing capabilities across distributed nodes.

Research has shown that effective data partitioning strategies, such as range partitioning, hash partitioning, and round-robin partitioning, are critical for load balancing and minimizing data movement across nodes (57). Adaptive partitioning techniques, which dynamically adjust partitions based on workload characteristics, further enhance the efficiency of partitioned parallelism (58).

3.2.1.3 Dataflow Parallelism

Dataflow parallelism extends the concept of pipeline parallelism by modeling query execution as a directed acyclic graph (DAG) of operators, where edges represent data flow between operators. This model allows for fine-grained parallelism and optimizations, such as operator fusion and scheduling (59).

Frameworks like Apache Flink (60) employ dataflow parallelism to execute complex analytical queries on large-scale data. These systems provide fault tolerance and elasticity, enabling them to handle varying workloads effectively. Recent research has focused on optimizing dataflow execution with techniques like dynamic task rebalancing and speculative execution to further improve performance (61).

3.2.2 Inter-Query Parallelism

Inter-query parallelism focuses on executing multiple queries concurrently by leveraging the multi-threading and multi-processing capabilities of modern hardware. This approach is beneficial for environments with high query throughput, such as transactional systems and real-time analytics platforms.

3.2.2.1 Multi-Threaded Execution

Multi-threaded execution involves spawning multiple threads within a single process to handle different queries or subqueries concurrently. This approach takes advantage of multi-core processors to improve query response times and system throughput. Database systems like PostgreSQL (62) employ multi-threaded execution for parallel query processing.

Recent advancements in multi-threaded execution include techniques like thread-level speculation, which allows speculative execution of queries to mitigate the impact of dependencies and contention (63). Additionally, fine-grained locking mechanisms and lock-free data structures have been developed to reduce synchronization overhead and improve scalability (64). Lock-free data structures, such as concurrent queues and hash tables, allow multiple threads to operate on shared data without using locks, thus avoiding contention and potential deadlocks.

3.2.2.2 Multi-Processing and Distributed Execution

Multi-processing and distributed execution extend inter-query parallelism to a distributed environment, where queries are executed across multiple processes or nodes. This approach is prevalent in cloud-native databases and distributed systems like Google Spanner (65), which provide global consistency and fault tolerance through distributed consensus algorithms.

Techniques such as distributed query optimization and federated query processing have been proposed to efficiently execute queries across heterogeneous data sources and

geographically dispersed nodes (66). Distributed query optimization involves generating an execution plan that minimizes data transfer and balances the load across nodes. Federated query processing, on the other hand, allows queries to be executed across multiple databases as if they were a single database, using techniques like data localization and query rewriting to optimize performance.

3.2.3 Hybrid Approaches

Hybrid approaches combine intra-query and inter-query parallelism to leverage the strengths of both techniques. These approaches aim to maximize resource utilization and improve overall system performance by dynamically balancing the execution of multiple queries and subqueries.

3.2.3.1 Adaptive Query Execution

Adaptive query execution (AQE) is a hybrid approach that dynamically adjusts query execution plans based on runtime statistics and feedback (67). This technique allows for better handling of data skew, resource contention, and changing workloads. Systems like Apache Spark (68) have integrated AQE to optimize execution plans and improve performance.

Research in AQE has focused on techniques like re-optimization, which revisits and adjusts execution plans during query execution, and adaptive partitioning, which adjusts data partitioning strategies based on runtime information (69). Re-optimization allows the system to change the execution strategy mid-query, based on observed data distributions and system load. Adaptive partitioning dynamically changes the size and number of partitions to ensure a balanced load and efficient resource use.

3.2.3.2 Federated Query Processing

Federated query processing involves executing queries across multiple, potentially heterogeneous data sources. This approach combines intra-query parallelism for local execution with inter-query parallelism for coordinating across data sources. Systems like Presto (70) and Apache Drill (71) exemplify federated query processing, providing a unified query interface for disparate data sources.

Recent advancements in federated query processing include techniques like data virtualization (72), which abstracts underlying data sources, and query optimizations (73), which optimize query execution across heterogeneous systems. Data virtualization allows

users to query different data sources without needing to know the specifics of each source, while query optimizations aim to minimize data transfer and processing time by pushing computations closer to the data source.

3.3 DBMSs Supporting Path-finding

Graph path-finding is an essential feature in database management systems (DBMSs) for applications such as social networks, logistics, and recommendation systems. DBMSs supporting graph path-finding can be categorized into native graph systems and multi-model systems. Native graph systems refer to systems that are graph-oriented from the beginning of their design, and they usually have the most sophisticated graph operations. Multi-model systems refer to the fact that as graphs have become more important, some traditional relational databases have supported graph operations through plug-ins or extensions, etc., and they are easy to integrate with data sources already in use. Below is a summary of selected most popular DBMSs from the DB-Engines Ranking (74).

3.3.1 Native Graph DBMS

3.3.1.1 Neo4j

Neo4j (75) is a leading native graph database known for its efficient graph processing and robust querying capabilities. It employs the Cypher (76) query language, which is purpose-built for handling graph data, allowing users to express complex graph queries intuitively.

Neo4j includes a comprehensive library of graph algorithms and uses index-free adjacency, which allows for direct linking of nodes with their neighbors without intermediary structures. This significantly improves the performance of graph traversals and makes Neo4j suitable for applications requiring intricate relationship handling and real-time updates.

For path-finding, Neo4j supports a range of functions including shortest path, all shortest paths. The built-in algorithms provided by A to realize these functions are shown in Table 3.1, which lists the important features of these algorithms.

3.3.1.2 Memgraph

Memgraph (77) is an in-memory native graph database designed for real-time analytics and transactional workloads. Its architecture is optimized for low-latency processing, making

Algorithm	Source	Destination	Functionality	Directed	Undirected	Weighted	Parallel
Delta-Stepping	Single	Multi	Shortest path	√	√	√	√
Dijkstra	Single	Multi	Shortest path	√	√	√	×
A*	Single	Single	Shortest path	√	√	√	×
Yen's	Single	Single	K-Shortest path	√	√	√	√
BFS	Single	Multi	Traversal	√	√	×	√
DFS	Single	Multi	Traversal	√	√	×	×
Random walk	Multi	Multi	Random path	√	√	√	√
Bellman-Ford	Single	Multi	Shortest path	√	√	√(negative)	√
Prim	Single	Multi	Minimum spanning tree	×	√	√(negative)	×

Table 3.1: Path-finding algorithms for Neo4j

it suitable for use cases such as fraud detection, recommendation systems, and network analysis.

Memgraph's in-memory design ensures rapid access to graph data and supports dynamic schema evolution, allowing modifications to the graph structure without downtime. It also integrates with various data streaming platforms, enabling real-time data ingestion and processing.

For path-finding, Memgraph uses the Cypher query language to express complex graph queries. It supports four built-in path algorithms that can be directly supported by queries. For unweighted graphs, DFS is used to find all shortest paths from a given vertex to reachable vertices, and BFS returns only one shortest path for each pair of vertices. For weighted graphs, Dijkstra's algorithm is used to return a path with the smallest weight sum, and an alternative syntax can be used to implement all shortest paths. Memgraph can support more advanced algorithms, which exist as libraries. Memgraph leverages its in-memory processing to enhance the performance of these algorithms, ensuring low-latency responses for real-time analytics.

3.3.1.3 NebulaGraph

NebulaGraph (78) is a high-performance, scalable graph database designed for handling massive graphs with billions of nodes and edges. It supports the openCypher (79) query language, allowing users to execute complex graph queries efficiently.

NebulaGraph provides a distributed architecture that ensures data redundancy and fault tolerance, making it ideal for applications in social networks, recommendation systems, and network management. The system is optimized for large-scale graph data, ensuring efficient data handling and processing.

Algorithm	Source	Destination	Functionality	Directed	Undirected	Weighted	Parallel
A*	Single	Single	Shortest path	√	√	√	×
BFS	Single	Multi	Traversal	√	√	×	×
Edmond-Karp	Single	Single	Maximum flow	√	√	√	×
Prim	Single	Multi	Minimum spanning tree	×	√	√	√
Bellman-Ford	Single	Multi	Any shortest path	√	√	√(negative)	√

Table 3.2: Path-finding algorithms for TigerGraph

NebulaGraph natively supports pathfinding via the `FIND PATH` statement. This statement supports directed or undirected graphs, finds paths between two vertices, obtains single or all shortest paths, or returns all possible paths. The statement supports returning paths with loops, but the paths are of type **TRAIL**, which only supports duplication of vertices. Internally there are various algorithms that support the functional implementation, including BFS and some highly customizable algorithms. The statement runs in single-threaded mode. NebulaGraph has algorithm extension libraries (80) to support more advanced algorithms. The database employs multi-threaded execution and efficient indexing to parallelize graph queries, significantly enhancing the performance of path-finding operations. These acceleration techniques ensure quick query responses even for large-scale graph data.

3.3.1.4 TigerGraph

TigerGraph (81) is a high-performance, distributed graph database known for its ability to handle complex graph queries and real-time analytics. It supports the GSQL (82) query language, designed specifically for graph data, enabling users to execute intricate queries with ease.

TigerGraph excels in performing high-speed graph analytics, making it suitable for applications in fraud detection, recommendation systems, and social network analysis. The database is optimized for parallel processing of graph queries, ensuring efficient data handling and query execution.

TigerGraph supports complex graph operations through TigerGraph Graph Data Science Library (GDS), where path-finding algorithms can be specified directly in the query. The supported algorithms are shown in Table 3.2.

3.3.1.5 Dgraph

Dgraph (83) is an open-source, distributed graph database designed for high-performance graph processing and real-time queries. It supports the GraphQL (84) query language,

Algorithm	Source	Destination	Functionality	Directed	Undirected	Weighted	Parallel
bdBFS	Single	Single	All shortest path	√	√	×	√
DFS	Single	Single	All simple path	√	√	×	×
Dijkstra	Single	Single	Shortest path	√	√	√	√
bdDijkstra	Single	Single	Shortest path	√	√	√	√
Bellman-Ford	Single	Multi	Shortest path	√	√	√	×

Table 3.3: Path-finding algorithms for Oracle Graph

enabling users to perform complex graph operations using a flexible and expressive syntax.

Dgraph's native graph storage engine is optimized for fast data retrieval and efficient graph traversals. This ensures strong consistency and fault tolerance, making it suitable for applications requiring reliable graph processing.

Dgraph provides a built-in k-shortest path query statement. This query finds the shortest path for a pair of source and destination points at a time. It supports graphs with non-negative weights and the result does not contain circular paths. Internally, Dijkstra's algorithm is used for implementation.

3.3.2 Multi-model DBMS

3.3.2.1 Oracle

Oracle (85) Database is a leading relational database management system that also offers extensive support for graph processing through its Property Graph feature. Earlier Oracle used the PGQL (86) query language to support operations on graphs, and starting in 2023, Oracle began supporting SQL/PGQ (8), which is compliant with the SQL 2023 standard. in addition, Oracle REST Data Services (ORDS) recently supported the GraphQL language, which extends Oracle's ability to work with graphs even more.

Oracle Graph supports a rich set of graph algorithms, as shown in Table 3.3. Among them, the implementation of BFS and Dijkstra applies the idea of bi-directional search, i.e., the search starts from the source and the target at the same time and the algorithms will meet in the middle. This approach is able to reduce the amount of computation.

3.3.2.2 Microsoft SQL Server

Microsoft SQL Server (87) is a robust relational database system that extends its capabilities with graph processing features. It incorporates graph extensions within SQL to support graph data modeling and querying.

SQL Server integrates graph data into relational structures using graph tables, allowing seamless data handling. This integration is complemented by advanced security features, ensuring secure graph data processing. Starting with SQL Server 2017, Transact-SQL extensions are applied to support graph operations. Users are allowed to create node tables and edge tables to represent graphs, both of which are compatible with normal relational tables.

Shortest path queries in SQL Server are used with the statement `SHORTEST_PATH` in conjunction with `MATCH`. The query supports one-to-one, one-to-many, and many-to-many, and can only return any one shortest path between two vertices. The query supports only unweighted graphs. Since SQL Server is closed source, there is no clear source indicating what the implemented algorithm is. Based on the fact that the query does not support characteristics such as weights, it is reasonable to assume that the actual algorithm used is BFS. Listing 1 shows an example to perform shortest path finding.

Listing 1 Find shortest path between two people on SQL Server

```
SELECT PersonName, Friends
FROM (
    SELECT
        Person1.name AS PersonName,
        STRING_AGG(Person2.name, '->') WITHIN GROUP (GRAPH PATH) AS Friends,
        LAST_VALUE(Person2.name) WITHIN GROUP (GRAPH PATH) AS LastNode
    FROM
        Person AS Person1,
        friendOf FOR PATH AS fo,
        Person FOR PATH  AS Person2
    WHERE MATCH(SHORTEST_PATH(Person1(-(fo)->Person2)+))
    AND Person1.name = 'Jacob'
) AS Q
WHERE Q.LastNode = 'Alice'
```

3.3.2.3 PostgreSQL

PostgreSQL (88) is an advanced open-source relational database known for its extensibility and SQL compliance. It supports graph processing through extensions like pgRouting (89) and AgensGraph (90), enabling rich graph data functionalities.

PostgreSQL is highly extensible, allowing advanced customization and integration of various graph processing capabilities. The database supports a wide range of data types and indexing techniques, providing robust support for diverse applications.

For path-finding, PostgreSQL supports functions like shortest path and all pairs shortest path through pgRouting. It implements weighted shortest distance finding mainly by A* algorithm and Dijkstra's algorithm while bidirectional versions of these two algorithms are also implemented for speedup. BFS and DFS and their derived algorithms are used to construct minimum spanning trees. And K shortest path is implemented by Yen's algorithm. None of these algorithms support parallelization and are only capable of speeding up a batch of searches by running multiple queries simultaneously.

3.3.2.4 MariaDB

MariaDB (91) is a popular open-source relational database that extends its functionality with graph processing through the OQGRAPH (92) storage engines. These engines enable users to model and query graph data within the relational framework.

MariaDB's OQGRAPH storage engine integrates seamlessly with SQL, allowing users to perform graph operations within a relational context. This provides a straightforward approach to incorporating graph processing into existing SQL-based applications.

For path-finding, MariaDB supports functions like shortest path through the OQGRAPH storage engine. The graph still exists as a relational table at the physical layer, but there is no need to explicitly convert the relational table to a graph structure each time a graph query is executed. The BFS and Dijkstras algorithms are used to implement the shortest path query, and the desired algorithm can be specified in the query. OQGRAPH is not very robust, making it suitable for nightly work on pre-existing data.

4

Design & Implementation

The goal of this project is to refactor the shortest path finding functionality already in DuckPGQ, moving from the UDF implementation to the operator implementation, so that custom parallelization can be achieved in operators using DuckDB's Event and Task interfaces, rather than morsel-driven parallelization handled automatically by DuckDB. As we mentioned in Chapter 2, the shortest path-finding based on morsel-driven is not able to fully utilize the resources, and we expect that parallelizing the Multi-source breadth-first search (MS-BFS) algorithm within the operator itself leads to better performance. This chapter details how we built the parallelized shortest path finding operator, using the refactoring and parallelizing of the original algorithm as an entry point.

First, the most basic algorithmic refactoring will be outlined, with some alternative technical solutions at the implementation level described in detail. Furthermore, optimization methods can be applied, and their advantages, disadvantages, and feasibility will be discussed. Finally, the surrounding workflow of the algorithm will be described, where the execution of the algorithm will be switched from UDF to a local operator implementation.

4.1 Basic Algorithm Design

4.1.1 Multi-Source BFS in Implementation

The current implementation of the shortest path finding algorithm by DuckPGQ is based on MF-BFS proposed by (21), as shown in Algorithm 4, which we will refer to as Basic Top-down Multi-Source Breadth-First Search. The term "basic" indicates that the algorithm is the basis for subsequent parallel algorithms. The algorithm searches from visited vertices to find unvisited neighboring vertices, referred to as top-down. Alternatively, the bottom-up

algorithm is to find neighboring vertices that have already been visited from the unvisited vertices. A discussion of the bottom-up approach in the context of direction optimization will be presented in Section 4.5.

Algorithm 4 Basic Top-down Multi-Source Breadth-First Search

```
    change ← true
 2: while change = true do
        next ← ∅, change ← false
 4:     for each v ∈ V do
            if visit[v] ≠ ∅ then
 6:             for each n ∈ neighbors_v do
                    next[n] ← next[n] | visit[v]
 8:             end for
            end if
10:     end for
        for each v ∈ V do
12:         if next[v] ≠ ∅ then
                next[v] ← next[v] & ∼seen[v]
14:             seen[v] ← seen[v] | next[v]
                change ← change | next[v]
16:         end if
        end for
18:     ReachDetect()
        visit ← next
20: end while
```

Unlike the original theoretical algorithm, Algorithm 4 is a pseudo-code closer to the actual implementation to represent the details better. First, there are three important variables, *visit*, *next*, and *seen*, all of which are structured as two-dimensional arrays. The outer layer corresponds to the vertex array V, which has the same length as V. The inner layer is a bitset to introduce SIMD acceleration, where each bit represents an independent BFS search, which we call a lane, and the length of the bitset can be set to any local machine-allowable SIMD vector length. An element of the variable $visit[v][l] = true$ means that on lane l, vertex v is a visited vertex of the current BFS search iteration, but *visit* only includes information about the current iteration. An element of the variable $next[v][l] = true$ means that on lane l, vertex v is considered the visited vertex for the next iteration. An element of the variable $seen[v][l] = true$ means that on lane l, vertex v

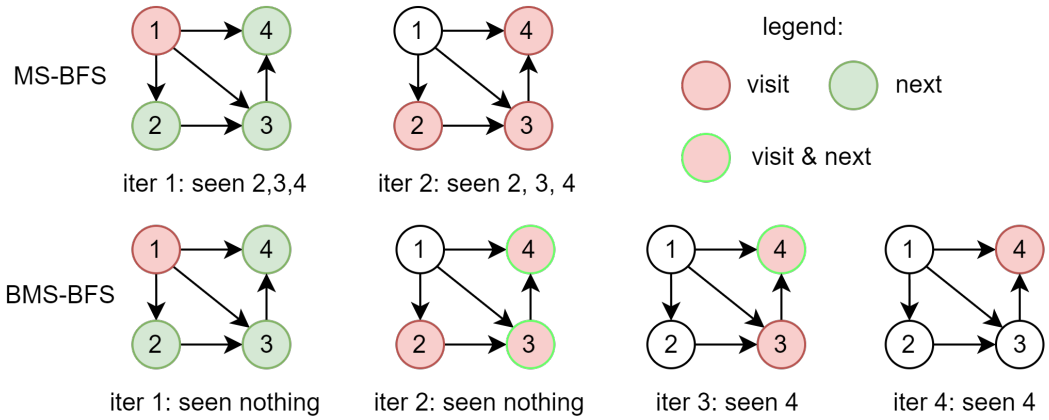

Figure 4.1: Finding the shortest path from 1 to 4 with a lower bound of length 3

has already been visited, and it is a superset of *visit*, containing information from earlier iterations.

Each run of the outermost loop of the algorithm indicates that the breadth-first search has completed one iteration. The interior of the loop is divided into four parts in a logical order. Line 3 is the first part used to initialize the variables for each iteration. Lines 4 to 10 are the second part used to scan each visited vertex on the frontier and find their neighbors. Line 11 to line 17 is the third part used to exclude already visited vertices from *next* so that the final path obtained is loop-free. Lines 18 to 19 are the final part to finish up, where we check if any of the destination vertices have been visited in the *seen*, and in addition, the *next* of this iteration will be the *visit* for the next iteration. we also have a variable *change* to indicate if the next iteration is necessary, as long as there are still vertices in the *next* the next iteration can be carried out.

4.1.2 Bounded Multi-Source BFS

Before discussing how to parallelize MS-BFS, this thesis proposes a shortest-path search algorithm that supports upper and lower bounds. Upper and lower bounds for shortest paths are defined as finding the shortest path whose length is between a given upper and lower bounds for a given source vertex s and destination vertex d. More unambiguously, the definition is equivalent to enumerating all possible paths between the upper and lower bounds and finding the shortest among them. Our proposed Bounded Multi-Source Breadth-First Search (BMS-BFS) is in the WALK mode, i.e., vertices and edges can be repeated in paths leading to loops, which is shown in Algorithm 5.

