GraphCSR: A Degree-Equalized CSR Format for Large-scale Graph Processing

Xinbiao Gan
University of Defense Technology, Changsha, China
xinbiaogan@nudt.edu.cn

Tiejun Li
University of Defense Technology, Changsha, China
litiejunnudt@163.com

Chunye Gong
National Supercomputer Center in Tianjin, China
gongchunyenudt@163.com

Dongsheng Li
University of Defense Technology, Changsha, China
lidongshengnudt@163.com

Dezun Dong
University of Defense Technology, Changsha, China
dongdezunnudt@163.com

Jie Liu
University of Defense Technology, Changsha, China
liujienudt@163.com

Kai Lu
University of Defense Technology, Changsha, China
luknudt@163.com

ABSTRACT

Graph processing underpins a vast array of data-centric applications, serving as a crucial component in fields such as social network analysis, recommendation systems, bio-informatics, and search engines. As graph data grows in scale and complexity, high-performance graph processing is increasingly essential. Many graph processing tasks depend on efficient data structures to manage the sparsity typical of real-world graphs, where most vertices have limited connectivity. This sparsity poses challenges for memory and computational efficiency in large-scale graph processing, and conventional sparse formats like Compressed Sparse Row (CSR) often struggle with memory and computation inefficiencies when handling massive graphs. To address these challenges, we introduce GraphCSR, a degree-equalized CSR format specifically tailored to enhance the spatio-temporal efficiency of distributed graph processing across various tasks. GraphCSR aggregates low-degree vertices into synthetic high-degree ones and applies group-wise compression to reduce storage overhead by recording only the starting index for each aggregated group. This reduces memory usage and supports batch-memory access to improve performance. Our extensive evaluations in various graph processing algorithms and datasets demonstrate that GraphCSR not only reduces the memory footprint required for large-scale graphs, but also improves performance across multiple types of graph processing tasks, outperforming popular sparse storage formats. Furthermore, when deployed on a production-scale supercomputer with 79,024 nodes, GraphCSR achieved a graph processing throughput that exceeded the top-ranked system on the Graph500 benchmark.

PVLDB Reference Format:
Xinbiao Gan, Tiejun Li, Chunye Gong, Dongsheng Li, Dezun Dong, Jie Liu, and Kai Lu. GraphCSR: A Degree-Equalized CSR Format for Large-scale Graph Processing. PVLDB, 18(11): 4255 - 4268, 2025.
doi:10.14778/3749646.3749691

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749691

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at *https://anonymous.4open.science/r/GraphCSR-450E/README.md.*

1 INTRODUCTION

Graph processing has emerged as a fundamental paradigm in modern data analytics, particularly in systems where relationships between entities play a central role. Many core graph algorithms, such as breadth-first search (BFS) and depth-first search (DFS), serve as essential building blocks for executing complex analytical queries over graph-structured data. These operations are increasingly supported in graph-aware database systems to enable efficient exploration and pattern discovery. Typical applications include epidemic trajectory analysis [35], influence maximization in social networks [5], and graph-based ranking mechanisms used in search engines [17, 25]. In such scenarios, graph processing tasks often resemble recursive query evaluation, subgraph matching, or transitive closure computations, challenges traditionally addressed in the database community. As graph workloads become more prominent in large-scale distributed systems [4, 13, 29, 32, 36, 46, 50], bridging the gap between graph analytics and database query processing has become increasingly important.

To support such diverse applications efficiently, graph data must be represented and processed in a way that aligns with both algorithmic requirements and system-level performance goals. A common approach in graph systems and graph-aware databases is to represent relationships between entities using adjacency structures. Among them, the adjacency matrix offers a simple yet expressive representation that supports a wide range of computations. However, leveraging this structure effectively requires addressing the inherent sparsity of real-world graphs, where most vertices are connected to only a small subset of others, leading to adjacency

matrices that are large but mostly empty [5, 7, 9, 48]. This sparsity introduces challenges for storage efficiency and computational performance, particularly in distributed settings where memory footprint and communication overhead are critical constraints.

Efficient representation of sparse matrices is critical for large-scale data processing, especially in memory-bound scenarios, such as graph analysis. Among various formats, the compressed Sparse Row (CSR) format has gained widespread adoption due to its compact structure, which avoids storing zero entries. By maintaining only the essential components, namely, the values of nonzero elements, along with their corresponding column indices and row delimiters, CSR significantly reduces memory usage. Although several other sparse matrix formats are optimized for computational throughput in operations such as Sparse Matrix Vector (SpMV) or Matrix-Matrix Multiplications (SpMM) [10, 34, 39], or designed to leverage heterogeneous computing platforms [26, 40], CSR mainly emphasizes storage compactness. This characteristic makes it especially advantageous in graph processing workloads [38, 42, 45, 48], where handling massive adjacency structures within limited memory budgets is often the main bottleneck.

Many widely adopted sparse matrix formats, including CSR and its successors [7, 9, 45], were not originally crafted with graph processing in mind. Although effective for general sparse data, these representations fail to scale efficiently when applied to real-world graph workloads. This shortfall stems from the unique nature of graph data that large graphs are not only sparse but also exhibit extreme degree imbalance, with a significant portion of vertices connected to just a handful of neighbors. These low-degree vertices, while lightweight in computation, create overhead in memory because of the inefficient way they are encoded. As the size of the graph increases, such inefficiencies accumulate, becoming a dominant bottleneck in processing performance.

Efforts to minimize the memory footprint in sparse representations have led to several CSR-based enhancements recently. DCSR [7, 21] tackles redundant zero entry storage by introducing two offset-related metadata arrays, *JC* and *AUX*, allowing for more compact index calculations. CSCSR [9, 21] further refines this approach by consolidating these auxiliary arrays into a single structure *Skiplist*, improving storage efficiency without sacrificing access patterns. Taking it a step further, BCSR [45] combines design elements from both predecessors and introduces additional compression metadata *OFFSET* and *BITMAP*, which further reduces memory usage, albeit with added computational overhead [21]. However, despite these layered optimizations, a key limitation remains that all these formats treat each vertex identically, regardless of its degree. In real-world graphs dominated by low-degree vertices, this undifferentiated encoding strategy misses critical opportunities for targeted memory savings.