Algorithm 5 Bounded Multi-Source Breadth-First Search (BMS-BFS)

```
    Input: lower, upper
    change ← true
    iter ← 1
    while change = true and iter ≤ upper do
        next ← ∅, change ← false
        for each v ∈ V do
            if visit[v] ≠ ∅ then
                for each n ∈ neighbors_v do
                    next[n] ← next[n] | visit[v]
                end for
            end if
        end for
        for each v ∈ V do
            if next[v] ≠ ∅ then
                if iter ≥ lower then
                    next[v] ← next[v] & ∼seen[v]
                    seen[v] ← seen[v] | next[v]
                end if
                change ← change | next[v]
            end if
        end for
        if iter ≥ lower then
            ReachDetect()
        end if
        iter ← iter + 1
        visit ← next
    end while
```

Algorithm 5 demonstrates the design of BMS-BFS. The upper bound is easy to implement as we keep track of the number of iterations to avoid exceeding the upper bound. The implementation of the lower bound is related to finding the correct shortest path because the shortest path with a lower bound may require repeated visits to already visited vertices during the search. Shown in Figure 4.1 is a graph consisting of 4 vertices and 5 edges, we want to find the shortest path starting from vertex 1 to vertex 4 with a minimum length of 3. MS-BFS does not allow visited vertices to be revisited, it only allows us to find the global shortest path $1 \rightarrow 4$. Our proposed BMS-BFS is divided into two parts by the lower bound. When the number of iterations is lower than the lower bound, the visited vertices are not written into *seen* and the visited vertices are not excluded from *next*. When the number of iterations is equal to or greater than the lower bound, the normal MS-BFS logic is executed. These modifications are located in lines 15 through 18. Thus vertices 3 and 4 can be selected as *next* in the second iteration, vertex 4 can be selected as *next* in the third iteration, and the final path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ of length 3 can be searched.

We prove that the path found by BMS-BFS is the shortest of all paths $\mathbb{P}$ whose lengths are between the upper and lower bounds. At the end of the first phase of BMS-BFS the vertex located at the frontier $\mathbb{F}$ is at a distance $lower - 1$ from the source vertex s, and the set of paths is $\mathbb{P}_1$. At the end of the second phase of BMS-BFS, if the destination vertex d is found, the path $\mathbb{P}_2$ in the second phase is the shortest path from $\mathbb{F}$ to d. $\mathbb{P}_1$ is the prefix of $\mathbb{P}$ that has the same length and $\mathbb{P}_2$ is guaranteed to be the shortest, then $\mathbb{P}_1 \cup \mathbb{P}_2$ is the shortest of all paths $\mathbb{P}$.

4.1.3 Multi-source BFS Parallelization

4.1.3.1 Intuitive parallelization

In Chapter 2 we mentioned several key points for parallelizing an algorithm, including eliminating race conditions, reducing communication, and lowering workload imbalance. We apply these principles to parallelize MS-BFS. MS-BFS is hard to be parallelized with strong data dependency between different iterations, while the individual loops within each iteration have low data dependency. The time-consuming operations within an iteration are concentrated in three loops that scan V. The first loop traverses *next* sequentially, the second loop traverses *visit* sequentially and randomly accesses *next*, and the third loop traverses *next* and *seen* sequentially. The sequential scanning of arrays can be easily parallelized by simply dividing V into multiple parts V_w which we refer to as tasks, and

each thread w processes on one of them, finally aggregating the results. The algorithm after parallelization is shown in Algorithm 6.

Algorithm 6 Basic Top-down Parallel Multi-Source Breadth-First Search

```
    change ← true
 2: run on each w ∈ workers
        while change = true do
 4:          change ← false
            for each v ∈ V_w do                                  ▷ First loop
 6:              next[v] ← ∅
            end for
 8:          for each v ∈ V_w do                                  ▷ Second loop
                if visit[v] ≠ ∅ then
10:                 for each n ∈ neighbors_v do
                        next[n] ← next[n] | visit[v]             ▷ Random access
12:                 end for
                end if
14:         end for
            for each v ∈ V_w do                                  ▷ Third loop
16:             if next[v] ≠ ∅ then
                    next[v] ← next[v] & ∼seen[v]
18:                 seen[v] ← seen[v] | next[v]
                    change ← change | next[v]
20:             end if
            end for
22:         if workerId = 0 then
                ReachDetect()
24:         end if
            visit ← next
26:     end while
```

4.1.3.2 Data Race

Algorithm 6 is not correct because there is a lot of data race.

First, line 11 has a data race because access to *next* is randomized and multiple threads may access an element of *next* at the same time. This race does not exist in a single-source BFS because $next[n]$ will always end up being set to *true* by one of the threads rather than being written to $false$. However, the elements of a multi-source BFS are arrays, and

operations on *next* cannot be considered to be naturally atomic.

Line 11 consists of three instructions, first reading $next[n]$ from memory to a register, then computing $next[n] \mid visit[v]$ and staging the result in the register, and finally writing the result back to memory; they are all non-atomic instructions. The read must occur strictly after the writeback to avoid information loss. Suppose two threads t_0 and t_1 access $next[n] = [l_0, l_1]$ at the same time, and the read of t_1 occurs before the write-back of t_0, at which point both threads consider $next[n] = [l_0, l_1]$. Assuming that t_0 writes back $next[n] = [l'_0, l_1]$ and t_1 writes back $next[n] = [l_0, l'_1]$, the final result will be $next[n] = [l_0, l'_1]$ if the write-back of t_0 occurs first, and $next[n] = [l'_0, l_1]$ if the write-back of t_1 occurs first, whereas we expect the result to be $next[n] = [l'_0, l'_1]$. In other words, since the unit of computation is an array, elements that have not been updated by one thread may have been updated by other threads, leading to the presence of old information and possibly causing the old information to overwrite the new. Therefore, line 11 must be executed mutually exclusive across all thread scopes.

Secondly, there is also a data race between the four logical parts within an iteration. The second part randomly updates *next*, and it must wait for the first part to reset the entire *next*, otherwise, the update may be reset. The third part needs to read *next*, and since the second part updates *next* randomly, it needs to wait for the second part to execute completely to avoid missing information. The fourth part scans the list of destination vertices in the implementation to check if they appear in *seen*. Since the workload is not significant we assign it to thread 0 which will always be present. However, the third part needs to be executed completely. These four parts are logically causal and cannot be executed asynchronously. We could insert the reset of *next* into the fourth part to eliminate the wait in the second part, but this optimization breaks the logical readability of the algorithm and the reset task can be distributed equitably enough that we do not make it the final choice.

Thirdly, a data race occurs between iterations. In the fourth part of the previous iteration, after detecting that all the destination vertices have been found, we usually set *change* to $false$ to achieve early stopping. The next iteration needs *change* to determine if it can be started, so it must wait until the previous iteration has finished executing.

Finally, a data race also occurs when logging paths. In the serial implementation, we use the following fragment of the algorithm for logging paths:

for each $n, e \in neighbors_v$ **do**
2: $next[n] \leftarrow next[n] \mid visit[v]$
for $l \in S$ **do**

```
        if P_v[n][l] = ∅ and visit[v][l] = true then
            P_v[n][l] ← v
        end if
        if P_e[n][l] = ∅ and visit[v][l] = true then
            P_e[n][l] ← e
        end if
    end for
end for
```

Two two-dimensional arrays P_v and P_e are used to log the vertices and edges on the path, respectively. The line $P_v[n][l] = v$ means that vertex n has predecessor v on BFS instance l, the line $P_e[n][l] = e$ means that vertex n has edge e with its predecessor v on BFS instance l. The term S means the set of all lanes. Before assigning, it is necessary to check whether the predecessors and edges have already existed, this is to prevent destroying paths that have been constructed in previous iterations.

In a parallel environment, there is no guarantee that edges and vertices are matched. Suppose two threads t_0 and t_1 are processing vertices v_0 and v_1, and both vertices have a successor n at the same time, with corresponding edges e_0 and e_1. In this case, we have two alternative paths, $v_0 \xrightarrow{e_0} n$ and $v_1 \xrightarrow{e_1} n$. However, since the logging of the vertices and edges is executed separately, the result will likely be disordered to $v_0 \xrightarrow{e_1} n$ or $v_1 \xrightarrow{e_0} n$. So the logging of the vertices and the corresponding edges must be mutually exclusive. Considering that any arbitrary mutual exclusion technique would incur additional performance loss, we devise a lock-free logging approach with the core idea of using a single variable to log both vertices and edges:

```
for each n, e ∈ neighbors_v do
    next[n] ← next[n] | visit[v]
    for l ∈ S do
        if P_ve[n][l] = ∅ and visit[v][l] = true then
            P_ve[n][l] ← {v, e}
        end if
    end for
end for
```

Each element in P_{ve} is a 64-bit unsigned number whose first 30 bits are used to store vertices and the last 34 bits are used to store edges. The number of vertices and edges that can be represented by these bits exceeds the amount of data from the LDBC dataset when

$scale\ factor = 10000$, which is used to simulate 10,000 GiB of real data, so we believe that a 64-bit value is sufficient to represent both vertices and edges. For each assignment, vertices and edges are first composed into a new value, and then the new value is assigned to the elements in P_{ve} so that the assignment of vertices and edges is not disturbed by other threads. If more than one thread is assigned to an element at the same time, it means that there are multiple alternative paths, and we only need to get any one of them so no other synchronization measures are needed.

Algorithm 7 is the correct algorithm after eliminating the data race. We add **sync** at line 12 to indicate that the line of code needs to be executed mutually exclusive. Between parts, a **barrier** indicates that threads need to wait for all other threads to reach this point before continuing execution.

We analyze the performance of PMS-BFS through the key principles of parallel programs mentioned in Chapter 2:

- **Data partitioning:** Sequential access to arrays is naturally parallelizable without dependencies, i.e., one thread does not need to wait for the results of other threads. However, there are dependencies on random accesses of line 12. Additionally, there are dependencies between the different parts, where the next part needs to wait for the result of the former part.

- **Load balancing:** When each thread has the same $|V_w|$, the first and third loops have the same workload for each thread. However, the number of neighbors per vertex is variable, resulting in the second loop having varying workloads. We will discuss the workload balancing approach in detail in Section 4.4.

- **Communication overhead:** On a single machine, assignment of tasks, processing of tasks, and collection of results are all zero-copy operations on original arrays. However, the communication bottleneck exists between the CPU and memory. Each core requires a constant amount of bus bandwidth per unit of time, and as the number of threads grows, so does the bus bandwidth demand. When the demand of the threads exceeds the bus bandwidth, it leads to waiting.

- **Race conditions:** We eliminate the data race that exists in PMS-BFS through the **sync** and **barrier** instructions. However, the introduction of the synchronization mechanism makes it inevitable for threads to wait, where all threads need to wait for the thread that takes the longest to execute, which leads to performance degradation.

Algorithm 7 Top-down Parallel Multi-Source Breadth-First Search (PMS-BFS)

```
    change ← true
 2: run on each w ∈ workers
        while change = true do
 4:         change ← false
            for each v ∈ V_w do
 6:             next[v] ← ∅
            end for
 8:         barrier
            for each v ∈ V_w do
10:             if visit[v] ≠ ∅ then
                    for each n ∈ neighbors_v do
12:                     sync next[n] ← next[n] | visit[v]
                    end for
14:             end if
            end for
16:         barrier
            for each v ∈ V_w do
18:             if next[v] ≠ ∅ then
                    next[v] ← next[v] & ∼seen[v]
20:                 seen[v] ← seen[v] | next[v]
                    change ← change | next[v]
22:             end if
            end for
24:         barrier
            if workerId = 0 then
26:             ReachDetect()
            end if
28:         barrier
            visit ← next
30:     end while
```

- **Scalability:** Due to the synchronization mechanism, we believe that PMS-BFS cannot achieve linear speedup, i.e., the speedup ratio is equal to the number of threads. As the number of threads grows, individual threads take less time to execute, making the impact of the worst-performing threads increase, and the idle ratio of all threads increases. Also, the update to *next* in the second part of the algorithm is executed serially, and the execution time consumed grows linearly with the number of threads. It can be concluded from the analysis that as the number of threads increases, the speedup ratio first increases rapidly and then flattens out. When the number of threads is too high, the overall performance is degraded by the effect of synchronization. In addition, when the number of threads exceeds the number of physical cores available on the local machine, performance degradation also occurs because of scheduling between threads.

- **Data locality:** Except for global variables and unavoidable random accesses to *next*, each thread accesses only an entire block of memory within the task, with a high cache hit rate.

4.2 Barrier

There are many techniques for implementing the barrier, in this section, we choose three that are widely used and easy to implement: conditional variable, spin lock, and the POSIX thread (pthread) library. These techniques have advantages and disadvantages, and we will verify their performance in subsequent experiments.

4.2.1 Condition Variable

In C++, condition variables are a synchronization primitive used to block a thread until notified by another thread that a particular condition is true. The `std::condition_variable` class provides mechanisms for both waiting and notifying. The `wait` function is used to block the calling thread until it is awakened by a call to `notify_one` or `notify_all`. The `notify_all` function is used to wake up all threads currently waiting on the condition variable, making it particularly useful for implementing a barrier. A barrier ensures that all participating threads reach a certain point of execution before any of them proceed, by having each thread wait on the condition variable until the last thread reaches the barrier and calls `notify_all`, allowing all waiting threads to continue.

Under the hood, a condition variable in C++ operates by leveraging a combination of a mutex and a queue to manage the waiting and signaling of threads. When a thread calls `wait`, it first acquires the mutex to ensure mutual exclusion, then checks the condition in a loop to handle spurious wakeups. If the condition is not met, the thread is placed into the queue of waiting threads and the mutex is released, putting the thread into a blocked state. When another thread calls `notify_all`, the condition variable re-acquires the mutex, removes all threads from the queue, and transitions them back to a ready state. These threads then attempt to re-acquire the mutex to proceed. This mechanism ensures that waiting threads can be efficiently managed and notified in a thread-safe manner. Internally, this synchronization is typically implemented using platform-specific primitives such as futexes on Linux or condition variables on Windows, ensuring optimal performance and correct behavior across different operating systems.

Using condition variables for thread synchronization in C++ has several advantages, threads consume few resources while waiting and the code is portable. However, there are also disadvantages, such as the potential for spurious wakeups, where threads are awakened without any explicit signal, necessitating additional checks within a loop to revalidate the condition.

2 shows how we implement the barrier based on C++ conditional variables. Whenever a thread enters `Wait`, it decrements `mCount` by one. The last thread will reduce `mCount` to 0. Instead of waiting, the thread will call `notify_all` to wake up the other threads, and `mGeneration` will be incremented by one to represent the end of the current barrier, as well as resetting `mCount`. While blocking, threads avoid spurious wakeups by checking if `mGeneration` has changed.

4.2.2 Spin Lock

A spin lock barrier is a synchronization mechanism used in parallel computing to coordinate multiple threads, ensuring they all reach a certain point of execution before any can proceed. This is implemented using spin locks, which are simple locking mechanisms where threads repeatedly check if the lock is available, effectively "spinning" in a loop until they acquire the lock. The primary advantage of spin lock barriers is their efficiency in scenarios with short wait times, as they avoid the overhead associated with context switching, making them ideal for high-performance, fine-grained parallel applications where locking periods are brief and contention is low(93, 94).

However, spin lock barriers also have notable disadvantages. The busy-waiting nature of spin locks can lead to significant CPU resource wastage, especially under high contention,

Listing 2 C++ Condition Variable Barrier Implementation

```
class Barrier {
public:
    explicit Barrier(std::size_t iCount) :
      mThreshold(iCount),
      mCount(iCount),
      mGeneration(0) {
    }
    void Wait() {
        std::unique_lock<std::mutex> lLock{mMutex};
        auto lGen = mGeneration;
        if (!--mCount) {
            mGeneration++;
            mCount = mThreshold;
            mCond.notify_all();
        } else {
            mCond.wait(lLock, [this, lGen] { return lGen != mGeneration; });
        }
    }
private:
    std::mutex mMutex;
    std::condition_variable mCond;
    std::size_t mThreshold;
    std::size_t mCount;
    std::size_t mGeneration;
};
```

as threads consume processing power while waiting for the lock. This can degrade overall system performance. Additionally, implementing spin locks correctly is challenging due to the need for atomic operations to avoid race conditions, which may not be efficiently supported on all hardware architectures(94, 95). Despite these challenges, spin lock barriers are valuable in scenarios where their simplicity and low overhead in low-contention situations outweigh their drawbacks.

Listing 3 shows our code for implementing a spin lock-based barrier. Similar to the conditional variable, all threads enter the loop waiting except the last thread. The last thread adds one to `mGeneration` to notify the other threads that the wait is over. The

variables `mCount` and `mGeneration` must be atomic to avoid synchronization errors.

Listing 3 Spin Lock Barrier Implementation

```
class Barrier {
public:
  explicit Barrier(std::size_t iCount) :
    mThreshold(iCount),
    mCount(iCount),
    mGeneration(0) {
  }
  void Wait() {
    auto lGen = mGeneration.load();
    if (!--mCount) {
      mCount = mThreshold;
      ++mGeneration;
    } else {
      while (lGen == mGeneration.load()) {
        std::this_thread::yield();
      }
    }
  }
private:
  std::mutex mMutex;
  std::size_t mThreshold;
  std::atomic<std::size_t> mCount;
  std::atomic<std::size_t> mGeneration;
};
```

4.2.3 Pthread Library

In our project, we also utilize the POSIX Threads (Pthreads) library to implement a barrier. POSIX Threads, defined by the IEEE POSIX 1003.1c standard, provides a standardized API for creating and managing threads, facilitating efficient parallel execution in shared memory systems. Pthread barriers are implemented using the `pthread_barrier_t` type, which encapsulates the state of the barrier. To use a barrier, it must first be initialized with `pthread_barrier_init`, which sets the number of threads that must call `pthread_barrier_wait` before any of them can proceed. Each thread that

reaches the barrier calls `pthread_barrier_wait` and blocks until the specified number of threads have reached the barrier. Once the last thread reaches the barrier, all threads are released and can continue their execution. The underlying mechanism involves maintaining a count of the number of threads that have reached the barrier and using condition variables and mutexes to block and wake threads (96, 97). 4 shows how we encapsulate the pthread API.

The basic principle of the pthread barrier is similar to that of the C++ conditional variable-based barrier we defined in that it uses conditional variables. However, the pthread library is highly optimized for UNIX-like systems, leading to better performance on UNIX-like systems than the more general-purpose C++ condition variable library. Being oriented towards UNIX-like systems, the pthread library has compatibility issues with other systems. However DuckDB is a cross-platform DBMS, which makes the pthread library impractical to use, but we think it is still interesting for comparison.

Listing 4 Pthread Barrier Implementation

```
class Barrier {
public:
  explicit Barrier(std::size_t iCount) {
    pthread_barrier_init(&mBarrier, nullptr, iCount);
  }
  void Wait() {
    pthread_barrier_wait(&mBarrier);
  }
private:
  pthread_barrier_t mBarrier;
};
```

4.3 Synchronization

For the synchronization of the access to *next* in the neighbor exploration phase of the algorithm, there are many techniques to implement. The lanes in the original design are represented as bitsets, from which we can make atomic bitsets. In addition, classical locking mechanisms can be used for mutually exclusive accesses.

4.3.1 Atomic Bitset

In Algorithm 4 we define the data structure of *next*, *visit*, *seen* as `vector<bitset<LANE_LIMIT>>`. Access to each element in the vector can be done by SIMD vector instructions. The following is the assembly code of code `next[0] = next[0] | visit[0]` compiled by X86-64 gcc 14.1 with compilation option `-O3 -march=skylake-avx512 -mprefer-vector-width=512`:

```
vmovdqu64       zmm0, ZMMWORD PTR [r13+0]
mov             rdi, QWORD PTR [rsp+96]
mov             rsi, QWORD PTR [rsp+112]
vporq           zmm0, zmm0, ZMMWORD PTR [rbx]
sub             rsi, rdi
vmovdqa64       ZMMWORD PTR [rsp+128], zmm0
vmovdqu64       ZMMWORD PTR [rdi], zmm0
vzeroupper
```

Data transfer and `OR` operation on bitset are accomplished by SIMD instructions `vmovdqu64` and `vporq`.