After understanding the memory footprint and the execution time of current parallel CSR-based applications, a optimizing partitioned CSR-based SpMM is designed to to fully exploit the powerful computing capability of the Sunway TaihuLight supercomputer [12]. SuperCSR [21], along with its extensions tailored for scalable graph construction [19] and large-scale web graph querying [20], introduces a degree-aware design that distinguishes between high- and low-degree vertices. This enables constant-time access to auxiliary metadata structures, an advantage particularly suited to supercomputing platforms. However, SuperCSR-like formats [19–21] are all tightly coupled with BFS-based processing pipelines and do not take advantage of batching acceleration, severely limiting their broad applicability to general-purpose sparse matrix operations such as SpMV and SpMM.

This lack of generality hinders its applicability across a broader range of graph computing tasks. SuperCSR applies a fixed threshold to separate vertex degrees, but does not explain the rationale for its choice, which limits its flexibility in adapting to various graph structures. However, current state-of-the-art CSR-based optimizations face the challenge of handling high-throughput or batched graph operations effectively within general machines other than supercomputers.

In contrast, we introduce GraphCSR[1], a new sparse matrix storage format engineered to be versatile and efficient for general graph tasks, including SpMV and SpMM in diverse systems. Designed as an enhancement to the conventional CSR format, GraphCSR maintains compatibility with existing graph algorithm implementations, yet introduces greater flexibility and significantly reduces memory consumption. By incorporating design elements suitable for broader scenarios, GraphCSR extends the application scope beyond the BFS on supercomputers, making it feasible for more general graph-based processing tasks such as database transactions.

A central observation underpinning our design is that real-world graphs contain a disproportionately large number of vertices with identical, low degrees. Instead of storing these vertices individually, GraphCSR takes advantage of this structural redundancy to achieve a memory-efficient representation. In particular, many parallel graph frameworks [8, 17, 24, 25, 37, 38, 51] already rely on degree-based vertex sorting as a pre-processing step.

Using this approach, GraphCSR partitions low-degree vertices into degree-based groups and fuses complementary degrees to form synthetic high-degree vertices. This design achieves compact representation through group-wise storage and offset-based indexing, significantly cutting redundancy while maintaining access efficiency. At runtime, edge existence checks and traversals can be computed using lightweight arithmetic, without decompressing the full structure. This design not only shrinks the memory required to store the adjacency matrix but also enables efficient memory access patterns through group-wise batching, ultimately accelerating large-scale graph computation.

We evaluated GraphCSR by applying it to representative graph algorithms from typical graph datasets [18, 47, 52, 53]. Our baselines include six main sparse matrix storage formats [7, 17, 24, 25, 48, 51]. We tested GraphCSR on a large-scale HPC system using up to 79,024 nodes with more than 1.2 million processor cores. The experimental results show that GraphCSR consistently outperforms the baseline methods with higher storage efficiency and fewer processing times. When applying GraphCSR to the Graph 500 BFS benchmark, we can outperform the top-ranked supercomputer on the Graph500 list (June 2023), achieving 1.6× greater throughput while consuming less 25% memory and using fewer CPU cores. Extensive empirical evaluations show that GraphCSR consistently outperforms state-of-the-art sparse matrix formats in both processing speed and memory efficiency.

[1]Code available at *https://anonymous.4open.science/r/GraphCSR-450E/README.md*

This paper makes the following contributions.

- We present GraphCSR, a degree-equalized CSR format specifically tailored to enhance space and time efficiency by aggregating low-degree vertices into balanced groups, enabling memory reduction and batching access.
- Extensive evaluations show the effectiveness and efficiency of GraphCSR. More specifically, GraphCSR leads the leaderboard in **Graph500 BFS ranking** using up to 77.2K nodes with 57.8% higher GTEPS and 75% lower memory footprint.

2 BACKGROUND AND MOTIVATION

2.1 Large-scale Graph Processing

A typical large-scale graph processing pipeline usually comprises four main stages: raw data ingestion, graph preprocessing, graph construction, and graph application. The pipeline begins with raw data ingestion, which refers to any dataset that can be modeled as a graph, such as logs, relational records, or interaction data. In the graph preprocessing stage, we perform vertex sorting, typically based on vertex degrees, to enhance storage locality and computational efficiency. The graph construction phase transforms the data into a distributed graph structure. At this stage, the choice of graph storage format and partitioning strategy is critical, much like data layout design in distributed databases.

Both real-world and synthetic graphs are often characterized by extreme sparsity and skewed degree distributions, with a substantial fraction of vertices having few or no neighbors. To address this, existing storage formats such as CSR and its variants [7, 9, 45, 48] adopt strategies that explicitly record vertex degrees while omitting zero-degree vertices to reduce storage overhead. This is particularly important because zero-degree vertices, though structurally present, occupy non-trivial space in the *RST* array of the CSR format without contributing to actual connectivity information. Eliminating such redundant metadata is therefore a common technique to optimize memory usage during graph initialization. However, CSR-like formats failed to exploit low-degree vertices while there is an overwhelming majority of the low-degree vertices over high-degree vertices in a given graph [24].

Our work focuses on designing an efficient, compressed storage format tailored for large-scale graph data, enabling fast access and minimizing communication overhead. Finally, graph application includes typical analytics tasks such as subgraph query, pattern mining, and neighborhood search, which are closely aligned with graph query processing in modern graph-aware database systems.