On this basis, intuitively, we can atomize each bitset for mutually exclusive access. In C++, we can define data structures directly using `vector<atomic<bitset<LANE_LIMIT>>>`. The new assembly code for C++ code `next[0].store(next[0].load() | visit[0].load())` is:

```
...
mov             r14, QWORD PTR [rsp+96]
call            __atomic_load
vmovdqa64       zmm0, ZMMWORD PTR [rsp+256]
...
vmovdqa64       ZMMWORD PTR [rsp+128], zmm0
vzeroupper
call            __atomic_load
vmovdqa64       zmm0, ZMMWORD PTR [rsp+256]
...
vporq           zmm0, zmm0, ZMMWORD PTR [rsp+128]
vmovdqa64       ZMMWORD PTR [rsp+256], zmm0
vzeroupper
call            __atomic_store
```

When a bitset is atomized, operations on it can still be performed using SIMD instructions, but data reads and writes require explicit calls to built-in functions.

The atomized bitset only ensures that reads and writes to elements do not occur simultaneously, but the entire statement still cannot be executed mutually exclusive. We additionally need the Compare-And-Swap (CAS) mechanism to ensure that updates to *next* are not lost. CAS is an atomic operation used in concurrent programming to achieve synchronization without traditional locking mechanisms. CAS works by taking three parameters: a memory location, an expected old value, and a new value. The operation checks if the current value at the memory location matches the expected old value; if it does, the memory location is updated to the new value. If not, no change occurs, and the process can be retried. CAS is executed atomically, ensuring the check-and-update operation is indivisible, which prevents race conditions. However, CAS can lead to busy-waiting and higher CPU usage in high-contention scenarios.

CAS needs to poll the data, and we usually set the bitset size to 256 or 512 bits, making reads and writes expensive and putting a high load on the CPU bandwidth and cache. In some preliminary experiments, we have observed the low performance of atomized bitsets so we do not consider it an optional technique in the experiments section. The following code snippet demonstrates our modifications to the neighbor exploration phase of the parallel algorithm:

for each $n \in neighbors_v$ **do**
2: **do**
$oldNext \leftarrow next[n]$
4: $newNext \leftarrow oldNext \mid visit[v]$
while atomic_cas($next[n]$, $oldNext$, $newNext$)
6: **end for**

4.3.2 Atomic Segmented Lanes

On top of the atomic bitset, similar to Then et al.(11), we can segment the bitset to avoid having to read and write the full bitset each time, and to avoid calling the time-consuming built-in functions. Specifically, for a bitset that is 512 bits wide, we use eight 64-bit integer variables to represent it in segments. Each integer can be atomic to avoid race conditions. The new data structure is defined as `vector<array<atomic<idx_t>, 8>>`. In the meantime, we modify the Algorithm 7 to be compatible with the segmented bitset:

for each $n \in neighbors_v$ **do**

2: **for** $j \in [0, 8)$ **do**
do
4: $oldNext \leftarrow next[n][j]$
$newNext \leftarrow oldNext \mid visit[v][j]$
6: **while atomic_cas**$(next[n][j], oldNext, newNext)$
end for
8: **end for**

However, while inspecting the assembly code we found that the segmented bitset will not be able to utilize the SIMD instruction. For the following C++ code:

```
for (idx_t i = 0; i < 8; i++) {
    next[0][i] = next[0][i] | visit[0][i];
}
```

Its corresponding assembly code is:

```
        mov     rax, rbx
        mov     rcx, rbp
        lea     rdi, [rbx+64]
.L19:
        mov     rdx, QWORD PTR [rax]
        mov     rsi, QWORD PTR [rcx]
        or      rdx, rsi
        xchg    rdx, QWORD PTR [rax]
        add     rax, 8
        add     rcx, 8
        cmp     rdi, rax
        jne     .L19
```

We assume that this is due to the inability of the compiler to automatically recognize that the code snippet is vectorizable. It is possible to write correctly vectorized code using the low-level SIMD API provided by the system, but this poses a compatibility problem. The loss of the SIMD feature significantly affects serial programs because more instructions need to be executed, but we believe that this effect is attenuated in parallel programs. In the experiments section we use the program generated by the compiler without any explicit SIMD instructions

4.3.3 Lock

A lock is a synchronization mechanism used in concurrent programming to ensure that only one thread can access a critical section of code at a time, thereby preventing race conditions. In C++, the standard library provides `std::mutex` as a fundamental locking primitive. By employing `std::mutex`, a thread can lock a critical section by calling `lock()`, ensuring exclusive access, and unlock it by calling `unlock()` once the critical operation is complete. This mechanism guarantees mutual exclusion, making it a straightforward and reliable way to handle concurrent data access.

The performance bottlenecks of locks, particularly those implemented with C++ `std::mutex`, arise mainly from contention, context switching, and cache line effects. When multiple threads compete for the same lock, contention leads to threads being blocked and frequent context switching, which incurs significant overhead due to saving and restoring thread states. Locking operations involve atomic instructions such as compare-and-swap, which are inherently costly, ranging from tens to hundreds of nanoseconds depending on the platform and lock state. Moreover, lock contention causes cache line invalidations across different CPU cores, leading to cache line bouncing, further degrading performance by forcing frequent memory reloads. With simpler locks, such as spin locks, frequent cache invalidations also introduce synchronization overhead because threads that want to lock or unlock need to frequently query the critical zone. This issue is exacerbated by false sharing, where independent variables on the same cache line cause unnecessary invalidations.

Considering that the granularity of locks should be as small as possible to minimize collisions, but not too small to avoid consuming more than the protected code for locking and unlocking, we lock each bitset in the *next* vector. For *next* of length $|V|$, an array of mutexes of length $|V|$ is required to protect it. The following algorithmic snippet shows how we modify Algorithm 7:

for each $n \in neighbors_v$ **do**
2: **lock**$(mutex[n])$
$next[n] \leftarrow next[n] \mid visit[v]$
4: **unlock**$(mutex[n])$
end for

But defining an array of mutexes may bring low cache performance. This is because normally the CPU reads 64 bytes of data from memory at once, whereas `std::mutex` is 40 bytes on Linux and 80 bytes on x64_86 systems. Mutex locks and cache lines cannot be aligned and raise cache read and write requirements.

4.4 Task Distribution

Fair task distribution is crucial in parallel computing as it ensures that all processing units are utilized effectively, preventing some from being idle while others are overburdened. This balance maximizes resource utilization and reduces overall computation time, leading to improved performance and efficiency. When tasks are evenly distributed, the workload is processed concurrently, minimizing the time spent waiting for tasks to complete. This is especially important in high-performance computing applications where imbalanced workloads can lead to significant delays and inefficient use of computational resources. Fair distribution also helps in reducing bottlenecks and contention among threads or processes, thereby enhancing scalability and allowing systems to handle larger and more complex problems efficiently. Studies have shown that improper task distribution can lead to up to 30% degradation in performance due to idle CPU cycles and increased context switching (98). Therefore, employing strategies for dynamic load balancing and fair scheduling is essential for optimizing the performance of parallel systems.

Observing the PMS-BFS, we find that the workload imbalance mainly appears in the neighbor exploration part, because different vertices have different out-degrees. Accordingly, we design two different task creation strategies for the algorithm, namely dynamic task creation and static task creation.

4.4.1 Task Creation

In the neighbor exploration phase of PMS-BFS, the time consumption of threads is proportional to the out-degree of vertices on the frontier. To ensure that the work balance can be achieved by the number $numThreads$ of threads also called workers, we divide the vertex list V into uneven chunks, where the sum of the out-degrees of the vertices in each chunk is at least $taskSize$. During the traversal of the chunks, some vertices not on the frontier will be skipped, resulting in the actual workload being less than that contained in the chunks. We mitigate the skip impact by decreasing the $taskSize$ of each chunk so that a worker can extract tasks from the task list multiple times to attenuate the impact of a single task. At the same time, task creation follows a round-robin pattern so that the tasks belonging to each worker are distributed as evenly as possible on V to avoid some hot vertices being skipped frequently.

Figure 4.2 shows how dynamic task creation is performed on a graph with 12 vertices and 18 edges. The graph has three edges for vertices $V[0]$ and $V[9]$, two edges for vertices $V[4]$ and $v[10]$, and one edge for the other vertices. We create tasks for three workers, each

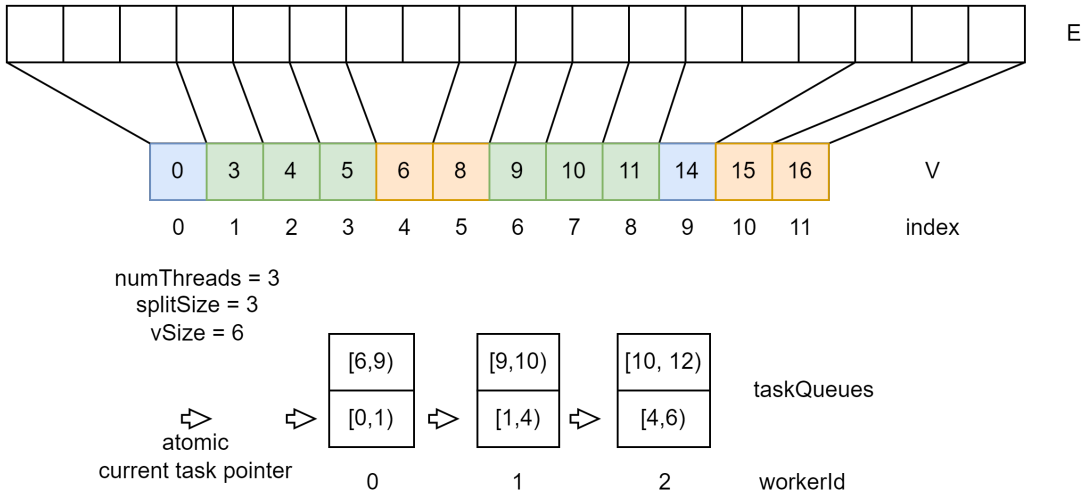

Figure 4.2: Dynamic task creation for graph with $|V| = 12$ and $|E| = 18$

with *splitSize* of 3. Each worker has its own queue of tasks. We traverse V, whenever the traversed vertices have no less than *splitSize* edges, these vertices are divided into a task chunk and assigned to a worker. Then the edge count is reset and the traversal of vertices continues with the next chunk being assigned to the next worker. When the last worker has been assigned a block, we revert back to the first worker for assignment. The result of task creation on this graph is that worker 0 has two tasks for index ranges $[0, 1)$ and $[6, 9)$, worker 1 has two tasks for $[1, 4)$ and $[9, 10)$, and worker 2 has two tasks for $[4, 6)$ and $[10, 12)$.

Algorithm 8 shows the details of dynamic task creation. The task queue for each worker is represented by $workerTasks[workerId]$, $curWorker$ indicates which worker needs to be assigned a task currently, *taskEdges* is used to count the number of edges while traversing V, and *taskStart* indicates the starting offset of the current chunk. During traversal, the number of edges at vertex i can be computed from $V[i + 1] - V[i]$ since the elements in V are upper bounds on the indexes of the edges of the previous vertices in E. Note that the last task is likely to be insufficient to form a complete chunk and needs to be treated specially. Once all tasks have been partitioned, we pair each $workerTasks[workerId]$ with an atomic task pointer. This pointer is used by the worker to retrieve the task from the queue. Dynamic task creation is efficient by traversing V only once.

In PMS-BFS, the initialization of *next* in the first part and the setting of *seen* and *next* in the third part do not involve access to edges, hence their workload is static. On this basis, for these two parts, we do not use dynamic tasks because of the time cost of fetching

Algorithm 8 Dynamic Task Creation

Input: $V, taskSize, numThreads$
$workerTasks \leftarrow \varnothing$
$curWorker \leftarrow 0$
$taskEdges \leftarrow 0$
$taskStart \leftarrow 0$
for $i = 0$ **to** $|V|$ **do**
  $vertexEdges \leftarrow V[i+1] - V[i]$
  if $taskEdges + vertexEdges > taskSize$ **and** $i \neq taskStart$ **then**
    $workerId \leftarrow curWorker \mod numThreads$
    $range \leftarrow \{taskStart, min(taskStart + taskSize, |V|)\}$
    $workerTasks[workerId] \leftarrow workerTasks[workerId] \cup range$
    $curWorker \leftarrow curWorker + 1$
    $taskStart \leftarrow i$, $taskEdges \leftarrow 0$
  end if
  $taskEdges \leftarrow taskEdges + vertexEdges$
end for
if $taskStart < |V|$ **then**
  $workerId \leftarrow curWorker \mod numThreads$
  $range \leftarrow \{taskStart, min(offset + taskSize, |V|)\}$
  $workerTasks[workerId] \leftarrow workerTasks[workerId] \cup range$
end if
$taskQueues \leftarrow \varnothing$
for $i = 0$ **to** $numThreads - 1$ **do**
  $taskQueues[i] \leftarrow \{0, workerTasks[i]\}$
end for
return $taskQueues$

tasks from the queue. We simply divide V equally into $numThreads$ chunks, such that each chunk contains a similar number of vertices, and each worker is assigned one chunk. The static assignment method ensures that workers have theoretically the same amount of work while each chunk is accessed consecutively with high cache hit performance. Algorithm 9 shows how we create tasks statically.

Algorithm 9 Static Task Creation

Input: $vSize, numThreads, workerId$
$blockSize \leftarrow \lceil \frac{vSize}{numThreads} \rceil$
$left \leftarrow blockSize \times workerId$
$right \leftarrow min(blockSize \times (workerId + 1), vSize)$
return $left$, $right$

4.4.2 Task Fetch

In static task creation, we directly return the range of the corresponding chunk to each worker, but in dynamic task creation, workers need to have a reasonable mechanism for extracting tasks from the queue. Initially, each worker can pull tasks from its own queue for processing. When a worker has processed all the tasks belonging to it while other workers are still running, it can try to steal work from other workers' queues. Work stealing is another mechanism to ensure that the workload of the workers is balanced, and it takes effect at runtime. Algorithm 10 shows how the fetching of tasks takes place. The variable $offset$ controls which queue the worker is currently accessing, when $offset = 0$ the worker accesses its own queue. Each queue has an atomic pointer to the currently available task in the queue, which ensures that only one worker acquires each task. After a worker accesses a queue, the pointer is atomically added to one. If the pointer is within the range of the queue, the task is available, otherwise, the worker tries to access the next worker's queue until all queues have been accessed.

4.5 Direction Optimization

Direction optimization in Breadth-First Search (BFS) focuses on improving efficiency by dynamically adjusting the search direction during neighbor exploration. Instead of always expanding the frontier from visited vertices, direction optimization alternates by exploring from the unseen vertices to the visited frontier when beneficial. This approach reduces the number of vertices and edges processed in each iteration, thereby speeding up the

Algorithm 10 Task Fetch

```
Input: taskQueues, workerId
offset ← 0
do
    i ← (workerId + offset) mod |taskQueues|
    taskId ← atomic_fetch_add(taskQueues[i], 1)
    if taskId < |taskQueues[i]| then
        return taskQueues[i][taskId]
    else
        offset ← offset + 1
    end if
while offset < |taskQueues|
return ∅
```

search process. By attempting to find frontier vertices from the set of unseen vertices, direction optimization minimizes unnecessary explorations and balances the workload, making it particularly effective in large-scale, highly connected graphs. This technique is highly advantageous in parallel and distributed computing environments, where it helps to reduce inter-processor communication and evenly distribute the computational load, leading to significant performance gains over traditional BFS methods. Many studies on optimizing breadth-first search have applied direction optimization (99, 100), but no study has explicitly addressed direction optimization under parallelization.

4.5.1 Bottom-up Parallel MS-BFS

When a regular BFS search is performed, the frontier starts at the source vertex and expands with iterations until it starts shrinking at some intermediate iteration, this process may take as little as 2 or 3 iterations to complete on small and dense graphs. After shrinking, the search continues until the destination vertex is found or all feasible paths have been visited but not found. During all of this time, the number of *seen* vertices gradually increases and the number of *unseen* vertices gradually decreases. At some intermediate iteration, the frontier is so large that finding neighbors from the frontier consumes more time than finding neighbors that are frontiers from *unseen* vertices. This is the basic principle of direction optimization, and finding frontiers from *unseen* vertices is called bottom-up BFS.

Algorithm 11 shows how we can modify the top-down PMS-BFS to bottom-up. Mainly the neighbor exploration phase of the algorithm is modified. When vertex v is not seen in

a lane, it has to be used as the starting point of bottom-up to explore whether a neighbor is in the frontier. If there is a neighbor vertex n within $visit[n]$, $next[v]$ is to be set to true. Since the element being updated does not collide between multiple threads, there is no longer the need to guarantee mutual exclusion. In addition, the barrier between the second and third part is no longer needed since each thread only updates vertices that are within its own task and there is no race condition.

Algorithm 11 Bottom-up Parallel Multi-Source Breadth-First Search

```
    change ← true
 2: run on each w ∈ workers
        while change = true do
 4:          change ← false
            for each v ∈ V'_w do
 6:              next[v] ← ∅
            end for
 8:          barrier
            for each v ∈ V'_w do
10:             if seen[v] has false then
                    for each n ∈ neighbors_v do
12:                     next[v] ← next[v] | visit[n]
                    end for
14:                 if next[v] ≠ ∅ then
                        next[v] ← next[v] & ~seen[v]
16:                     seen[v] ← seen[v] | next[v]
                        change ← change | next[v]
18:                 end if
                end if
20:         end for
            barrier
22:         if workerId = 0 then
                ReachDetect()
24:         end if
            barrier
26:         visit ← next
        end while
```

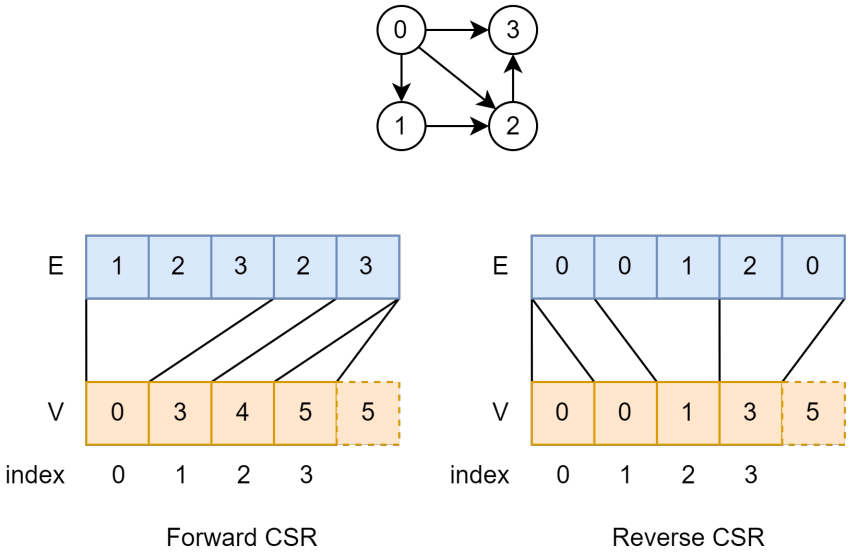

Figure 4.3: Forward and reverse CSRs for a graph

4.5.2 Reverse CSR

Since the type of graphs DuckPGQ deals with is directed and cyclic, the information about edges contained in CSR is unidirectional, which we refer to as forward CSR. it is easy to find child vertices from the parent vertices in forward CSR by traversing the set of edges pointed to by the parent vertices only, but it is very difficult to do so in reverse. Therefore, to work with bottom-up PMS-BFS, a reverse CSR needs to be created at the same time as the forward CSR. As shown in Figure 4.3 is the forward and reverse CSRs of a graph containing 4 vertices and 5 edges, each vertex of the forward CSR points to its set of child vertices, and each vertex of the reverse CSR points to its set of parent vertices. So we can use the reverse CSR to find its parent vertices in the frontier from the unseen vertices.

4.5.3 Direction Switching

Two types of metrics are commonly used in the current literature(11, 36, 101) to determine the timing of switching directions, with the core idea being to estimate the workload of the next iteration. The first is the number of vertices on the frontier, n_f, and the number of unvisited vertices, n_u, with top-down search enabled when $n_f < \frac{n_u}{\alpha}$ and bottom-up search enabled when $n_f >= \frac{n_u}{\alpha}$, where α is an empirical value. The second is the sum of the degrees of the vertices on the frontier, m_f, and the sum of the degrees of the unvisited vertices, m_u, with top-down search enabled when $m_f < \frac{m_u}{\beta}$, and bottom-

up search enabled when vice versa, where β is an empirical value. The degree-based workload estimation is more accurate. However, on some single-source BFSs (102), the vertex count-based approach performs better because counting vertices is simpler and less consuming than counting edges. But in multi-source BFS, we have to start counting from zero in each iteration and cannot use data from the previous iteration, so there is no significant performance gap between the two methods. Also, metrics based on the number of vertices struggle to have similar performance on dense and sparse graphs. In subsequent experiments, we choose the degree-based metrics to determine the direction switching.

The next fragment of the algorithm shows how we count the sum of degrees, where V denotes the list of vertices in the forward CSR and V' denotes the list of vertices in the reverse CSR:

if $next[v] \neq \varnothing$ **then**
    $m_f \leftarrow V[v+1] - v[v]$
end if
if $seen[v]$ has false **then**
    $m_u \leftarrow V'[v+1] - V'[v]$
end if

4.5.4 Effectiveness Argument

On a single-source BFS, it is intuitive that m_f grows and m_u decreases as iterations proceed. However, on a multi-source BFS, the growth of m_f can be certain, but m_u may not decrease as iterations proceed. The reason for this is that multi-source BFS operates on multiple lanes, and as long as vertex v has not been visited in any of the lanes, its degree needs to be added to m_u. If multiple sources are spread out on the graph, it is difficult to make it possible for v to have been visited on all lanes, and this becomes less likely as the number of lanes increases. We select the graph at $scale\ factor = 10$ from the LDBC dataset, which has 68 673 vertices and 1 839 354 edges, and randomly select 1 and 512 sources and destination pairs on the graph, respectively. Figure 4.4 shows the change of m_f and m_u for each iteration when the number of lanes is 512.

As can be seen from Figure 4.4, when there is only one pair of source and destination vertices, the changes of m_f and m_u are close to the single-source BFS. it is worth noting that m_f decreases abnormally on the 5th and 6th iterations, which may be due to the frontier entering the articulation point of the graph. When there are 512 pairs, it is clear that the variation of m_f is smoother, showing an increase followed by a decrease. However

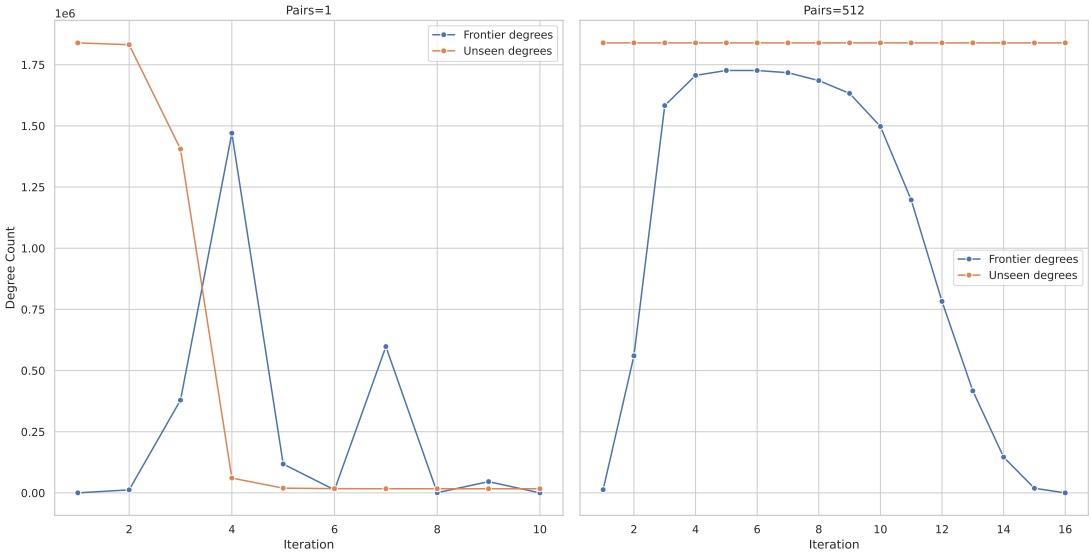

Figure 4.4: Frontier degree (m_f) and Unseen degree (m_u) for 1 and 512 pairs search

m_u stays maximal, indicating that no vertex is visited on all lanes from the beginning to the end. This suggests that bottom-up will always be more computationally intensive than top-down. However, we assume that in this case when m_u is slightly larger than m_f bottom-up search is still worth enabling because it does not need to synchronize during neighbor exploration while using one less barrier. This may bring performance advantages to the bottom-up approach. We can use the empirical value β to adjust the threshold for direction switching. When β is greater than 1, it allows the bottom-up search to be initiated even if m_u is greater than m_f. We will explore the optimal β in the experiment section.

4.6 Operator Workflow

Figure 4.5 shows the workflow within the shortest path finding operator.

After properly parallelizing MS-BFS to PMS-BFS, we can integrate it into DuckDB's shortest path finding operator.

At DuckDB, each operator has common interfaces. There are multiple LocalState and one GlobalState within an operator. LocalState refers to the work that a thread carries on a single morsel based on morsel-driven parallelism as the lower-level operator pushes data upwards. LocalState is eventually merged into GlobalState. The process of pushing data by the lower level operator is called Sink and the process of LocalState being merged into GlobalState is called Combine. After all the Combines are over the complete data is

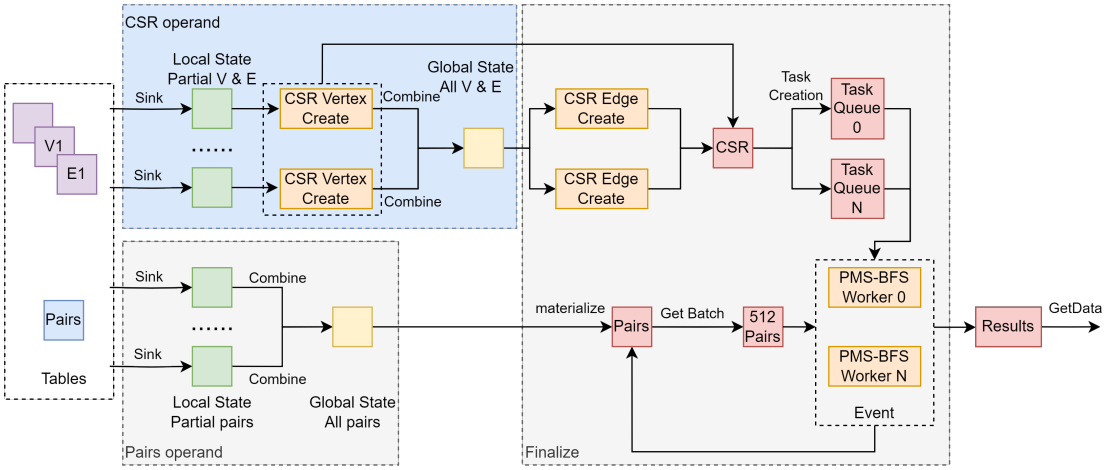

Figure 4.5: Workflow for path-finding operator

obtained and the operator enters the Finalize phase. After Finalize is executed, the result data obtained is pushed upwards by the GetData interface.

Our operators follow these interfaces. There are two kinds of Sinks within the operator, corresponding to the two operands of the operator, CSR creation and src-dst pairs collection.

In the CSR operand, the data obtained from the scanning of the vertex and edge tables from the lower level will be pushed to the operator concurrently by the Sink interface. The input data for each input is the vertices and the number of outgoing edges for each vertex. The value of each element of the vertex list in CSR is the upper bound of the offset of the previous vertex's edges in the edge list, which corresponds to how many edges there were before that vertex. We can set the number of edges for each vertex in the vertex list in parallel, and then scan over the list once and compute the rolling sum. So the vertex list can be constructed in parallel at the same time during the Sink process. Because the scan-and-compute for constructing the vertex list can only be done after all the CSR Sinks have been materialized, constructing the edge list can also only be done afterward.

The process is similar for the pairs operand. The lower-level scan operator scans the data from the pairs table and then concurrently pushes the data through the Sink interface. The Combine interface is also called to assemble to get all the pairs.

When both sinks are materialized, the Finalize interface will be called. The first step is to complete the construction of the edge list for the CSR. First, the vertex list is scanned over to compute and set the rolling sum, where the offset of each vertex's edge on the edge list is known. After that, the edge list can be constructed in parallel. Thereafter tasks can

be partitioned on the CSR to get the task queue for each worker. For pairs, first, the global state needs to be materialized because the complete arrays of source and target vertices need to be obtained. After two arrays are ready, each batch has up to 512 pairs from which to extract to PMS-BFS, corresponding to the number of lanes set by the algorithm. We set the number of lanes to 512 because the widest SIMD array (with AVX-512) is 512.

The above process follows DuckDB's push-based execution model. The two operands correspond to two separate pipelines with morsel-driven parallelization on the pipeline. They can be abstracted as Execute events. When all Execute events are finished, Finalize events are fired, which computes all the data obtained. When Finalize events are completely finished, Complete events are fired to indicate that the pipeline execution is finished. Our MS-BFS parallelization extends the event mechanism. Between the Finalize event and the Complete event, we insert BFS events, with custom parallelization within each BFS event and serial execution between BFS events, corresponding to multiple batches of BFS.

More detailed, PMS-BFS is wrapped by `duckdb::BasePipelineEvent`, where multiple `ExecutorTasks` can be defined in each Event corresponding to the workers. In an event, any number of workers can be defined and started, and the scheduling between workers is hidden by DuckDB. An event ends when all the workers have finished executing. This mechanism is an abstract encapsulation of threading, providing developers with an easy-to-use generic threading interface. As a result, we replace DuckDB's default morsel-driven parallelism with our custom implementation of the parallelism mechanism.

After the execution of each batch event is completed, the shortest path found will be inserted into the result. After finally combining the results of all batches, the operator will push the result data to the upper level via the GetData interface.

5

Evaluation

We verify the performance of parallelized shortest path finding through a series of experiments. First, we perform performance fine-tuning validation to check which barriers, synchronization techniques, task sizes, and number of threads are optimal. The effectiveness of the direction optimization is also verified. After choosing the best tuning parameters, we compare the best parallel results with the serial results. Finally, we check the percentage of time consumed by each part inside the operator to identify possible performance bottlenecks.

5.1 Experiments Setup

5.1.1 Environments

All experiments were performed on the same machine, which had the following hardware configuration:

- DuckDB version v0.10.2 Development
- 2 NUMA nodes
- 2 × Intel(R) Xeon(R) Gold 5115 CPU @ 2.40GHz, 10 cores with Hyper-Threading; total 20 cores and 40 threads
- 384 GiB RAM
- L1 data cache: 32 KiB per core; L1 instruction cache: 32 KiB per core; L2 cache: 1024 KiB per core; L3 cache: 13.75 MiB shared
- Compiler: GCC 14.1.1

Scale factor	Number of vertices	Number of edges
1	10 620	219 450
3	25 870	668 431
10	70 800	2 304 951
30	175 950	6 880 584
100	487 700	23 116 805
300	1 230 500	68 313 982

Table 5.1: Number of vertices and edges for different scale factors

- Operating system: Fedora release 40

5.1.2 LDBC Social Network Benchmark

The Linked Data Benchmark Council (LDBC) (103) provides datasets and standardized benchmarks designed to evaluate the performance of graph database systems. It provides a comprehensive set of graph data and queries that simulate real-world scenarios, such as social networks, to test various aspects of graph processing capabilities, including query performance, scalability, and data loading efficiency. The LDBC provides different types of benchmarks, including the Social Network Benchmark (SNB) (104), specifically tailored to model the complex interactions and relationships found in social network data.

A crucial aspect of the SNB dataset is the concept of the scale factor (SF), which determines the size of the dataset. The scale factor allows the dataset to be scaled up or down to meet different testing requirements. For example, a scale factor of 1 (SF1) represents a smaller dataset suitable for testing on a single machine, whereas a scale factor of 1000 (SF1000) would generate a significantly larger dataset intended for testing on distributed systems or high-performance computing environments. In general, the value of SF can be viewed approximately as how many GiB of data there are.

We select SNB as the dataset and choose scale factors from 1 to 300. The original dataset will be trimmed down to just two tables, `Person` and `Person_Knows_Person`, which will be used to compose the graph. `Person` is used as vertices on the graph and edge information is obtained from `Person_Knows_Person`. These graphs are relatively sparse with an average degree of 200 for millions of vertices. Usually, BFS traversal can be accomplished within seven to eight iterations. Table 5.1 shows the number of vertices and edges in the graph at each scale factor. For pairs, we randomly select 1 to 4096 pairs of source and destination vertices from each graph.

5.2 Tuning Performance

According to the design, we have five fine-tuning parameters, namely the technique of barrier, the technique of synchronization, the number of threads, the size of the task, and the β in direction optimization. This subsection determines the optimal parameters through experiments. To speed up the experiments, all experiments have removed the recording of paths because it is very time-consuming. The experiments in this section only query the length of the shortest path between vertices.

5.2.1 Barrier

For the barrier, we mentioned that there are three implementations, which are based on C++ condition variables, based on spinlock, and based on the pthread library, in addition, we removed the barriers to observing the effect of barriers on parallel performance. Synchronization and barrier are two separate events, in the experiments the synchronization technique is always lock to reduce the noise associated with synchronization. The number of threads is highly correlated with the performance of the barrier, we chose 1, 2, 4, 8, 16, 32, 64, and 128 number of threads for our experiments. The task size may have impact on the barrier efficiency, because the larger the task, the more likely the workload is unbalanced, we chose 128, 256, 512, and 1024 as the experiment configuration. Direction optimization is disabled in this experiment. The number of pairs was fixed to 4096 such that each BFS was run filled and was run for 8 batches to reduce noise.

Figure 5.1 illustrates the experiment results. There are six subgraphs, each representing results on SF from 1 to 300. The x-axis of each subgraph represents the number of threads, and the y-axis represents the complete execution time of the shortest-length query. Each line in the figure represents the query execution time for one pairing of barrier and task size. It can be observed that each barrier possesses a similar time profile, showing a decrease followed by an increase with the increasing number of threads on each SF. The increase in execution time when the number of threads is too large is because threads exceeding the number of physical cores lead to scheduling, reducing the degree of parallelism. Where the condition variable performs slightly worse than pthread for large thread numbers, suggesting that pthread is better optimized for large numbers of threads. The spinlocks, on the other hand, show a very noticeable performance degradation for large thread numbers at $SF < 100$, considering that the machine has two NUMA nodes, we believe that this is a performance loss due to node synchronization caused by frequent polling of atomic

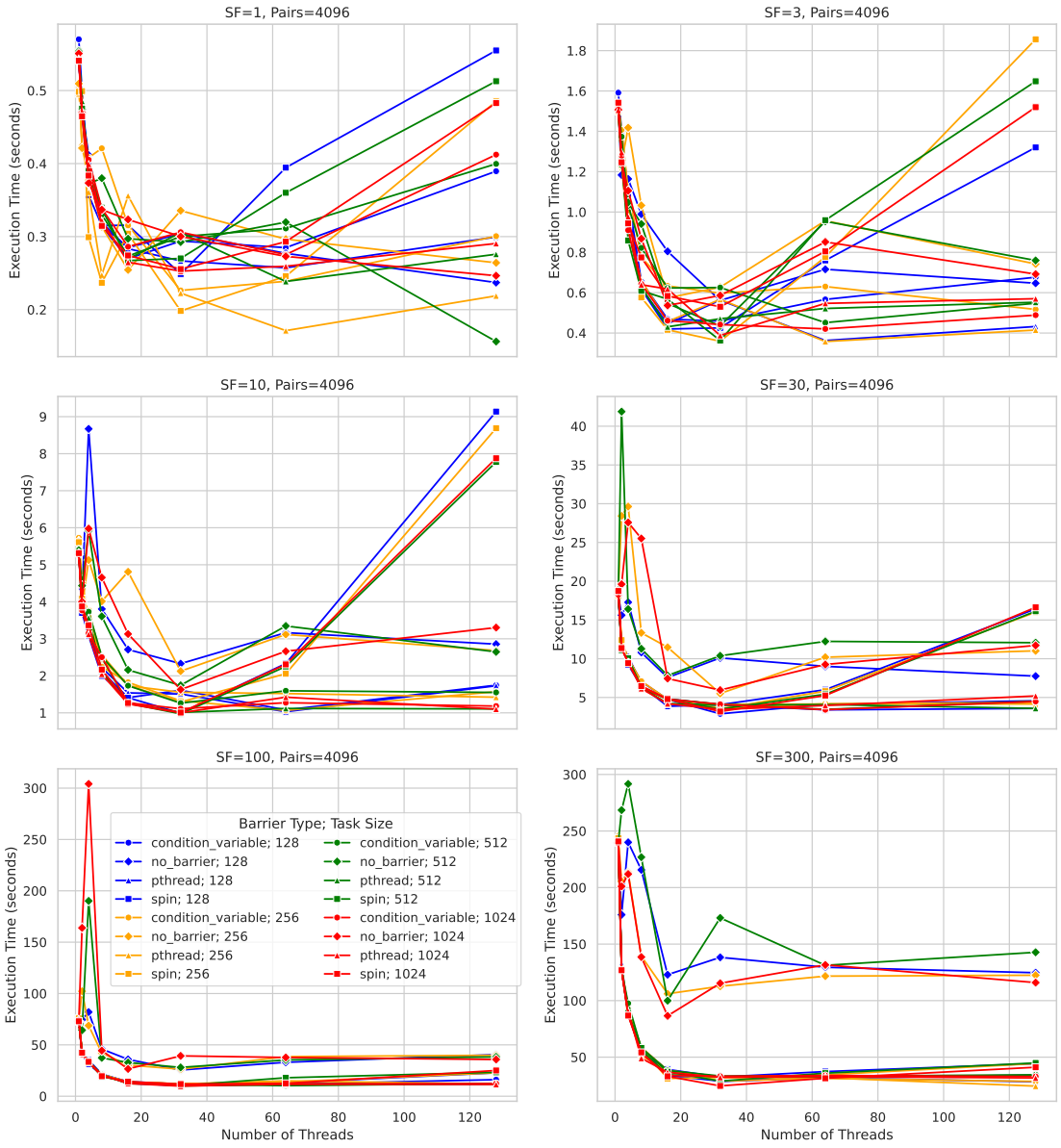

Figure 5.1: Execution time for different barrier implementations

variables. When PMS-BFS has no barriers, the execution time becomes longer, and the performance gap reaches 4x when $SF = 300$, indicating that the loss of information causes additional iterations to be executed. Lastly, task size had no significant effect on the performance of the barrier. Considering various techniques, we finally pick the condition variable as the best implementation because it achieves good performance and can follow DuckDB's cross-platform requirements

5.2.2 Synchronization

For the neighbor exploration synchronization, we mentioned that there are three implementations, which are atomic bitset, atomic segmented bitset, and lock. As we discussed in Section 4.5, because the atomic bitset has shown poor performance in the design phase, we do not consider it. In addition, we removed the synchronization to observe the effect of synchronization on parallel performance. The barrier has been determined by previous experiments to be based on the condition variable. The number of threads is highly correlated with the performance of the synchronization. We chose 1, 2, 4, 8, 16, 32, 64, and 128 number of threads for our experiments. For task size, we chose 128, 256, 512, and 1024 as the experiments configuration. Direction optimization is disabled in this experiment. The number of pairs was fixed to 4096 such that each BFS was run filled and was run for 8 batches to reduce noise.