2.2 CSR-like Formats for Graph Representation

The CSR format has become a de facto standard for representing sparse matrices in large-scale graph processing due to its balance of storage efficiency and access performance. In contrast to alternative formats such as Dictionary of Keys (DOK), List of Lists (LIL), and Coordinate List (COO) [11, 34, 39], CSR offers more compact memory layouts and faster traversal for common matrix operations. A CSR encoded sparse matrix is made up of three primary one-dimensional arrays: `val`, `RST`, and `COL`. The `val` array holds all nonzero elements in row-major order, while the `RST` array records the offset of each row within the `val` array. The `COL` array stores the column indices corresponding to the nonzero entries. For graph-based applications,

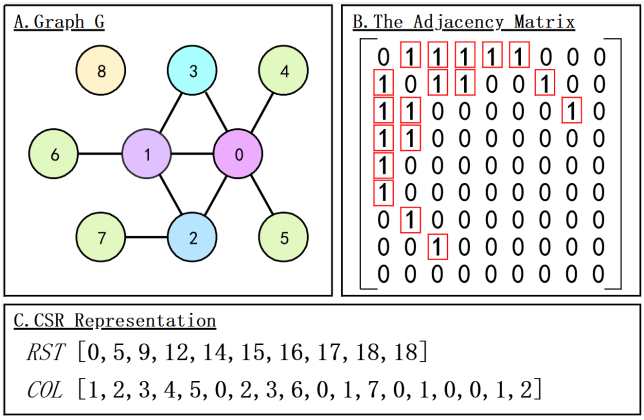

Figure 1: CSR representation (C) of the graph adjacency matrix (B) for a graph G (A).

Algorithm 1: Determine if a two vertices i and j are connected using CSR

Input: (i, j) `// Vertex's index in adjacency matrix`

```
start ← RST[i]
end ← RST[i+1]
for index = start to end do
    if j ∈ COL[index] then
        return 1
return 0
```

where adjacency structure rather than actual edge weights is typically of primary concern, the `val` array can be omitted without loss of structural information, simplifying interpretation and reducing memory usage.

Figure 1 gives an example of how an undirected graph (Figure. 1A) and its adjacency matrix (Figure 1B) can be represented using CSR (Figure 1C). Algorithm 1 shows how to determine a nonzero value between two vertices i and j, where a non-zero (one in this example) indicates that the two vertices are directly connected. Here, the `COL` array stores the column indices of nonzero values. The `RST` array contains the starting index of each row in the sparse matrix within the `COL` array and an additional element indicating the end of the last row. To determine if the element at position (i, j) is nonzero in CSR, we first examine the column range of the `RST` array corresponding to row i. The starting index is `start=RST[i]`, and the ending index is `end=RST[i+1]`. Next, we iterate through the indices between `COL[start]` and `COL[end]` in the `COL` array. If a value in `COL` matches the desired column index j, the element at position (i, j) is non-zero. Otherwise, it is a zero value.

2.3 Graph 500 Benchmark

As concrete use cases, this work targets two fundamental graph algorithms, BFS and SSSP, defined in the Graph500 benchmark [1]. Graph500 is the *de facto* standard for assessing the ability of a

computer system to process graphs [24, 37, 38, 43, 44]. Graph500 provides a graph generator to mimic real-life graph structures. This tool takes two parameters, a graph factor and an edge factor. For a graph size m and an edge factor n, it generates a graph of 2^m vertices and $n \times 2^m$ edges. In addition to the synthetic graph data generated by Graph500, we also evaluated our GraphCSR on two public graphs collected from real-life social networks [2, 3].

In accordance with the benchmarking methodology established by the Graph500 ranking, we adopt Giga Traversed Edges Per Second (GTEPS) as the primary performance metric to evaluate the efficiency of graph traversal. GTEPS quantifies the number of graph edges successfully visited per second during execution, serving as a standard indicator of throughput in large-scale graph processing tasks. As a performance-oriented metric, higher GTEPS values reflect more efficient traversal capabilities, making it suitable for comparing implementations across different systems and algorithmic frameworks.

2.4 Motivation

In real-world graphs, the majority of vertices have low degrees, and graphs often exhibit high regularity when sorted accordingly. Vertices with identical degrees tend to be frequently placed in consecutive positions within the storage arrays. Despite this regularity, conventional CSR and its many extensions treat all vertices independently, recording metadata for each row regardless of whether it encodes useful connectivity information. In particular, vertices with zero degree still occupy space in the indexing structures, and low-degree vertices, though inexpensive to compute, can collectively contribute significant memory overhead.

Current CSR-based extensions, such as DCSR [7], CSCSR [9], and BCSR [45], aim to reduce space consumption by eliminating explicit storage for zero-degree vertices or compressing row metadata. However, these methods continue to treat each vertex independently, disregarding the potential for aggregation among equal-degree vertices. As a result, they fail to fully exploit the regularity introduced by graph sorting, missing opportunities for coalesced memory access, and further memory reduction, particularly in large-scale graphs where low-degree vertices dominate.

To address this limitation, we propose GraphCSR, a lightweight extension to CSR that exploits degree regularities in sorted graphs. GraphCSR applies a degree-equalized grouping and fusion strategy to compress low-degree vertices, storing only shared base positions for adjacency reconstruction. This approach substantially reduces metadata overhead while enabling batched memory accesses that improve bandwidth utilization and cache performance. Importantly, GraphCSR preserves compatibility with CSR-based execution models, requiring minimal changes to existing graph algorithms. By leveraging overlooked structural patterns, GraphCSR achieves a more space- and access-efficient representation for large-scale, traversal-centric graph processing.

3 GRAPHCSR PHILOSOPHY

3.1 Overview of GraphCSR

GraphCSR is designed to reduce the memory overhead associated with storing low-degree vertices in sparse graphs, building upon the widely adopted CSR format. While traditional CSR and its variants efficiently eliminate zero-degree vertices by avoiding the storage of empty rows, they do little to compress vertices with low but non-zero degrees, such as those with only one or two neighbors. In large graphs where such vertices dominate, their collective impact on memory consumption can be significant, particularly in the *RST* and *COL* arrays.

To improve storage efficiency for low-degree vertices, GraphCSR introduces a degree-equalized fusion strategy that systematically pairs vertices with complementary degrees to form synthetic high-degree ones. By consolidating neighbor information into contiguous memory regions, this approach enables more effective compression while preserving access locality. Unmatched vertices are handled separately to maintain structural completeness, allowing the method to balance compression effectiveness with algorithmic generality. This structural regularity, which arises naturally during the preprocessing phase of many distributed graph processing frameworks [24, 41, 44, 45, 49], allows GraphCSR to avoid redundant storage while retaining addressability for all vertices in the group.

Specifically, GraphCSR partitions the vertex set into two categories based on a tunable degree threshold parameter, denoted as *Thr*. Vertices with degrees higher than *Thr* are treated as in conventional CSR and stored in a dedicated *high_deg_RST* array, which mirrors the structure of the original CSR *RST*. In contrast, vertices with degrees at or below *Thr* are compressed using a hybrid folding and fusion scheme. In addition to grouping low-degree vertices by identical degree values, GraphCSR further improves storage efficiency by introducing a fusion mechanism that pairs vertices with complementary degrees (e.g., 1-degree with 4-degree when $Thr = 4$) to form synthetic high-degree vertices. The neighbor lists of these fused vertices are reorganized into contiguous regions in the *COL* array to maximize spatial locality and enable group-wise compression. The number of such fused combinations is tracked to ensure correct index reconstruction during traversal.

For the remaining unmatched low-degree vertices, GraphCSR groups them by degree and compresses their representation using a shared-offset scheme. Specifically, vertices with the same degree are stored consecutively, and their group information is recorded in *low_deg_index*, which marks the starting index of each group. Each entry in *low_deg_RST* then corresponds to the column offset of the first vertex in the group within the original *COL* array. Since all group members have identical degrees, their adjacency lists occupy fixed-size blocks in memory and can be implicitly reconstructed via arithmetic offset computation. However, because a subset of vertices in each degree group may have already been fused into synthetic vertices, GraphCSR maintains a *fusion_count* array to track how many such vertices have been removed from the group through fusion. This count is subtracted from the relative index to obtain the adjusted position within the compressed layout. During traversal, the neighbor list of any vertex is derived by combining the group's base offset with the vertex's relative position, thus eliminating the need for per-vertex row pointers.

The threshold *Thr* serves as a tunable hyperparameter that balances the trade-off between compression ratio and computational overhead. Lower values of *Thr* result in fewer folded vertices and thus less compression, while higher values may introduce more decoding cost. A detailed analysis of *Thr*'s impact on performance is provided in Section 4.3.

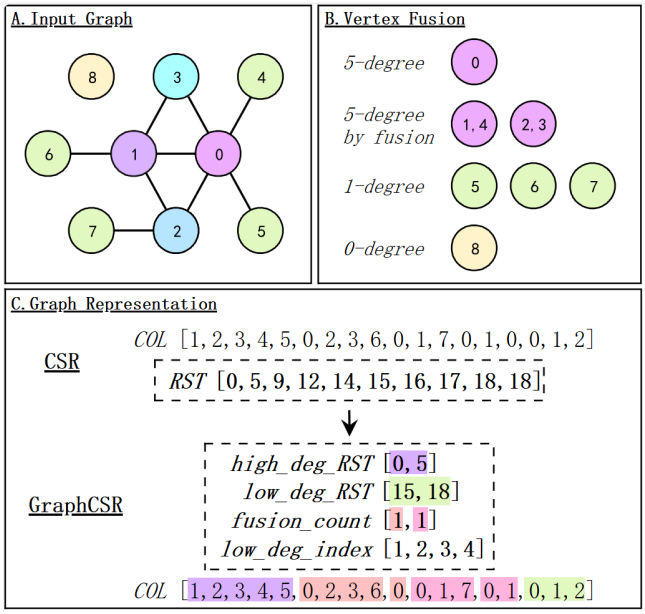

Figure 2: (A) the input graph from Figure 1; (B) vertex grouping after integrating multiple low-degree vertices into high-degree vertices; (C) the graph representation with GraphCSR. Here, `high_deg_RST` records the `RST` values for high-degree vertices, `low_deg_RST` records the starting offsets in `COL` for the vertices with degrees smaller than `Thr` which are not been fused, `fusion_count` counts the number of fusion vertices per degree combination, and `low_deg_index` records the first vertex id for each degrees.

By compressing structurally redundant entries, GraphCSR reduces the worst-case space complexity of storing low-degree rows from $O(N_{nz})$ to $O(1)$ per group, where N_{nz} denotes the number of nonzero elements. The folded structure enables more cache-friendly batched access and reduces metadata overhead without sacrificing compatibility with CSR-based graph algorithms. In the remainder of this section, we detail the algorithmic construction and runtime behavior of GraphCSR, demonstrating its applicability to real-world graphs with skewed degree distributions.

3.2 GraphCSR Sparse Storage Format

To provide a concrete understanding of how GraphCSR encodes sparse graph structures, we use the graph previously shown in Figure 1 and illustrate its representation under GraphCSR in Figure 2. In this example, we set the edge-degree threshold parameter $Thr = 4$, meaning that any vertex with a degree greater than 4 is classified as a high-degree vertex, while those with a degree of 4 or less are considered low-degree vertices.

Under this configuration, GraphCSR separates vertices into two storage pathways. High-degree vertices are stored in the array *high_deg_RST*, which adopts the conventional CSR row-pointer structure. Low-degree vertices are grouped by their degrees and managed using several auxiliary arrays: *low_deg_RST* records the starting offsets in the *COL* array for unmatched low-degree vertices, *low_deg_index* stores the first vertex ID in each degree group to facilitate indexing, and *fusion_count* tracks the number of fused vertices generated by combining low-degree vertices according to degree pairings.