Figure 5.4 shows the results of the experiments. The definition of each subplot is the same as for the barrier experiments. It can be seen that the time curves for each synchronization technique are similar, showing a rapid decrease and then a slight increase. Lock-based synchronization has significantly better performance than segmented bitset, since the latter introduces serial code, this performance gap is to be expected. Similar to the barrier experiments, task size has no significant effect on synchronization performance. Given the obvious performance difference, we chose the lock as the best implementation.

We can also note that queries without the synchronization mechanism have significantly better performance on all datasets. The performance gap between with and without synchronization is largest when the number of threads is small, illustrating the substantial performance loss associated with synchronization. However, even without synchronization, the time curve is still similar to that with synchronization, where the query fails to achieve linear speedup, suggesting that the bottleneck in the algorithm comes from outside of synchronization. The results above are for the entire query, which contains parts that cannot be parallelized, giving incorrect speedups. To compare the speedups more accurately, we removed the barriers and synchronization from PMS-BFS and only used

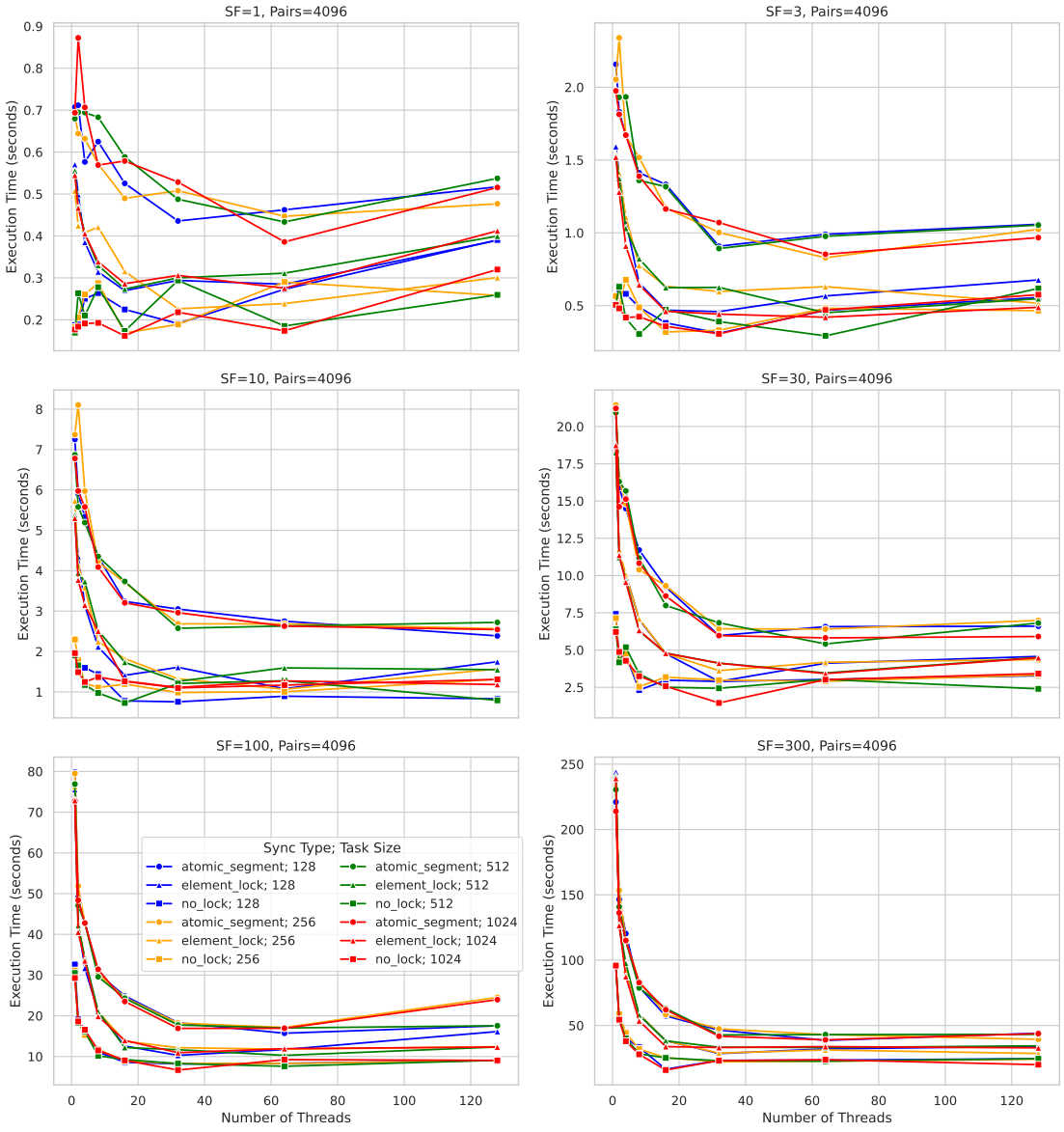

Figure 5.2: Execution time for different synchronization implementations

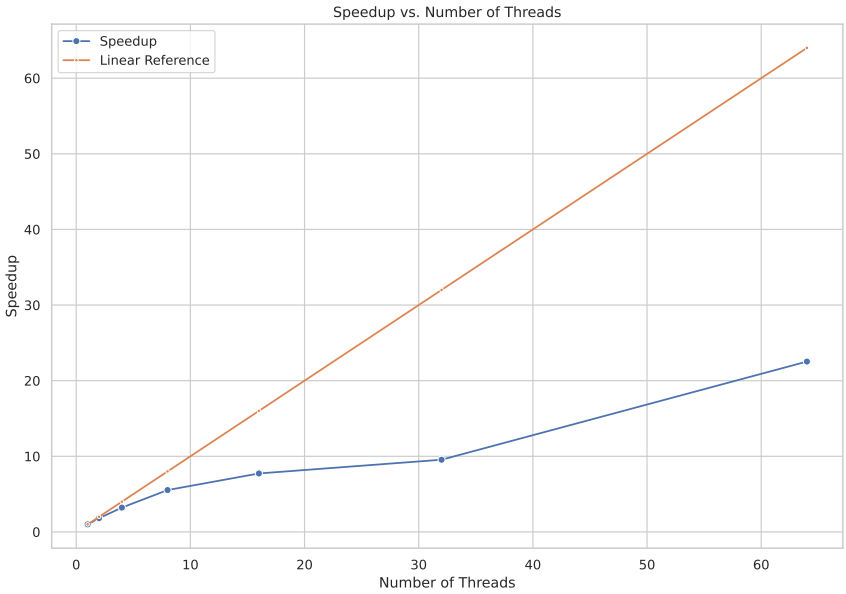

Figure 5.3: No synchronization and no barrier PMS-BFS Speedup for different threads on $SF = 300$ and $pairs = 4096$

bottom-up search since it has the same amount of work in each iteration. As shown in Figure 5.3 is the speedup of the modified algorithm when the number of threads goes from 1 to 64. Unlike previous experiments, the speedup here is calculated purely on the running time of PMS-BFS itself, not the time of the query. It can be seen that the speedup is far from linear, indicating that there is indeed a bottleneck affecting performance beyond synchronization.

5.2.3 Threads Number

We now explore the impact of the number of threads on parallel performance. When parallelizing a serial program, the best-case performance is that every time the number of threads doubles, the execution time of the program is halved. However, due to the presence of unserializable code, communication overheads, and synchronization overheads, it is often not possible to achieve a linear speedup. We choose condition-variable-based barrier, and lock-based synchronization, to perform experiments on the number of threads from 1 to 128 and task sizes from 128 to 1024.

Figure 5.4 illustrates the results of the experiment, with the same definitions of the subplots as in the previous experiments. It can be seen that as the number of threads

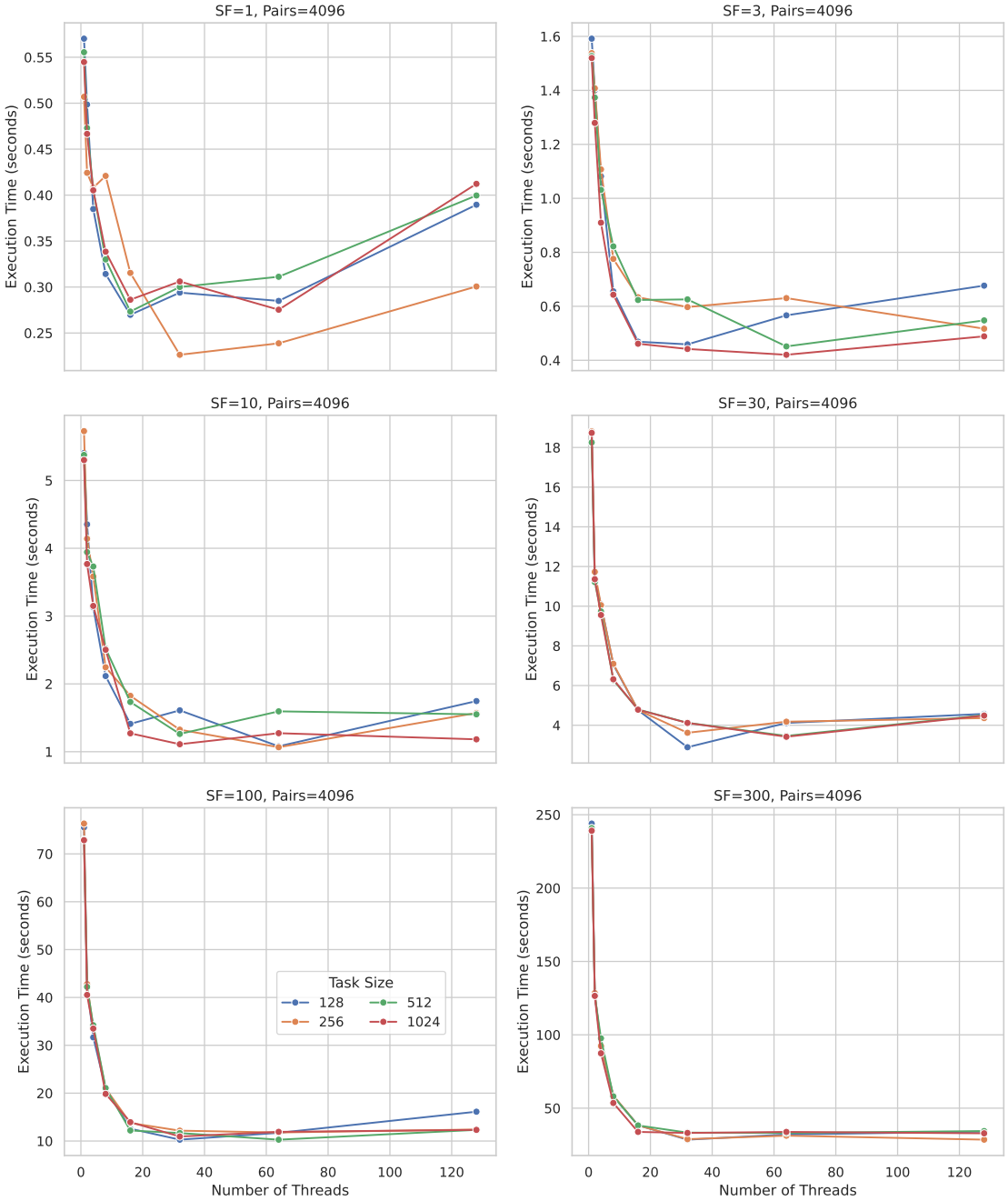

Figure 5.4: Execution time for different threads number and task size

increases, the speedup gradually decreases. From a single thread to two threads, double the speedup can be achieved on large-size graphs, this phenomenon is less noticeable with smaller graph sizes, because the percentage of time taken by the part of the program that is not parallelized increases. The machine has 20 physical cores and 40 concurrent threads, we can see that after the number of threads exceeds 20, the speedup multiplier still grows, indicating that the hyperthreading technique helps the concurrency of our program. When $SF = 1$, the performance gradually decreases after the number of threads is greater than 16, indicating that the overhead from synchronization and scheduling is increasing. This phenomenon also exists in other SFs, but it is not obvious because BFS itself consumes a lot of time. In addition, the correlation between task size and number of threads is not obvious. Finally, considering the convenience of the experiment, we choose 32 as the optimal number of threads. In a real product, all physical cores should be used.

5.2.4 Task Size

Although we did not observe any significant effect of task size on performance in our previous experiments, for a more comprehensive comparison we plot Figure 5.5. The first six subplots in the figure correspond to each combination of a barrier and a synchronization method, and the last subplot is the sum of the first six subplots. The x-axis of each subfigure represents the task size and the y-axis represents the number of threads. Each grid in the subplot means the number of times that the PMS-BFS of that configuration performs best on datasets with SF ranging from 1 to 300 and pairs ranging from 1 to 4096 in comparison with other possible configurations. In other words, the larger the number in the grid, the better the robustness of that configuration on each type of payload. No significant effect of task size on the performance of the algorithm can be observed from the figure. We choose 256 as the final task size.

5.2.5 Direction Optimization

Recall that in the design phase we assumed that although bottom-up PMS-BFS typically has more accesses across iterations than top-down, the nature of not requiring synchronization may allow it to achieve better performance on some iterations. In this subsection, we verify the correctness of this assumption through experiments and try to find the optimal β. After applying the optimal parameters obtained from previous experiments, we enable bottom-up search and set β from 1.0 to 3.0. This is because the value is the multiplier of the theoretical computation of the bottom-up over the top-down method.

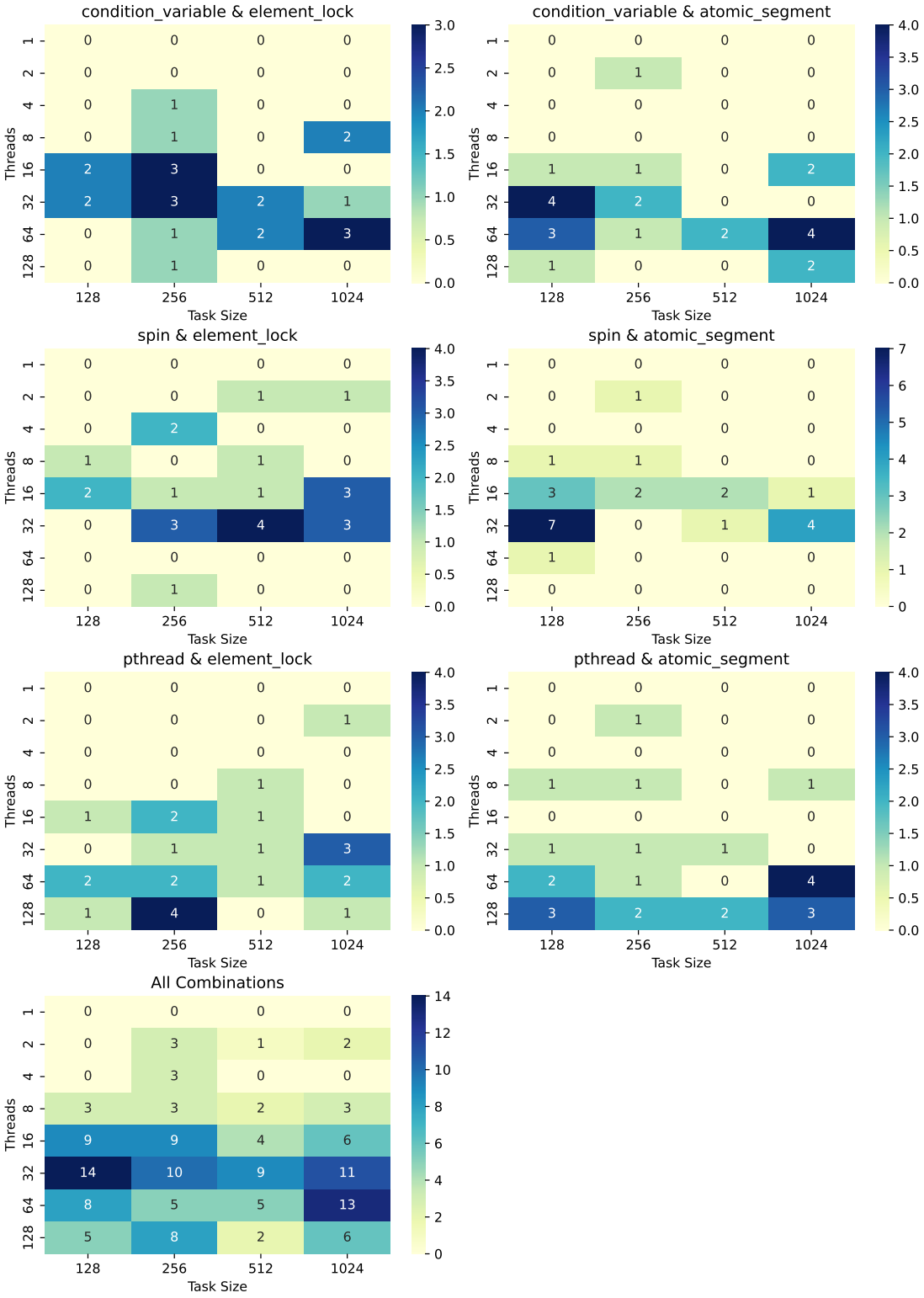

Figure 5.5: Number of times the specified task size and number of threads reached the fastest execution time

SF	Top-down	$\beta = 1.0$	$\beta = 1.25$	$\beta = 1.5$	$\beta = 1.75$
1	0.219836	0.407989	0.408516	0.231391	0.434389
3	0.433990	0.658036	0.464988	0.459835	0.463083
10	0.933430	1.733386	**0.806883**	1.128601	1.088325
30	2.584600	3.772880	**2.047760**	3.919304	**2.080028**
100	7.849802	11.351795	**7.194967**	12.653576	13.466585
300	21.877030	32.020254	**19.457272**	**17.504649**	37.523230
SF	$\beta = 2.0$	$\beta = 2.25$	$\beta = 2.5$	$\beta = 2.75$	$\beta = 3.0$
1	0.321863	0.411080	0.305971	0.416304	0.240690
3	0.943509	0.568147	**0.391666**	0.516448	**0.387394**
10	1.073819	1.108598	1.912498	1.919550	1.007618
30	4.532985	5.993518	**2.207222**	6.141169	**2.436672**
100	13.346917	13.654138	**7.174697**	**6.400421**	15.923058
300	**20.256315**	41.702507	37.182964	41.895926	**20.926702**

Table 5.2: Query execution time for top-down PMS-BFS and bottom-up PMS-BFS on different β, $Pairs = 4096$

Bottom-up methods can only be initiated with more computations when β is greater than 1. When β is greater than 3, the difference in the amount of computation is too large and likely exceeds the difference in performance due to synchronization.

Table 5.2 shows the query completion times for top-down PMS-BFS and bottom-up PMS-BFS with different βs when the number of pairs is 4096. It can be seen that the bottom-up method is not faster in most cases. The optimal β is 1.25, at which point the bottom-up method is slightly faster. There are two main reasons for the unsatisfactory performance of the bottom-up method. First, when using the bottom-up method, we need to note m_f and m_u during the iteration, adding two additional trips to traverse the vertex list in effect. Second, as we can learn from previous experiments, the algorithm does not have a very large performance gain with 32 threads when there is no synchronization mechanism, giving less potential for improvement. So we do not enable bottom-up search in the final comparison with the serial version.

5.3 Performance Scalability

Through tuning experiments, an optimal set of parameters is obtained. This configuration is considered best in combination when the barrier is based on the condition variable, synchronization is based on lock, the number of threads is 32, the task size is 256, and

direction optimization is not enabled. After applying the optimal parameters, we compare the parallel operator with the serial version. There are two serial versions, user-defined function (UDF) based shortest path finding and operator-based shortest path finding, where the operator-based one replaces the PMS-BFS with MS-BFS. The scalability of the operator to the graph size and the workload (number of pairs) will be verified. In addition to performing shortest length finding, we also perform shortest path finding.

5.3.1 Graph Size Scalability

The result of the shortest length finding is shown in Figure 5.6. We keep the number of pairs constant and change the size of the graph to compare the performance. Regardless of the implementation of the search, the execution time grows linearly with increasing SF. It can be noticed that the sequential operator is much slower than the sequential UDF, considering that both implementations have the same kernel algorithm, then the sequential operator spends a lot of time on parts other than the search algorithm. The parallel operator and the sequential UDF have similar performance. As the number of pairs increases, the performance advantage of parallelism becomes more and more obvious.

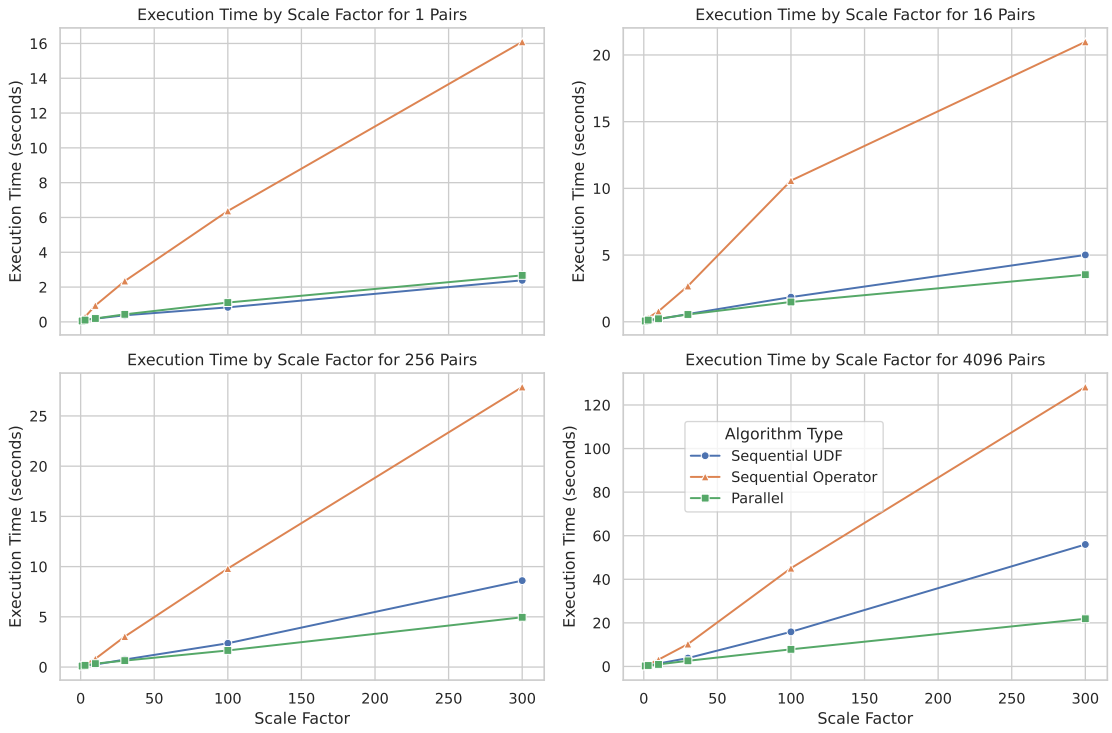

Figure 5.6: Execution time of serial and parallel shortest length operators for different graph sizes

Figure 5.7 shows the results of pathfinding. Unlike length finding, path-finding will have more work per thread. The only difference between path-finding and length-finding is the addition of logging the path, recalling the logging method mentioned in Section 4.1.3.2. Since the parent vertex of each vertex on lanes is not the same, it is necessary to record each lane individually, so it is not possible to utilize SIMD speedup. The time complexity of one iteration of the serial MS-BFS is $O(|V|+|E|)$, and the complexity rises to $O(|V|+|E|\cdot|S|)$ with the addition of the path logging, which is an increase of about $|S|$ times. The time complexity of parallel MS-BFS is similar. So we can see that the UDF and operator implementations of serial pathfinding have similar time consumption because the time consumed by the parts other than the algorithm is reduced. Also, the parallel operator can be significantly accelerated because we designed the path logging to be lock-free. The algorithm can be further optimized to reduce the memory footprint by using 32-bit values instead of 64-bit values to store parent vertices or edges. However, this may lead to overflow when processing super-large graphs. Currently, there are 3,709,057,850 edges in the SNB dataset when $SF = 10000$, and 32-bit values can represent 4,294,967,296 edges, which is close to the upper limit.

The parallelization speedup is ineffective when there is only one pair, clearly, the BFS search is taking too low a percentage of the total query execution time. When there are more pairs, the effect of parallelization speedup is very obvious.

5.3.2 Workload Scalability

We fix the size of the graph and vary the number of pairs to see how the performance of the serial and parallel programs changes when there are different amounts of work on a graph. Figure 5.8 shows the results of performing the shortest length finding. It can be seen that the serial UDF performs much better than the serial operator. As the size of the graph increases, the parallel operator becomes more effective. When SF is approximately equal to 5, it can be estimated that the parallel operator and the serial UDF have close performance.

Figure 5.9 shows the results of the shortest path finding. All three implementations of the find follow linear time growth. The sequential UDF and operator have very close execution times, since the core algorithm takes up the majority of the time, making the impact of the other parts smaller. The parallel operator has an overwhelming performance advantage over the other two implementations.

Finally, we summarize the speedup of parallel shortest length finding and shortest path finding compared to sequential UDF in Table 5.3. The number to the left of each cell of

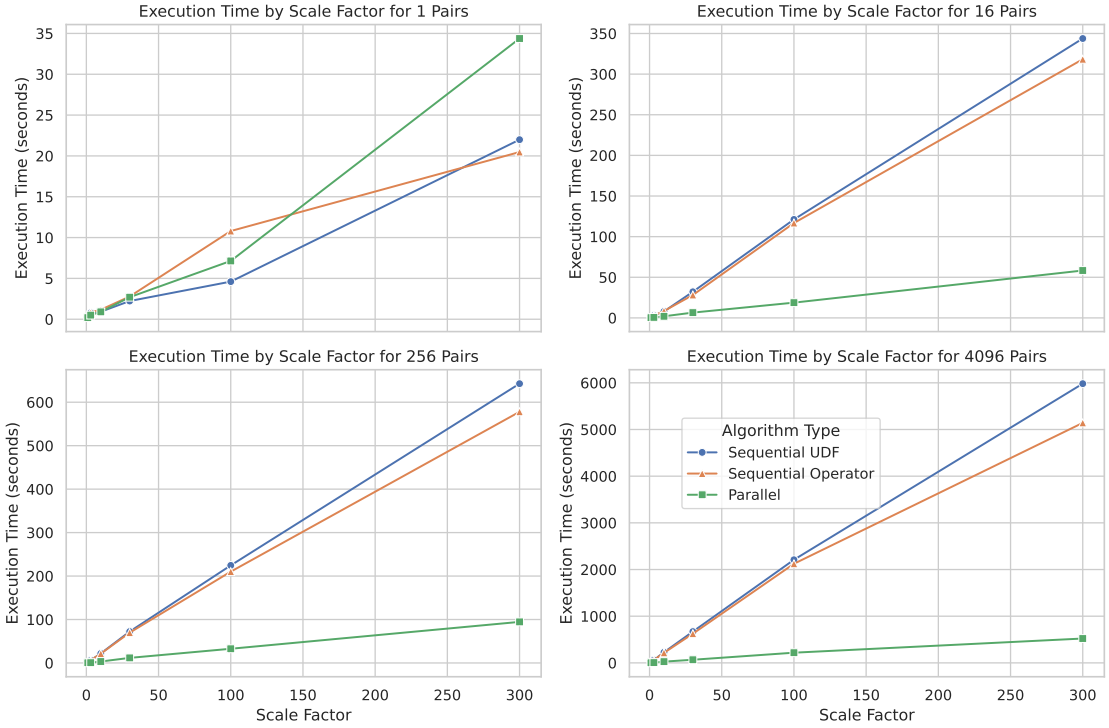

Figure 5.7: Execution time of serial and parallel shortest path operators for different graph sizes

data in the table is the shortest length speedup and the right is the shortest path speedup. The best speedup we achieved using 32 threads on the dataset involved in the experiment was 11.46.

5.4 Operator Component Costs

In our previous experiments, we mentioned that there are many other parts inside an operator besides the BFS. These parts consume the main execution time of the operator instead when the BFS workload is small. The main time-consuming parts of an operator are the following:

- *csr_vertex*: The creation of CSR vertices list.
- *csr_edge*: The creation of CSR edges list.
- *pairs_get*: Sink and combine source and destination vertex pairs.

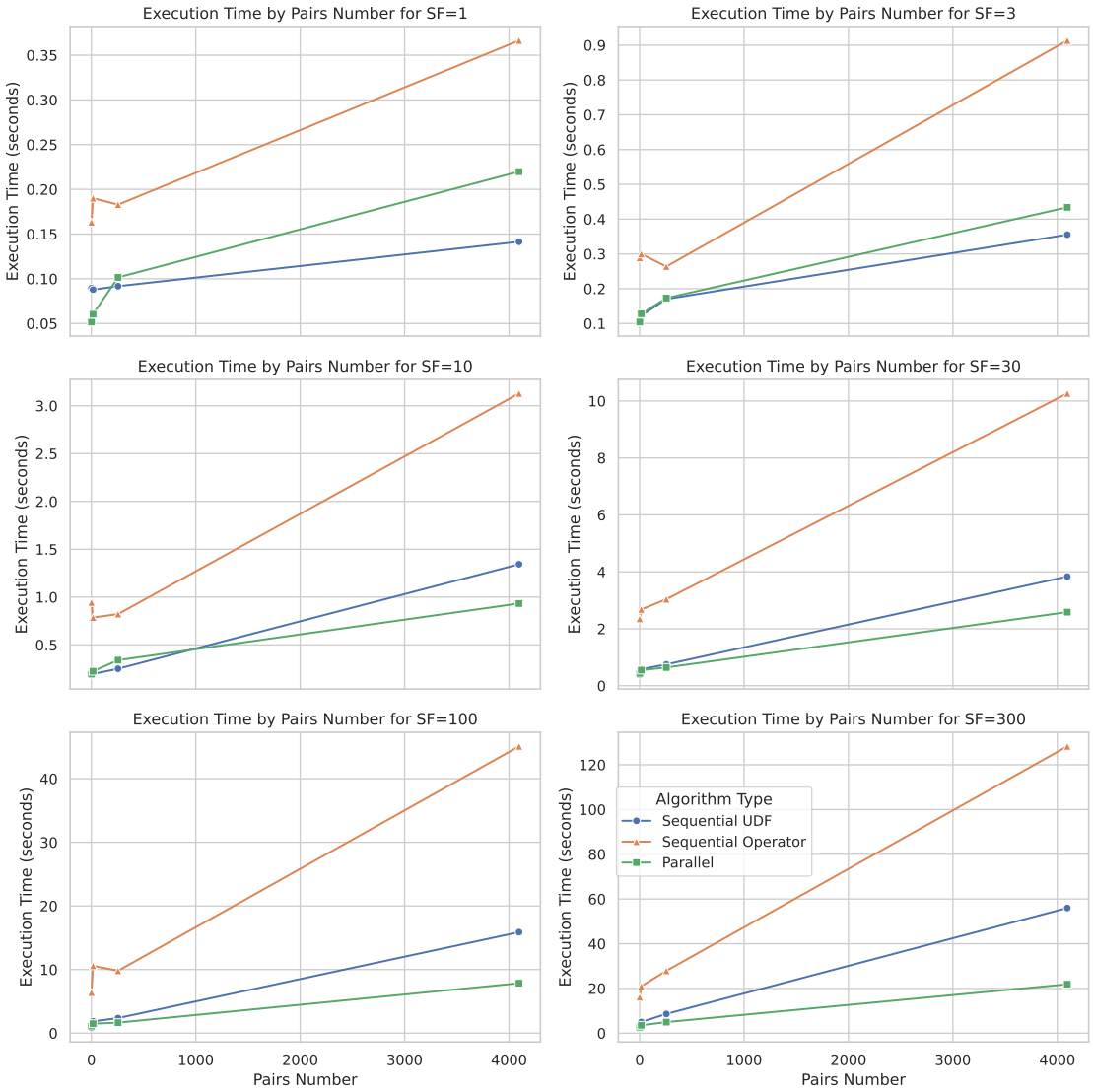

Figure 5.8: Execution time of serial and parallel shortest length operators for different pairs number

SF	$Pairs = 1$	$Pairs = 16$	$Pairs = 256$	$Pairs = 4096$
1	1.74 | 1.33	1.46 | 2.08	0.90 | 4.54	0.64 | 9.12
3	1.10 | 1.56	0.96 | 4.59	0.98 | 8.55	0.82 | 9.98
10	0.96 | 1.02	0.87 | 4.00	0.74 | 6.60	1.44 | 8.48
30	0.87 | 0.83	1.06 | 4.93	1.17 | 6.17	1.48 | 10.00
100	0.75 | 0.65	1.24 | 6.45	1.43 | 6.87	2.02 | 10.12
300	0.89 | 0.64	1.42 | 5.90	1.74 | 6.80	2.56 | 11.46

Table 5.3: Speedup between parallel operator and sequential UDF (length and path)

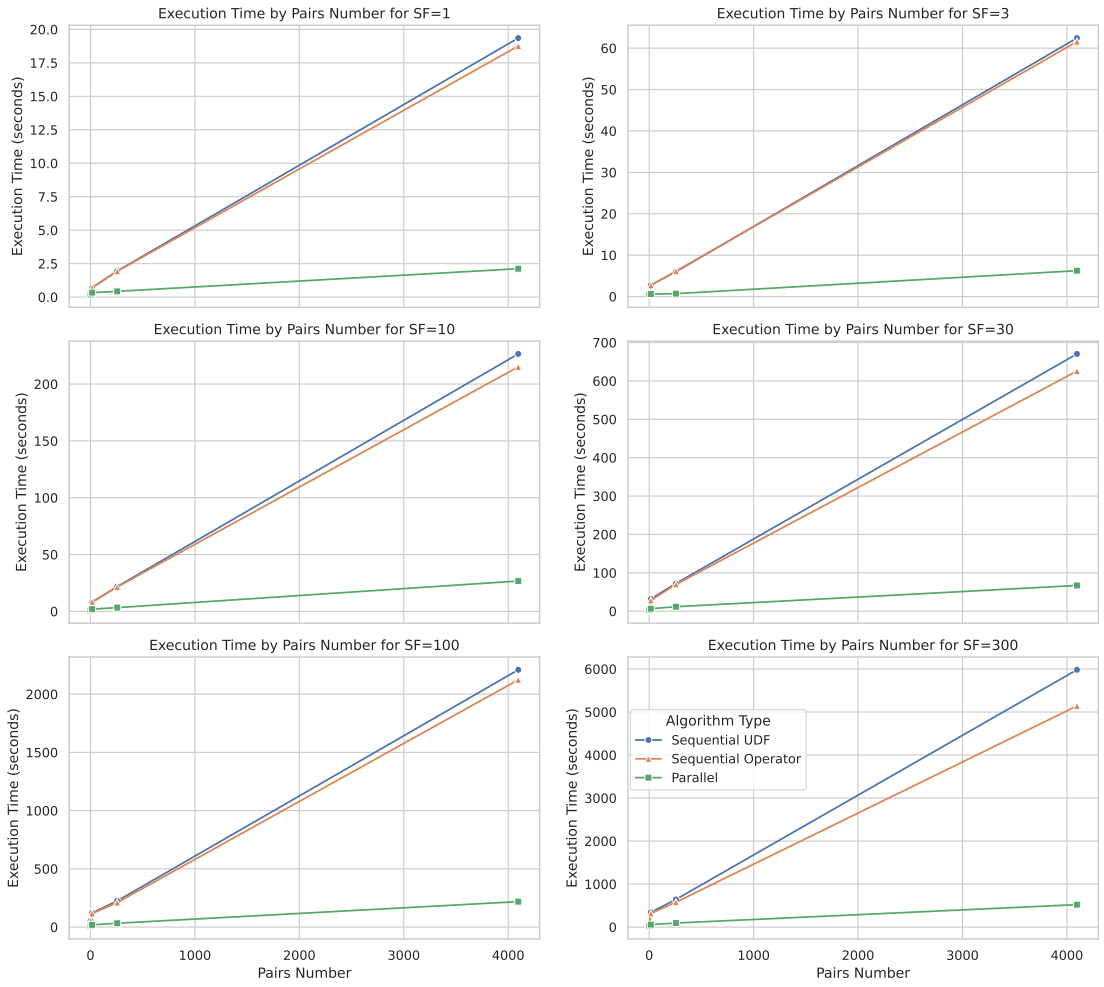

Figure 5.9: Execution time of serial and parallel shortest path operators for different pairs number

- *pairs_materialize*: Converting the data structure of pairs into a form that can be easily processed by algorithms.
- *state_init*: Initialize BFS runtime variables and parameters, e.g. the three main variables *seen*, *visit*, and *next*. When paths need to be logged, the parent vertex array also needs to be initialized. Initialization of variables consumes a lot of time when the amount of data is huge. Variables are initialized serially in the experiments, whether parallelism will speed up the process needs to be verified in the future.
- *event_init*: Extract a batch of pairs, initialize the BFS event and start it.
- *bfs*: The PMS-BFS algorithm.

- *results_get*: Push all results to upper-level operators.
- *others_in_operator*: Other code in the operator.
- *out_of_operator*: parts outside of the operator, such as scanning tables and outputting results.

Figure 5.10 and Figure 5.11 show the proportion of the total time consumed by each internal component of the shortest length finding when the pairs are 1 and 4096, respectively. It can be seen that when the number of pairs is 1, the BFS search occupies only a very small portion of the time, while the major portion of the time is spent on constructing the CSR. In addition, the parts outside the operator also consume time, but the percentage decreases as the graphs are expanded. When the number of pairs is 4096, the BFS search takes up the majority of the total time, followed by CSR construction, and finally, parts out of the operator are still visible as a noticeable percentage. Overall, the impact of the other components decreases significantly as the BFS workload increases. This will be more noticeable in the shortest path finding because BFS with path logging is very time-consuming.

5.4.1 Bottleneck Analyze

Ultimately, we found that the PMS-BFS-based shortest path operator could not achieve linear speedup. There are many sources of bottlenecks in a parallel program, we analyze the sources and importance of bottlenecks by:

- **Memory Bandwidth Limitation:** The rate at which data can be transferred between RAM and CPU caches is limited. As more threads simultaneously access memory, the bandwidth may become saturated, causing contention. According to the tool stream(105), the actual sequential copy rate of the experiment machine is 111.6 GB/s. When $SF = 300$, there are 1,230,500 vertices in the graph, each vertex needs 64 bits i.e. 8 bytes to store, so the size of the vertex list V of CSR is 9.844 MB. There are 68,313,982 edges in the graph, each edge also needs 8 bytes to store, so the size of the edge list E is about 547 MB. The length of the three important arrays, *seen*, *visit*, and *next*, is the same as the number of vertices, each element is 512 bits long, and the size of each array is 78.752 MB. Since we are using a pure bottom-up search, no locks are used in the process. In top-down search, the effect of locks needs to be taken into account, for our experiment configuration the amount of data read for locks in the case of top-down search is about 219 GB.