The `fusion_count` array records the number of fused vertex pairs for each degree combination. For example, `fusion_count[0]` = 1 indicates that one pair of 4-degree and 1-degree vertices has been fused, while `fusion_count[1]` = 1 indicates one pair of 3-degree and 2-degree vertices has been fused. In addition, all low-degree vertices are grouped by their degrees. The array `low_deg_index` stores the starting vertex ID for each degree group. For instance, `low_deg_index[0]` = 1 means that 4-degree vertices begin at ID 1, and `low_deg_index[1]` = 2 means 3-degree vertices begin at ID 2. The `low_deg_RST` array maintains the starting `COL` offset for unmatched vertices within each degree group. For example, `low_deg_RST[0]` = 15 denotes that the unmatched 1-degree vertices begin at index 15 in the `COL` array, and `low_deg_RST[1]` = 18 indicates the end of the compressed region for low-degree vertices. Together, these three arrays enable efficient determination of whether a low-degree vertex has been fused, and allow accurate reconstruction of its adjacency list.

Furthermore, GraphCSR supports batched access, which groups the adjacency retrieval of multiple vertices into a single memory operation. This improves data locality and cache utilization, particularly beneficial in large-scale distributed graph processing where memory bandwidth often becomes a bottleneck.

Consequently, Figure 2 illustrates how GraphCSR leverages degree-aware folding and fusion strategies to compactly represent low-degree vertices while maintaining the conventional CSR layout for high-degree vertices. This hybrid design ensures compatibility with existing CSR-based graph algorithms while significantly improving storage efficiency and improving memory locality.

3.3 GraphCSR-based Graph Processing

To demonstrate the practical applicability of GraphCSR in real-world graph processing workloads, we provide two representative examples that integrate the folded storage format into fundamental graph algorithms. These examples not only validate the functional correctness of the proposed method, but also highlight its advantages in indexing efficiency, memory access locality, and computational throughput.

We begin with Algorithm 2, which illustrates how GraphCSR supports efficient neighborhood access in a BFS traversal. In this scenario, the graph vertices are assumed to be sorted in descending order of their edge degrees, a preprocessing step commonly applied in distributed graph systems to improve parallelism and memory access patterns [24, 45]. For high-degree vertices, GraphCSR defaults to the traditional CSR access strategy, retrieving the adjacency list via the standard row pointer array *high_deg_RST* and column index array *COL* (lines 2-4). This ensures compatibility and avoids unnecessary overhead for vertices with large neighborhoods, where compression benefits are minimal.

For low-degree vertices whose degrees fall below the predefined threshold *Thr*, a more compact access mechanism is used. GraphCSR first uses *low_deg_index* to locate the degree group of a low-degree vertex and compute its relative offset within the

Algorithm 2: GraphCSR-based Neighbor-indexing for Descending Graph

Input: COL of CSR format
Degree Thr
v_c // Current node
Output: neighbors // Neighbors of current node

1 Retrieve v_c's ID id_c and degree deg_c
// Get the start and end addresses in COL for high-degree vertices
2 **if** $deg_c < Thr$ **then**
3 $A_{start} = high\_deg\_RST[id_c]$
4 $A_{end} = high\_deg\_RST[id_c + 1]$
// Get the start and end addresses in COL for low-degree vertices
5 **else**
// Compute the bias of v_c in its same-degree vertices and the degree combination index
6 $bias = id_c$ - $low\_deg\_index[Thr - deg_c]$
7 $comb = 2 * deg_c > Thr \ ? \ Thr - deg_c : deg_c - 1$
8 **if** $bias < fusion\_count[comb]$ *or* $2 * deg_c == Thr + 1$ *and* $bias < 2 * fusion\_count[comb]$ **then**
// v_c is fused
9 $outer\_index = high\_deg\_RST[-1] + (Thr + 1) * (\sum_{i=0}^{comb-1} fusion\_count[i] + bias)$
10 $inter\_index = 2 * deg_c > Thr \ ? \ 0 : Thr - deg_c + 1$
11 **if** $2 * deg_c == Thr + 1$ *and* $bias \geq fusion\_count[comb]$ **then**
12 $outer\_index = outer\_index - (Thr + 1) * (2 * fusion\_count[comb] - bias - 1)$
13 $inter\_index = deg_c$
14 $A_{start} = outer\_index + inter\_index$
15 **else**
// v_c is not fused
16 $begin = low\_deg\_RST.length() - deg_c - 1$
17 $A_{start} = low\_deg\_RST[begin] + bias$
18 $A_{end} = A_{start} + deg_c$
19 Based on the A_{start} and A_{end} positions in COL, index the neighbors of the current vertex
20 return neighbors

group (lines 6–7). This offset is compared with *fusion_count* to determine whether the vertex has been fused (line 8). If fused, its index is derived from its position among all fused vertices (lines 9–14); otherwise, its *COL* offset is obtained from *low_deg_RST* using the adjusted offset (lines 15–17). The adjacency list is then reconstructed by reading a fixed-length segment from *COL*, based on the vertex's degree (lines 18–20).

To further evaluate the generality of GraphCSR, Algorithm 3 demonstrates its use in an SpMV setting, where the graph is viewed as an adjacency matrix A, and the output vector $Y = A \cdot X$ is computed for a given dense vector X. In each iteration, we use the same neighbor retrieval mechanism from Algorithm 2 (line 2) to access the adjacency list of a vertex, and then aggregate the corresponding values from X (lines 4–6). This operation is common in a wide array of graph analytics tasks, including PageRank, personalized recommendation, and spectral clustering. The use of GraphCSR ensures that such computations benefit from compact indexing structures and improved cache locality, especially in graphs dominated by low-degree vertices.

Algorithm 3: GraphCSR-based SpMV for Descending Graph

Input: Graph G and GraphCSR format
Degree Thr
X // Dense vector
Output: Y // Y = A * X, A is the adjacency matrix of graph G represented by GraphCSR

1 **for** $v \in G$ **do**
2 Retrieve the neighbors of v ($neighbors\_v$) according to Algorithm 2
3 $sum = 0$
4 **for** $neighbor \in neighbor\_v$ **do**
5 $sum = sum + X[neighbor]$
6 Y.append(sum)
7 return Y

Table 1: Evaluating testbed settings

Name	#Nodes	CPU	Memory/Node
Tianhe-Exa	79,024	Phytium 16-core	16GB
WuzhenLight	512	HG2 64-core	256 GB
Single-machine	1	Intel i7-10750	16GB

Overall, the integration of GraphCSR into both traversal-centric and linear-algebraic graph algorithms confirms its versatility and efficiency. The folded structure is not tied to a specific algorithm or graph topology and can be readily extended to other computational models, such as pull-based or hybrid execution engines. In the following sections, we provide further evaluation of GraphCSR 's performance, including its compression effectiveness and runtime behavior on large-scale real-world graphs.