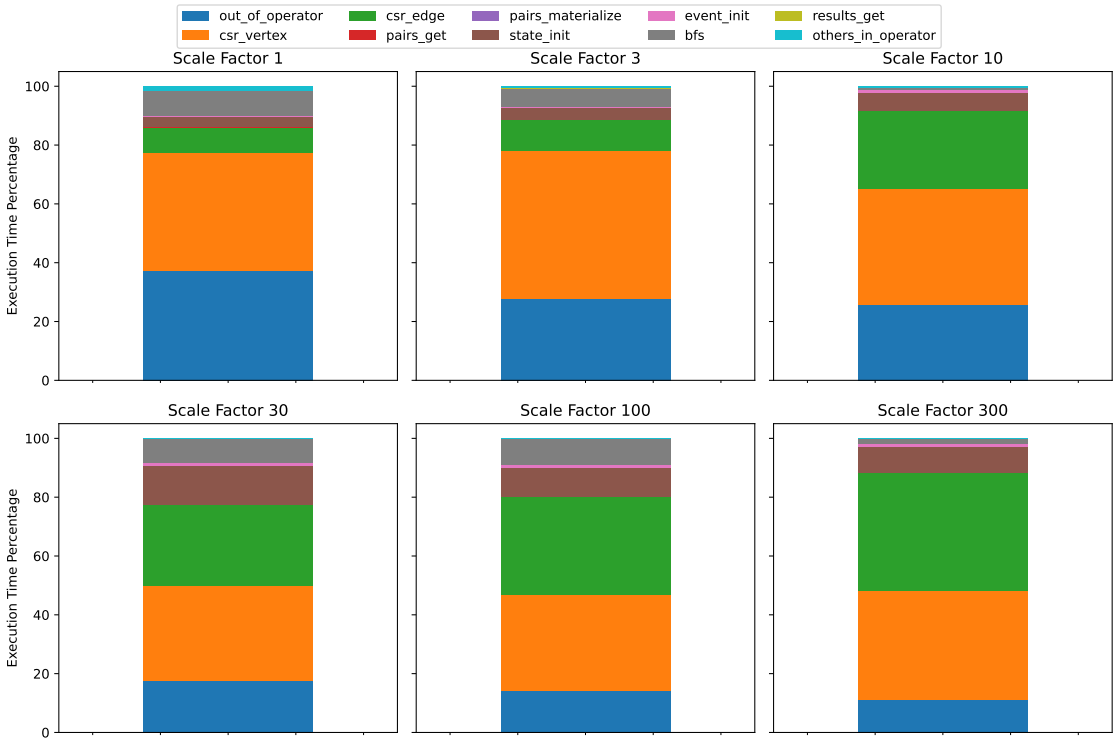

Figure 5.10: The time used for each part of the operator in the shortest length finding ($Pairs = 1$)

In the experiment shown in Figure 5.3 we know that a bottom-up search running 8 batches of 10 iterations each would take 12.376 seconds. Due to the nature of the multi-source bottom-up search, each iteration needs to traverse the complete V and E. Where each iteration V will be read once, E will be read once, *next* will be read twice, the elements of *visit* will be randomly read $|E|$ times, and *seen* will be read once. About 33.4GB of data needs to be transferred per second. According to these figures, without considering other effects, the algorithm's bandwidth demand does not exceed the physical upper limit. But obviously if the amount of data increases, the read and write requirements are likely to exceed the bandwidth.

- **Cache Contention:** Modern CPUs have multiple levels of cache (L1, L2, L3). If multiple threads access data that map to the same cache lines, it can cause cache contention and cache thrashing. Due to the large amount of data required for searching, it is likely to result in multiple threads accessing a cache line, but the cache line corresponds to a different memory. This will cause performance loss.

- **Cache Miss:** Fast CPU access to memory relies heavily on cache performance.

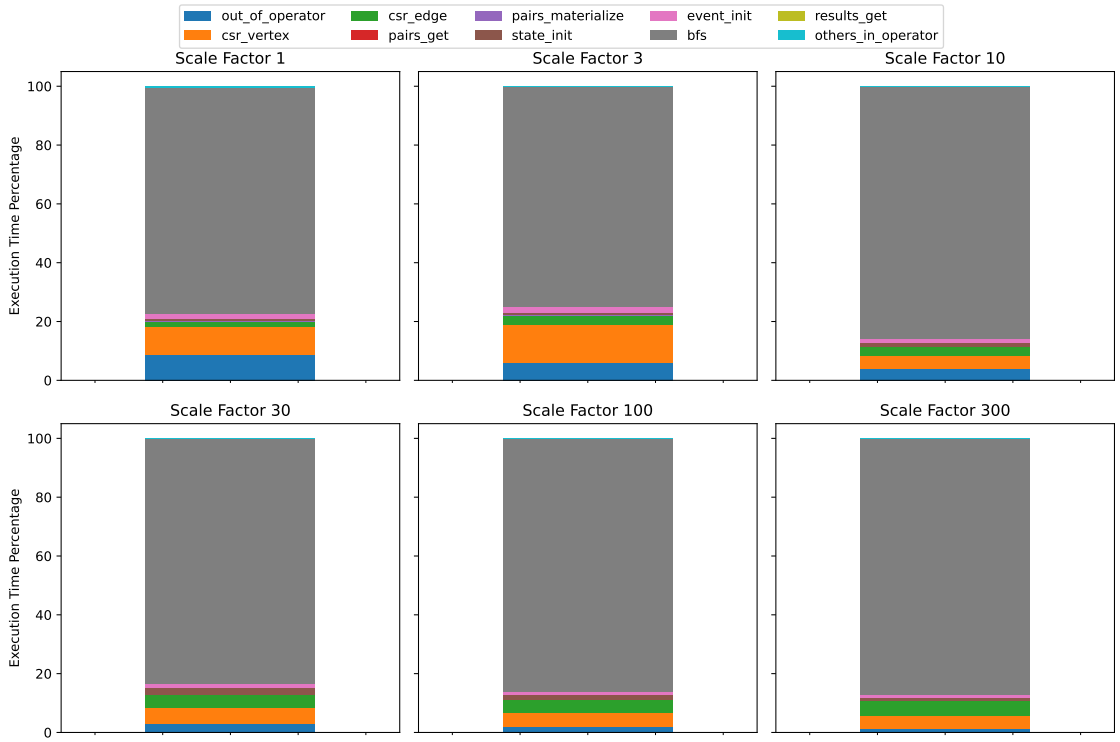

Figure 5.11: The time used for each part of the operator in the shortest length finding ($Pairs = 4096$)

According to Table 2.1, modern CPUs typically access cache and memory at speeds of 0.4167 - 1.25 nanoseconds for L1 cache, 4.167 - 8.333 nanoseconds for L2 cache, about 8.333 - 20.835 nanoseconds for L3 cache, and about 41.67+ nanoseconds for DRAM memory. In our bottom-up search, accesses to the *visit* array, which has the largest amount of data, are random and the size of each element is 64 bytes, which is equivalent to the size of a cache line that a Xeon processor typically has. The L3 cache of the experiment machine is 13.75MiB, which is much smaller than the amount of data the program runs. This means that each access to *visit* is likely to result in a cache miss and the number of data transfers per second drops from the theoretical 10.4 GT/s to 24 MT/s. All 32 cores are reading data at the same time, and each cache miss results in the transfer of 64 bytes of data at a rate of 24 MT/s per core. it can be calculated that the theoretical random read rate that can be provided by 32 cores is 49.2 GB/s, which is very close to the actual speed derived from our previous calculations. Besides memory access time, there are three levels of cache misses time, Translation Lookaside Buffer (TLB) misses time, etc., so the actual rate may be lower. Although TLB misses can be avoided by increasing the

page size, the default page size was not modified in our experiments. With a size of 4KB per page, the *visit* array requires 19,688 pages to store, so clearly TLB misses cannot be ignored. In conclusion, the cache miss due to random access becomes a performance bottleneck.

- **Synchronization Overhead:** Synchronization mechanisms like locks, barriers, and atomic operations ensure correct access to shared resources. However, they introduce overhead and can become a bottleneck.

- **Load Imbalance:** If the work is not evenly distributed among threads, some threads may finish earlier and remain idle while others are still processing. We mitigate the imbalance through dynamic task creation and small-sized tasks.

- **Thread Management Overhead:** Creating, scheduling, and managing threads incur overhead. Frequent context switches between threads can degrade performance. In the experiments, the number of threads used did not exceed the number available on the system, so scheduling due to too many threads had no effect. However, there is a large number of barriers in the algorithm. We learned from the experiment that each barrier of 32 threads will consume 121 milliseconds. When the number of threads is fixed, the consumption of the algorithm on barriers only grows with the number of iterations. Considering that for the SNB dataset, the number of iterations is usually less than 10, the overhead caused by the barrier only accounts for a very small portion. Also based on previous experiments with barriers, removing the barriers instead causes the algorithm to be difficult to stop and consumes more time.

- **NUMA Effects:** In Non-Uniform Memory Access (NUMA) systems, the latency, and bandwidth to memory can vary depending on the memory location relative to the processor accessing it. The experiment machine has two NUMA nodes that access the same memory. Considering that the machine's memory is very large, it should not be local to any of the NUMA nodes. So the overhead does exist. As NUMA nodes result in slower access to memory and synchronization between cores, it will exacerbate bottlenecks caused by cache misses and synchronization.

- **Amdahl's Law:** Amdahl's Law states that the speedup of a parallel program is limited by the serial portion of the program. Based on the operator component cost experiments, it can be seen that the serial part severely pulls down the overall performance when the amount of data is not large.

6

Future Work

In this chapter, we discuss what other optimizations can be made in the future based on the work already stated.

6.1 Parallel Bounded Path-finding

In Section 4.1.2 we designed the shortest path-finding algorithm with upper and lower bounds, which has many optimization directions in the future, including parallelization, path logging optimization, and so on.

The algorithm has two parts, the first part performs MS-BFS but does not exclude already visited vertices from the next, and the second part performs normal MS-BFS. Both parts can directly be parallelized using the parallelism that we designed in our thesis. Each iteration of the first part will use one less barrier, but the frontier will grow faster, resulting in more computation.

The most important reason for the degradation of the performance of the algorithm is the low performance of the path logging. Unlike normal MS-BFS, in the first part of the algorithm, we need to use the ternary (i, v, l) instead of (v, l) to locate the successor of a vertex, where i refers to the iteration, v refers to the vertex, and l refers to the lane. Because vertices may be visited repeatedly on different iterations during the search process, each visit has a different successor. These successor relations cannot be randomly overwritten like normal MS-BFS, otherwise it will lead to errors in the path chain. But this means that we need to use three-dimensional arrays to store the information about the successors, which will greatly increase the memory requirement. Developing a high-performance and low-consumption path logging method is an important direction for the future.

6.2 SIMD Effects

In our design and experiments, we have always set the number of lanes to 512, which corresponds to the maximum number of SIMD register bits available in the AVX-512 instruction set. One piece of unfinished work is the effect of different lanes on performance. Recall that we proposed the segmented bitset, a data structure that does not use SIMD instructions at the instruction level. We have experimentally demonstrated that this approach does not perform particularly poorly. So it is necessary to study how much SIMD actually affects the performance. In addition, in the bottleneck analysis subsection, we point out that each read of a bitset of 512 lanes is likely to result in a cache miss. So it makes sense to analyze whether fewer lanes add unexpectedly to performance. Finally, different CPUs support different SIMD instruction widths; Intel's latest products have widely used the AVX-512 instruction set, but the NEON technology (24) currently used by ARM only supports 128-bit SIMDs. so investigating the performance of different lane widths is necessary in the future.

6.3 Workload Elimination

In addition to parallelizing the algorithm to speed up pathfinding, we can also reduce the actual workload of the algorithm. This idea comes from the frequent behavior of databases in real scenarios. When a user performs a shortest path finding, it is very common to have a query that starts at one vertex and goes to multiple vertices, or vice versa. This kind of query can be efficiently optimized as a single-source shortest path search since BFS is essentially a one-to-many search algorithm.

The key difficulty in achieving this optimization is how to efficiently identify from the input pairs that can be optimized, i.e., how to efficiently remove duplicate elements from the array. The basic idea is to use hash tables for de-duplication. The offset of the elements of the hash table is determined by the source vertices, and each element is a list containing the destination vertices, thus realizing the one-to-many relationship. Since the hash table needs to be queried after each iteration to determine the completed search, the high performance of the hash table is critical.

6.4 Advanced Parallelism

The parallelization proposed in the thesis still has room for optimization. We added several barriers to each iteration of PMS-BFS to avoid race conditions. However, the process can

be optimized to reduce the use of barriers. One idea is to implement asynchronous PMS-BFS. the last barrier of each iteration ensures that the *change* can affect all threads. With the correct result, we can remove this barrier, allowing some threads to perform the reset to *next* without knowing if they can globally move on to the next iteration. In the bottom-up approach, the barrier between the reset *next* and the neighbor exploration can be deleted because the elements of *next* that the threads need to access are determined, but this requires that the *next* reset phase also use dynamic task creation.

In addition, given that the efficiency of speedup grows slower as the number of threads increases, we can run two or more instances of PMS-BFS at the same time. For example, running two PMS-BFS on a 40-core machine with 20 cores per algorithm. In this way, the efficiency of each PMS-BFS in using resources rises, and the global speedup performance may be better.

6.5 Distributed Parallelism

Although DuckDB is designed as a DBMS for standalone machines, it has the potential to be extended to distributed databases, e.g., MotherDuck (106). In addition, machines under the NUMA architecture can be considered as distributed systems with less communication overhead. So the study of shortest path-finding under distributed systems is also necessary in the future. Several related works (99) have explored the computation of BFS under distribution and the core idea is to reduce the communication overhead between threads. Each thread should compute on local memory as much as possible. Due to the randomness of top-down search, this approach is not applicable to distributed computation. Instead, the randomness of the bottom-up search comes from read-only access to the *visit*, which can simply be exchanged across threads before each iteration. Since distributed computation requires exactness in the amount of computation, work-stealing cannot be applied anymore to avoid threads computing on the memory of other threads.

6.6 Lock Free Algorithm

Top-down search requires locks to help synchronize, and in previous experiments, we have shown that lock-free algorithms can improve performance by up to 2x. Before developing lock-free algorithms, algorithms that use fewer locks can be explored. Currently, we lock each neighbor vertex every time we explore the neighbor. However, not every exploration leads to contention. Larger granularity of locks can be used to optimize performance.

There are two technical approaches to implement lock-free. Firstly error recovery can be implemented. In the shortest path finding MS-BFS, we have implemented lock-free path logging. The parent vertex array practically duplicates the information contained in the *next* array, i.e., which vertices will appear on the frontiers of the next iteration. In this way, after each iteration, we can correct the *next* array using the information from the parent vertex array. The second technical approach is inspired by distributed BFS, where updates to elements in the *next* array can be modified to occur only in the worker that owns it. This approach avoids the possibility of multiple workers modifying the same element but requires additional space to store modification requests from other workers. And more communication is required between workers.

6.7 GPU Optimization

GPUs can provide more cores and larger bandwidths, making them ideal for parallel computing. However, performing computations containing branches in GPUs is expensive. Analyzing top-down and bottom-up search separately we can conclude that bottom-up search is more suitable for GPU parallelization. Firstly it does not require synchronization while exploring neighbors and secondly, the data that needs to be updated by each worker is determined for each iteration making it very easy for task allocation. However, GPU parallelization introduces new problems, firstly SIMD instructions on GPUs are obviously different from CPUs, which involves program refactoring, introducing additional complexity and reducing maintainability. In addition, the current parallelized MS-BFS contains a large number of barriers whereas global synchronization on GPUs is expensive.

7

Conclusion

In this thesis, we successfully implement PMS-BFS, a shortest path finding algorithm with custom parallelism integrated into operators, replacing the existing UDF-based finding that relies on morsel-driven parallelism. During the design process, we also considered different implementation techniques and compared their performance. In addition, we tried direction optimization and proved that it has limited optimization for multi-source BFS, but has potential for distributed computing. Finally, we proved that the new shortest path finding operator has better performance than UDF through experiments.

Following are answers to the research questions defined for this thesis work:

How best to refactor path-finding in DuckPQG? Instead of UDF, we use an operator-based parallel shortest path finding algorithm, where the existing event and task mechanisms of DuckDB can be leveraged for custom parallelization. Inside the operator, we can control the full flow of function execution, including the construction of the CSR, the collection of vertex pairs, and performing the BFS search. In addition, operators have a uniform interface, which facilitates consistency in the development style of DuckDB. Finally, operator-based methods have better optimization potential and can be incorporated into the optimization mechanisms of DuckDB.

How to correctly parallelize the MS-BFS algorithm? A requirement for proper parallelization is the elimination of race conditions. Since race conditions between iterations are difficult to eliminate, parallelization occurs within iterations. Race conditions between individual loops within an iteration are eliminated by barriers, and race conditions due to random neighbor exploration are eliminated by mutual exclusion. These measures are introduced at the cost of algorithmic performance degradation.

What techniques can be used to optimize parallelized MS-BFS to improve speed? First, the barriers and synchronization techniques used by the algorithm in parallelization can be selected as optimal. We compared three techniques for barriers, C++ condition variable, spin lock, and Pthread library, and experiments proved that the condition variable is the best while maintaining compatibility. There are also three techniques for synchronization, which are atomized bitset, segmented atomized bitset, and locks. Experiments prove that locks have the best performance. Secondly, we designed dynamic task creation and work stealing to ensure that each thread has the same amount of work. Finally, we also explored direction optimization but were experimentally shown to have limited optimization.

How to embed parallelized MS-BFS into the path-finding operator instead of UDF? We utilize the generic operator interface of DuckDB. The shortest path operator has two Sinks for materializing the CSR and vertex pairs. One of them, CSR, can construct vertex lists in parallel when Sinking. After both are fully materialized, we call the parallelized MS-BFS in the Finalize interface of the operator, and we need to complete the construction of the edges of the CSR and the initialization of the runtime environment parameters of the algorithm before the algorithm is called. The execution of a batch of vertex pairs is wrapped in a DuckDB event, which can define any task containing the logic of the algorithm, corresponding to a thread. The scheduling of threads is hidden by DuckDB. After all batches are executed, the results are pushed upwards by the GetData interface of the operator.

What is the performance of parallelized MS-BFS? We show through experiments that the speedup cannot be linear, that it grows slower as the number of threads increases, and that ultimately there is a lower bound on the execution time. Acceleration is most efficient at 2 to 4 threads, after which speedup increases less. In addition, it is demonstrated by experiments that the size of the task block processed by each thread does not have a significant effect on performance. Finally, we find that for the shortest-length finding without path logging, the UDF implementation already performs well enough on small graphs and even outperforms the parallel algorithm. However, when the graph is very large, the parallel algorithm still has a performance advantage. For shortest path finding, the parallel algorithm has a considerable speedup, which can reach 11.46 in the case of $SF = 300$. The larger the graph size the more obvious the performance advantage of the parallel algorithm.

What are the bottlenecks of parallelized MS-BFS? There are many sources of bottlenecks in the parallel algorithm PMS-BFS. First, the introduction of synchronization clearly slows down the speed, especially on large graphs. While barriers should theoretically reduce performance, experiments have shown that the algorithm is difficult to stop without them. Second, the bottleneck comes from the communication between the CPU and memory. Although the experiments we conducted with the largest amount of data did not reach the theoretical memory bandwidth, it is clear that the bandwidth will be insufficient with larger amounts of data. Finally, by computation, we believe that cache misses lead to performance degradation.