4 EXPERIMENTAL SETUP

4.1 Evaluation Platforms

To assess the portability and scalability of GraphCSR across heterogeneous high-performance computing (HPC) environments, we conduct experiments on two supercomputing systems with distinct CPU architectures, as well as a single-node baseline platform. Table 1 summarizes the hardware and software configurations of all platforms, along with the maximum number of compute nodes used in our evaluations.

The first HPC system, WuzhenLight, consists of compute nodes equipped with dual 64-core HG2 CPUs (2.5 GHz), which implement the AMD x86-64 instruction set. In our experiments, we utilize up to 512 nodes from this system. The second platform, Tianhe-Exa, is based on Phytium 16-core CPUs (2.0 GHz) per node, with up to 79,024 nodes used in large-scale runs. Both systems run customized Linux distributions built on kernel version 9.3.0.

In addition to the HPC-scale experiments, we include a single-node comparison platform to evaluate the baseline performance of GraphCSR. This system is powered by an Intel Core i7-10750H processor, providing six cores at a base frequency of 2.6 GHz. It serves as a representative desktop-class environment for evaluating GraphCSR 's efficiency on commodity hardware.

All systems use MPICH 10.2.0 as the MPI implementation and libgomp 4.5 for OpenMP-based parallelism. Benchmarks are compiled using GCC 10.2.0 with the -O3 optimization flag to ensure high-performance binary generation.

4.2 Workloads

To evaluate the effectiveness and portability of GraphCSR across diverse graph processing workloads, we conduct experiments using both synthetic and real-world datasets, with a primary focus on traversal-centric algorithms. Our main benchmark task is BFS, following the specification of the widely adopted Graph500 benchmark suite [27]. Graph500 is recognized as the de facto standard for assessing the performance of large-scale graph processing systems [24, 37, 38, 43, 44], as it provides scalable graph generators and well-defined performance metrics.

The Graph500 generator produces synthetic Kronecker-style graphs that exhibit structural characteristics similar to those found in real-world networks, such as power-law degree distributions and small-world properties. As explained in Section 2.3, the generator takes two parameters: the scale m and the edge factor n. In our experiments, unless otherwise stated, we use the default edge factor $n = 16$, which is the standard setting recommended by Graph500.

To complement the synthetic datasets, GraphCSR is also evaluated on two large-scale real-world graphs to validate its applicability beyond the synthetic benchmarks. The first data set is clueweb12[3], a massive web hyperlink graph containing approximately 987 million vertices and 42.6 billion edges. The second is twitter-2010[2], a social interaction graph consisting of 41.7 million vertices and 1.47 billion edges. These two graphs present different structural characteristics, including significant skewness and irregular degree distributions, which make them ideal for assessing compression efficiency and access performance under realistic conditions.

Although BFS serves as the primary workload for evaluating traversal performance, we further extend our evaluation to include several additional graph processing kernels to assess the generality of GraphCSR. These include DFS, Single-Source Shortest Path (SSSP), PageRank (PR), Connected Components (CC), Betweenness Centrality (BC), and Triangle Counting (TC). Together, these tasks cover a wide spectrum of algorithmic patterns, ranging from traversal and path computation to iterative and structural analysis, allowing us to comprehensively examine the efficiency and versatility of GraphCSR across various graph computing scenarios.

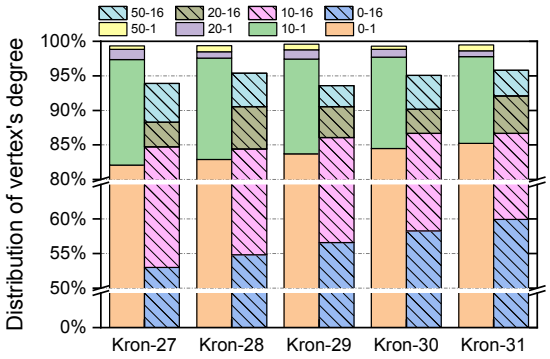

Figure 3: Degree Distribution of Kronecker graphs (Kron-x). Such that x represents the scale of the graph, a larger x refers to a larger graph. The first column represents the graph with $edge\_factor = 1$ and the second $edge\_factor = 16$. The larger $edge\_factor$, the denser graph. For example, the percentage of "50-16" means the distribution of vertex's degree between 20 to 50 in a graph with $edge\_factor = 16$.

4.3 Key Parameter (Thr) Tuning

In this subsection, we will take the synthetic graphs generated by Graph500-Kronecker [7] to demonstrate how the hyperparameter Thr will affect our approach. In addition, by digging deeper, we demonstrate how to further finetune the Thr for fast graph processing.

It is important to know the graph we are dealing with before actually tuning Thr. In Figure 3, we first show the vertices distribution of Kron-x (x represents the scale of the graph), generated by the Kronecker generator, to simulate the real-world graphs. A Kron-x graph may have 2^x vertices and $edge\_factor * 2^x$ edges, in which we manually set $edge\_factor$ as 1 and 16 in this scenario. Note that a larger $edge\_factor$ generates a denser graph.

From Figure 4, we have observed that 0-degree vertices are up to 85.21%, vertices with $0 \leq degree \leq 9$ are climbing to 97.79%, the proportion of vertices ($0 \leq degree \leq 20$) is steadily increasing to 98.61%, while the ratio of vertices with $0 \leq degree \leq 50$ is close to 99.46% but climbs at a snail's pace. Interestingly, the low-degree distribution does not vary greatly with the sparsity of the graph, though the number of zeros decreases as the graph gets denser. For example, a denser kron-31 ($edge\_factor$=16) would have less 0-degree vertices (59.94%), but it still holds 86.66% of the vertices with $0 \leq degree \leq 9$.