References

[1] I. Robinson, J. Webber, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, 2013. 1

[2] Jonathan L. Gross, Jay Yellen, Lowell W. Beineke, and Robin J. Wilson. **Introduction to Graphs**. In Jonathan L. Gross and Jay Yellen, editors, *Handbook of Graph Theory*, Discrete Mathematics and Its Applications, pages 1–55. Chapman & Hall / Taylor & Francis, 2003. 1

[3] Shalini Batra and Charu Tyagi. **Comparative analysis of relational and graph databases**. *International Journal of Soft Computing and Engineering (IJSCE)*, **2**(2):509–512, 2012. 1

[4] Neo4j. **Neo4j - The World's Leading Graph Database**, 2012. 1

[5] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. **KÙZU Graph Database Management System**. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org, 2023. 1

[6] Daniel ten Wolde, Gábor Szárnyas, and Peter A. Boncz. **DuckPGQ: Bringing SQL/PGQ to DuckDB**. *Proc. VLDB Endow.*, **16**(12):4034–4037, 2023. 1, 13

[7] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1981–1984. ACM, 2019. 1, 9

[8] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. **Graph Pattern Matching in GQL and SQL/PGQ**. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 2246–2258. ACM, 2022. 1, 38

[9] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. **Foundations of Modern Query Languages for Graph Databases**. *ACM Comput. Surv.*, **50**(5):68:1–68:40, 2017. 1

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. 1

[11] Moritz Kaufmann, Manuel Then, Alfons Kemper, and Thomas Neumann. **Parallel Array-Based Single- and Multi-Source Breadth First Searches on Large Dense Graphs**. In Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Bress, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 1–12. OpenProceedings.org, 2017. 1, 28, 57, 66

[12] Yannis Foufoulas and Alkis Simitsis. **User-Defined Functions in Modern Data Engines**. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 3593–3598. IEEE, 2023. 2, 12

[13] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. **Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age**. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014. 2, 12

[14] Pawan Harish and P. J. Narayanan. **Accelerating Large Graph Algorithms on the GPU Using CUDA**. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing - HiPC 2007, 14th International Conference, Goa, India,*

December 18-21, 2007, Proceedings, **4873** of *Lecture Notes in Computer Science*, pages 197–208. Springer, 2007. 2, 27, 29

[15] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. **Efficient Parallel Graph Exploration on Multi-Core CPU and GPU**. In Lawrence Rauchwerger and Vivek Sarkar, editors, *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, pages 78–88. IEEE Computer Society, 2011. 2, 27

[16] Songning Lai, Jiakang Li, and Yonggang Lu. **A Comprehensive Review of Community Detection in Graphs**. *CoRR*, **abs/2309.11798**, 2023. 2

[17] Esti Yeger-Lotem, Laura Riva, Linhui Julie Su, Aaron D Gitler, Anil G Cashikar, Oliver D King, Pavan K Auluck, Melissa L Geddie, Julie S Valastyan, David R Karger, et al. **Bridging high-throughput genetic and transcriptional data reveals cellular responses to alpha-synuclein toxicity**. *Nature genetics*, **41**(3):316–323, 2009. 2

[18] Xuesong Zhou, Lu Tong, Monirehalsadat Mahmoudi, Lijuan Zhuge, Yu Yao, Yongxiang Zhang, Pan Shang, Jiangtao Liu, and Tie Shi. **Open-source VRPLite package for vehicle routing with pickup and delivery: a path finding engine for scheduled transportation systems**. *Urban Rail Transit*, **4**:68–85, 2018. 2

[19] Khee Meng Koh, Fengming Dong, and Eng Guan Tay. *Introduction to Graph Theory - With Solutions to Selected Problems*. WorldScientific, 2024. 5

[20] DuckDB. **Overview of duckdb internals**, Jul 2024. 10

[21] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. **The More the Merrier: Efficient Multi-Source Graph Traversal**. *Proc. VLDB Endow.*, **8**(4):449–460, 2014. 13, 41

[22] Pawel Gepner. **Using AVX2 instruction set to increase performance of high performance computing code**. *Computing and Informatics*, **36**(5):1001–1018, 2017. 15

[23] Ray Kinsella, Chris MacNamara, and Georgii Tkachuk. **Intel® AVX-512 - Writing Packet Processing Software with Intel® AVX-512 Instruction Set**. https://networkbuilders.intel.com/docs/networkbuilders/intel-avx-512-writing-packet-processing-software-with-intel-avx-512-instruction-set-t pdf. 15

[24] Venu Gopal Reddy. **Neon technology introduction**. *ARM Corporation*, **4**(1):1–33, 2008. 15, 92

[25] Jiri Barnat, Lubos Brim, and Jakub Chaloupka. **Parallel Breadth-First Search LTL Model-Checking**. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, pages 106–115. IEEE Computer Society, 2003. 22

[26] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. **Efficient Breadth-First Search on the Cell/BE Processor**. *IEEE Trans. Parallel Distributed Syst.*, **19**(10):1381–1395, 2008. 22, 23

[27] Yuichiro Yasui, Katsuki Fujisawa, and Kazushige Goto. **NUMA-optimized parallel breadth-first search on multicore single-node system**. In Xiaohua Hu, Tsau Young Lin, Vijay V. Raghavan, Benjamin W. Wah, Ricardo Baeza-Yates, Geoffrey C. Fox, Cyrus Shahabi, Matthew Smith, Qiang Yang, Rayid Ghani, Wei Fan, Ronny Lempel, and Raghunath Nambiar, editors, *2013 IEEE International Conference on Big Data (IEEE BigData 2013), 6-9 October 2013, Santa Clara, CA, USA*, pages 394–402. IEEE Computer Society, 2013. 23, 24, 25

[28] Chenglong Zhang, Huawei Cao, Xiaochun Ye, Guobo Wang, Qinfen Hao, and Dongrui Fan. **Highly Efficient Breadth-First Search on CPU-Based Single-Node System**. In Zheng Xiao, Laurence T. Yang, Pavan Balaji, Tao Li, Keqin Li, and Albert Y. Zomaya, editors, *21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10-12, 2019*, pages 2066–2071. IEEE, 2019. 23, 25

[29] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. **Ordered and unordered algorithms for parallel breadth first search**. In

Valentina Salapura, Michael Gschwind, and Jens Knoop, editors, *19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*, pages 539–540. ACM, 2010. 24

[30] Andy Yoo, Edmond Chow, Keith W. Henderson, Will McLendon III, Bruce Hendrickson, and Ümit V. Çatalyürek. **A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L**. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*, page 25. IEEE Computer Society, 2005. 24, 25

[31] Aydin Buluç and Kamesh Madduri. **Parallel breadth-first search on distributed memory systems**. In Scott A. Lathrop, Jim Costa, and William Kramer, editors, *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 65:1–65:12. ACM, 2011. 24

[32] Yuichiro Yasui and Katsuki Fujisawa. **Fast and scalable NUMA-based thread parallel breadth-first search**. In *2015 International Conference on High Performance Computing & Simulation, HPCS 2015, Amsterdam, Netherlands, July 20-24, 2015*, pages 377–385. IEEE, 2015. 24

[33] Huiwei Lv, Guangming Tan, Mingyu Chen, and Ninghui Sun. **Compression and Sieve: Reducing Communication in Parallel Breadth First Search on Distributed Memory Systems**. *CoRR*, **abs/1208.5542**, 2012. 24

[34] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. **Efficient Breadth-First Search on Massively Parallel and Distributed-Memory Machines**. *Data Sci. Eng.*, **2**(1):22–35, 2017. 24

[35] Scott Beamer, Krste Asanović, and David A. Patterson. **Searching for a Parent Instead of Fighting Over Children : A Fast Breadth-First Search Implementation for Graph 500**. 2011. 25

[36] Scott Beamer, Krste Asanovic, and David A. Patterson. **Direction-optimizing breadth-first search**. *Sci. Program.*, **21**(3-4):137–148, 2013. 26, 66

[37] Osama G. Attia, Tyler Johnson, Kevin Townsend, Phillip H. Jones, and Joseph Zambreno. **CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search**. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 228–235. IEEE Computer Society, 2014. 25

[38] Jesper Larsson Träff. **A Note on (Parallel) Depth- and Breadth-First Search by Arc Elimination**. *CoRR*, **abs/1305.1222**, 2013. 25

[39] Tom St. John, Jack B. Dennis, and Guang R. Gao. **Massively parallel breadth first search using a tree-structured memory model**. In Minyi Guo and Zhiyi Huang, editors, *Proceedings of the 2012 PPOPP International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2012, New Orleans, LA, USA, February 26, 2012*, pages 115–123. ACM, 2012. 25

[40] Yuan Zhang, Huawei Cao, Yan Liang, Jie Zhang, Junying Huang, Xiaochun Ye, and Xuejun An. **FSGraph: fast and scalable implementation of graph traversal on GPUs**. *CCF Trans. High Perform. Comput.*, **5**(3):277–291, 2023. 27

[41] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. **A Parallelization of Dijkstra's Shortest Path Algorithm**. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic, August 24-28, 1998, Proceedings*, **1450** of *Lecture Notes in Computer Science*, pages 722–731. Springer, 1998. 29

[42] Ulrich Meyer and Peter Sanders. **[Delta]-stepping: a parallelizable shortest path algorithm**. *J. Algorithms*, **49**(1):114–152, 2003. 29

[43] Zbigniew J. Czech. *Introduction to Parallel Computing*. Cambridge University Press, 2017. 29

[44] Alexandre Tiskin. **All-Pairs Shortest Paths Computation in the BSP Model**. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*, **2076** of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2001. 29

[45] David A. Bader, Guojing Cong, and John Feo. **On the Architectural Requirements for Efficient Execution of Graph Algorithms**. In *34th International Conference on Parallel Processing (ICPP 2005), 14-17 June 2005, Oslo, Norway*, pages 547–556. IEEE Computer Society, 2005. 30

[46] Federico Busato and Nicola Bombieri. **An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures**. *IEEE Trans. Parallel Distributed Syst.*, **27**(8):2222–2233, 2016. 30

[47] Akihiro Kishimoto, Alex S. Fukunaga, and Adi Botea. **Scalable, Parallel Best-First Search for Optimal Sequential Planning**. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI, 2009. 30

[48] Nathan R. Sturtevant and Michael Buro. **Partial Pathfinding Using Map Abstraction and Refinement**. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1392–1397. AAAI Press / The MIT Press, 2005. 30

[49] Sergiy Pogorilyy and Maxim Slynko. **Research and development of Johnson's algorithm parallel schemes in GPGPU technology**. In Ivan Sergienko and Philip Andon, editors, *Proceedings of the 10th International Conference of Programming UkrPROG'2016, Kyiv, Ukraine, May 24-25, 2016*, **1631** of *CEUR Workshop Proceedings*, pages 105–112. CEUR-WS.org, 2016. 30

[50] Avadhesh Singh and Dhirendra Singh. **Implementation of K-shortest path algorithm in GPU using CUDA**. *Procedia Computer Science*, **48**:5–13, 12 2015. 31

[51] Goetz Graefe. **Volcano - An Extensible and Parallel Query Evaluation System**. *IEEE Trans. Knowl. Data Eng.*, **6**(1):120–135, 1994. 32

[52] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. **Column-stores vs. row-stores: how different are they really?** In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data,*

SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, pages 967–980. ACM, 2008. 32

[53] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. **Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine**. In Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan, editors, *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, pages 5–17. ACM, 2024. 32

[54] Michael Stonebraker. **The Case for Shared Nothing**. *IEEE Database Eng. Bull.*, **9**(1):4–9, 1986. 32

[55] Sérgio Fernandes and Jorge Bernardino. **What is BigQuery?** In *Proceedings of the 19th International Database Engineering & Applications Symposium*, IDEAS '15, page 202–203, New York, NY, USA, 2015. Association for Computing Machinery. 32

[56] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. **Amazon Redshift and the Case for Simpler Data Warehouses**. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1917–1923. ACM, 2015. 32

[57] Peter A. Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution**. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005. 32

[58] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. **Performance Tradeoffs in Read-Optimized Databases**. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 487–498. ACM, 2006. 32

[59] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. **Dryad: distributed data-parallel programs from sequential building blocks**. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 59–72. ACM, 2007. 32

[60] Fabian Hueske and Timo Walther. **Apache Flink**. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019. 33

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing**. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. 33

[62] Regina Obe and Leo Hsu. *PostgreSQL - Up and Running: a Practical Guide to the Advanced Open Source Database*. O'Reilly, 2012. 33

[63] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I. August. **Perspective: A Sensible Approach to Speculative Automatic Parallelization**. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 351–367. ACM, 2020. 33

[64] Alexander Baumstark and Constantin Pohl. **Lock-free Data Structures for Data Stream Processing - A Closer Look**. *Datenbank-Spektrum*, **19**(3):209–218, 2019. 33

[65] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and

Dale Woodford. **Spanner: Google's Globally Distributed Database**. *ACM Trans. Comput. Syst.*, **31**(3):8, 2013. 33

[66] Victor Giannakouris. **Building Learned Federated Query Optimizers**. In Zhifeng Bao and Timos K. Sellis, editors, *Proceedings of the VLDB 2022 PhD Workshop co-located with the 48th International Conference on Very Large Databases (VLDB 2022), Sydney, Australia, September 5, 2022*, **3186** of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022. 34

[67] Adrian Daniel Popescu, Debabrata Dash, Verena Kantere, and Anastasia Ailamaki. **Adaptive query execution for data management in the cloud**. In Xiaofeng Meng, Ying Chen, Jianliang Xu, and Jiaheng Lu, editors, *Proceedings of the Second International CIKM Workshop on Cloud Data Management, CloudDB 2010, Toronto, Ontario, Canada, October 30, 2010*, pages 17–24. ACM, 2010. 34

[68] **Performance Tuning - Spark 3.5.1 Documentation — spark.apache.org**. `https://spark.apache.org/docs/latest/sql-performance-tuning.html`. [Accessed 15-07-2024]. 34

[69] Junyi Zhao, Huanchen Zhang, and Yihan Gao. **Efficient Query Re-optimization with Judicious Subquery Selections**. *Proc. ACM Manag. Data*, **1**(2):185:1–185:26, 2023. 34

[70] **Presto: Free, Open-Source SQL Query Engine for any Data — prestodb.io**. `https://prestodb.io/`. [Accessed 15-07-2024]. 34

[71] **Apache Drill - Schema-free SQL for Hadoop, NoSQL and Cloud Storage — drill.apache.org**. `https://drill.apache.org/`. [Accessed 15-07-2024]. 34

[72] Amirhossein Aleyasen, Mohamed A. Soliman, Lyublena Antova, F. Michael Waas, and Marianne Winslett. **High-Throughput Adaptive Data Virtualization via Context-Aware Query Routing**. In Naoki Abe, Huan Liu, Calton Pu, Xiaohua Hu, Nesreen K. Ahmed, Mu Qiao, Yang Song, Donald Kossmann, Bing Liu, Kisung Lee, Jiliang Tang, Jingrui He, and Jeffrey S. Saltz, editors, *IEEE International Conference on Big Data (IEEE BigData 2018), Seattle, WA, USA, December 10-13, 2018*, pages 1709–1718. IEEE, 2018. 34

[73] Damla Oguz, Belgin Ergenc, Shaoyi Yin, Oguz Dikenelli, and Abdelkader Hameurlain. **Federated query processing on linked data: a qualitative survey and open challenges**. *Knowl. Eng. Rev.*, **30**(5):545–563, 2015. 34

[74] **DB-Engines Ranking — db-engines.com**. `https://db-engines.com/en/ranking/graph+dbms`. [Accessed 16-07-2024]. 35

[75] **Neo4j Graph Database & Analytics – The Leader in Graph Databases — neo4j.com**. `https://neo4j.com/`. [Accessed 16-07-2024]. 35

[76] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. **Cypher: An Evolving Query Language for Property Graphs**. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018. 35

[77] **Memgraph — memgraph.com**. `https://memgraph.com/`. [Accessed 16-07-2024]. 35

[78] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. **Nebula Graph: An open source distributed graph database**. *CoRR*, **abs/2206.07278**, 2022. 36

[79] Alastair Green, Martin Junghanns, Max Kiessling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. **openCypher: New Directions in Property Graph Querying**. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 520–523. OpenProceedings.org, 2018. 36

[80] **GitHub - vesoft-inc/nebula-algorithm: Nebula-Algorithm is a Spark Application based on GraphX, which enables state of art Graph Algorithms to run on top of NebulaGraph and write back results to NebulaGraph. — github.com**. `https://github.com/vesoft-inc/nebula-algorithm`. [Accessed 16-07-2024]. 37

[81] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. **TigerGraph: A Native MPP Graph Database**. *CoRR*, **abs/1901.08248**, 2019. 37

[82] **GSQL: Graph Query Language | TigerGraph — tigergraph.com**. https://www.tigergraph.com/gsql/. [Accessed 16-07-2024]. 37

[83] **Dgraph | GraphQL Cloud Platform, Distributed Graph Engine — dgraph.io**. https://dgraph.io/. [Accessed 16-07-2024]. 37

[84] Antonio Quiña-Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés. **GraphQL: A Systematic Mapping Study**. *ACM Comput. Surv.*, **55**(10):202:1–202:35, 2023. 37

[85] Oracle. **Integrated Graph Database**. https://www.oracle.com/database/integrated-graph-database/. [Accessed 16-07-2024]. 38

[86] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. **PGQL: a property graph query language**. In Peter A. Boncz and Josep Lluís Larriba-Pey, editors, *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, page 7. ACM, 2016. 38

[87] **SQL Server 2022 | Microsoft — microsoft.com**. https://www.microsoft.com/en-us/sql-server/sql-server-2022. [Accessed 16-07-2024]. 38

[88] PostgreSQL Global Development Group. **PostgreSQL — postgresql.org**. https://www.postgresql.org/. [Accessed 16-07-2024]. 39

[89] **pgRouting Project &x2014; Open Source Routing Library — pgrouting.org**. https://pgrouting.org/. [Accessed 16-07-2024]. 39

[90] **GitHub - BitnineGlobal/agensgraph — github.com**. https://github.com/BitnineGlobal/agensgraph. [Accessed 16-07-2024]. 39

[91] **MariaDB Foundation - MariaDB.org — mariadb.org**. https://mariadb.org/. [Accessed 16-07-2024]. 40

[92] **OQGRAPH — mariadb.com**. https://mariadb.com/kb/en/oqgraph-storage-engine/. [Accessed 16-07-2024]. 40

[93] Thomas E. Anderson. **The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors**. *IEEE Trans. Parallel Distributed Syst.*, **1**(1):6–16, 1990. 52

[94] Bijun He, William N. Scherer III, and Michael L. Scott. **Preemption Adaptivity in Time-Published Queue-Based Spin Locks**. In David A. Bader, Manish Parashar, Sridhar Varadarajan, and Viktor K. Prasanna, editors, *High Performance Computing - HiPC 2005, 12th International Conference, Goa, India, December 18-21, 2005, Proceedings*, **3769** of *Lecture Notes in Computer Science*, pages 7–18. Springer, 2005. 52, 53

[95] John M. Mellor-Crummey and Michael L. Scott. **Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors**. *ACM Trans. Comput. Syst.*, **9**(1):21–65, 1991. 53

[96] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. **Bringing Parallel Patterns Out of the Corner: The P^3 ARSEC Benchmark Suite**. *ACM Trans. Archit. Code Optim.*, **14**(4):33:1–33:26, 2017. 55

[97] S. Loosemore, R. McGrath, R.M. Stallman, and A. Oram. *The GNU C Library Reference Manual*. Servizio editoriale universitario, 1999. 55

[98] Jianbin Fang, Chun Huang, Tao Tang, and Zheng Wang. **Parallel programming models for heterogeneous many-cores: a comprehensive survey**. *CCF Trans. High Perform. Comput.*, **2**(4):382–400, 2020. 60

[99] Moritz Kaufmann, Manuel Then, Alfons Kemper, and Thomas Neumann. **Parallel Array-Based Single-and Multi-Source Breadth First Searches on Large Dense Graphs.** In *EDBT*, pages 1–12, 2017. 64, 93

[100] Scott Beamer, Krste Asanovic, and David A. Patterson. **Direction-optimizing breadth-first search**. In Jeffrey K. Hollingsworth, editor, *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, page 12. IEEE/ACM, 2012. 64

[101] Daegun Yoon and Sangyoon Oh. **SURF: Direction-Optimizing Breadth-First Search Using Workload State on GPUs**. *Sensors*, **22**(13):4899, 2022. 66

[102] Mireya Paredes, Graham D. Riley, and Mikel Luján. **Exploiting Parallelism and Vectorisation in Breadth-First Search for the Intel Xeon Phi**. *IEEE Trans. Parallel Distributed Syst.*, **31**(1):111–128, 2020. 67

[103] Peter A. Boncz. **LDBC: benchmarks for graph and RDF data management**. In Bipin C. Desai, Josep Lluís Larriba-Pey, and Jorge Bernardino, editors, *17th International Database Engineering & Applications Symposium, IDEAS '13, Barcelona, Spain - October 09 - 11, 2013*, pages 1–2. ACM, 2013. 72

[104] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. **The LDBC Social Network Benchmark**. *CoRR*, **abs/2001.02299**, 2020. 72

[105] John D McCalpin. **Stream: Sustainable memory bandwidth in high performance computers**. 87

[106] R. J. Atwal, Peter A. Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves Le Maout, Boaz Leskes, Yao Liu, Alex Monahan, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, and Yannick Welsch. **MotherDuck: DuckDB in the cloud and in the client**. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. 93