Moreover, we can conclude that low-degree vertices, e.g., whose degree is larger than 0 and smaller than 10, hold more than 12% and 27% for sparse and dense Kronecker graphs. Accordingly, the optimal Thr should theoretically be larger than 9, but still, the performance would be varied as the low-degree vertices proportion changed. With this distribution in hand, we can set up a Thr range and evaluate the sensibility of Thr across different scales of graphs, see Figure 4.

In addition to clueweb12 and twitter-2010, we also conduct tens of public graphs and get similar distributions. Further, the selection

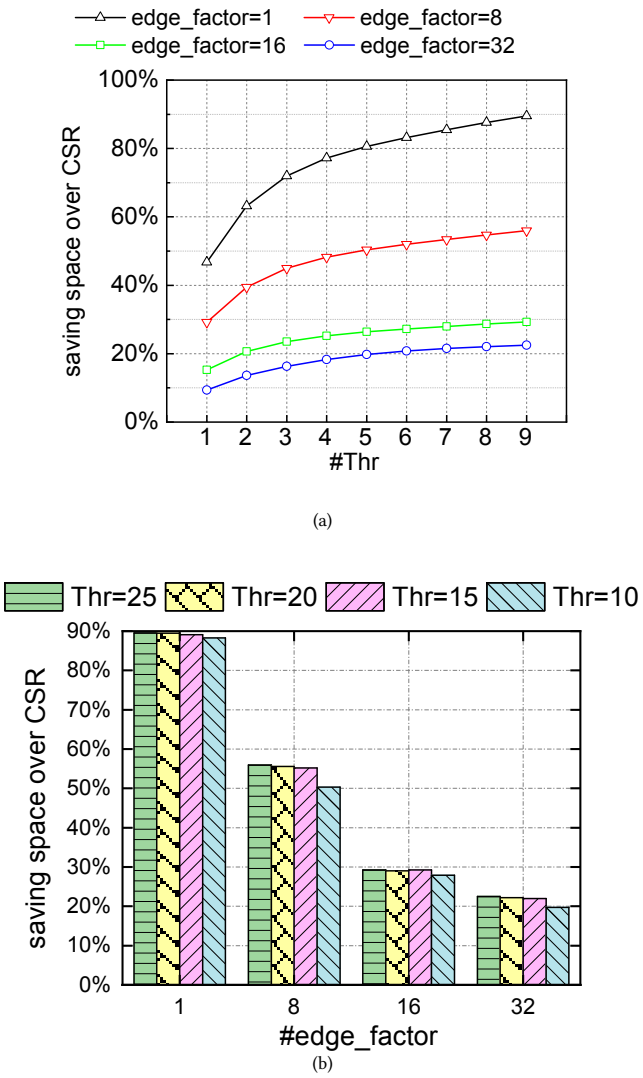

Figure 4: Sensibility of *Thr* and varying degrees of graph Kron-31, in which saving space = $\frac{|M(CSR)-M(\textsc{GraphCSR})|}{M(CSR)} \times 100\%$, M(X) represents the memory cost of format X (i.e., CSR or GraphCSR).

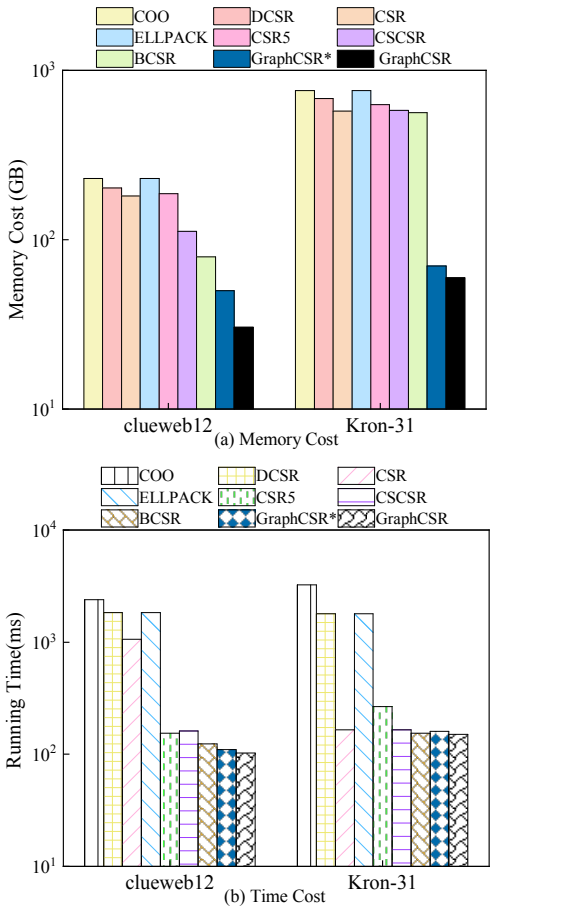

Figure 5: Space and time comparison on BFS based on various spare matrix formats.

policy of *Thr* for real-world graphs demonstrate similarities with synthetic graphs.

We evaluate the performance of GraphCSR carefully by tuning *Thr*. We first list the results of $Thr \leq 9$ as shown in Figure 4(a) to prove that based on the degree distribution in Figure 3, every increase in *Thr* brings obvious benefits and the overall yield is linear. On the other hand, Figure 4(b) shows that when $Thr > 9$, the changes in *Thr* have little effect on its performance within the same $edge\_factor$. The largest performance gap would be around 5% when $edge\_factor = 16$. So far, we may draw a conclusion that: (i) The majority of the graphs have a large scale of N-degree vertices such that $N \leq 9$. In this case, $Thr = 9$ gains significant benefits. We strongly recommend that when one needs to carefully fine-tune *Thr*, one should first refer to the graph degree distribution. (ii) Although a larger *Thr* may give a better GraphCSR performance, GraphCSR is generally *Thr* -oblivious when $Thr > 9$.

4.4 Evaluation Methodologies

For performance evaluation, we adopt the standard methodology defined by the Graph500 benchmark. Each test case is executed ten times on an unloaded system, with 64 root vertices randomly selected in each run to ensure statistical robustness. We report the geometric mean of the resulting GTEPS values to mitigate the impact of outliers. In all experiments, we observed minimal variance (less than 2%) in the measured performance, indicating high stability and reproducibility of the results.

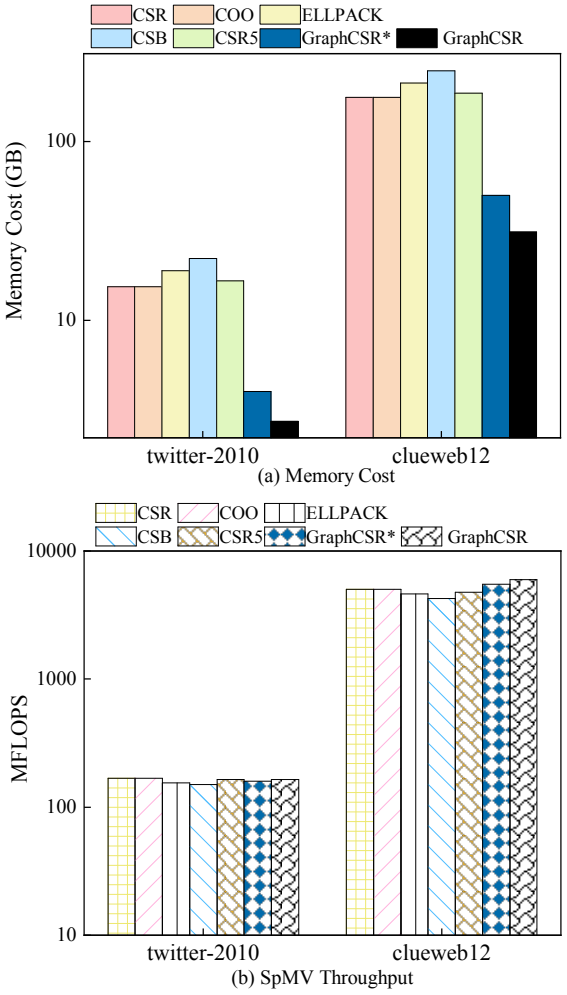

Figure 6: The SpMV performance of GraphCSR with (a) Memory and (b) MFLOPS (Mega FLoating point Operations Per Second) against the state-of-the-art CSR-like formats for SpMV.

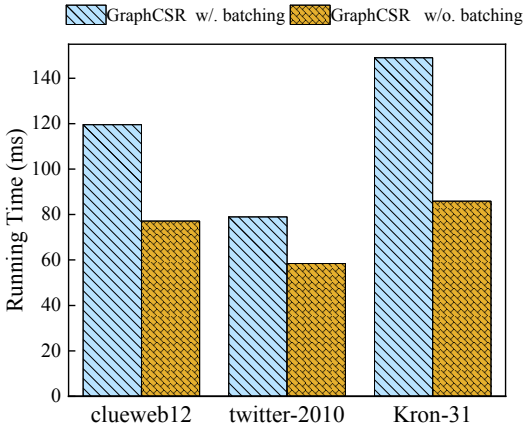

Figure 7: Batching improvement with GraphCSR on various graphs.

5 EXPERIMENTAL RESULTS

5.1 GraphCSR vs. CSR-like Formats in Memory Consumption

we first examine the memory footprint, which reports the memory cost when using different sparse matrix storage formats for BFS.

We find that the experimental memory cost is very close to the theoretical values claimed [21]. Let Δdr represent the deviation rate between the theoretical value (that is, V_{the}) and the experimental value (that is, V_{exp}) in the following.

$$\Delta dr = \frac{|V_{the} - V_{\exp}|}{V_{\exp}} \times 100\% \quad (1)$$

Our approach outperforms all the other CSR-like formats by saving more than 90% [2] memory space against most CSRs and up to 99. 8% space against the CSCSR format. The average deviation rate with respect to SuperCSR [21], is less than 0. 4%, even the highest deviation rates are less than 2%, which proves the reliability of GraphCSR. All show that GraphCSR has the smallest memory cost over state-of-the-art CSR-like formats, since GraphCSR not only eliminates 0-degree vertices, but also compresses the identical-degree vertices for graphs.

In addition to synthetic graphs (e.g. Kron-31), GraphCSR is also applied to real-world social graphs such as clueweb12 [3]. We also examine space and time cost of BFS based on various popular sparse matrix formats, including CSR, DCSR, CSCSR, BCSR, COO [33], CSR5 [39] and GraphCSR [20] with disabled fusion (marked as GraphCSR*). It shows that GraphCSR has the smallest memory cost, as shown in Figure 5(a) and the fastest running time, as shown in Figure 5(b) among all testing sparse matrix formats.

5.2 SpMV Performance

Although GraphCSR is specifically designed for large graph traversal, it can effectively support graph computation like SpMV. We evaluate the format GraphCSR in an isolated SpMV test on two real-world datasets Twitter-2010 and clueweb12. For a fair comparison, we reuse the two input graphs' topology with the randomly generated index vectors as the testing scenario. Figure 6 shows the SpMV performance of the CSR-based variants, CSB [6], GraphCSR* (i.e., GraphCSR [20] with fusion disabled) and CSR5 [39]. and non-CSR-formats including ELLPACK [28], and COO (Coordinate list) format [15].

Figure 6(a) shows that GraphCSR yields a memory cost lower than all previous CSR-like formats by up to orders of magnitude. On average, those CSR-based formats require extra memory to support vectorization and tiling, which perform well in small graph computations but are fatigued when facing large graphs. Building on

[2] Since COL is identical for all CSRs, it is not considered for space saving.

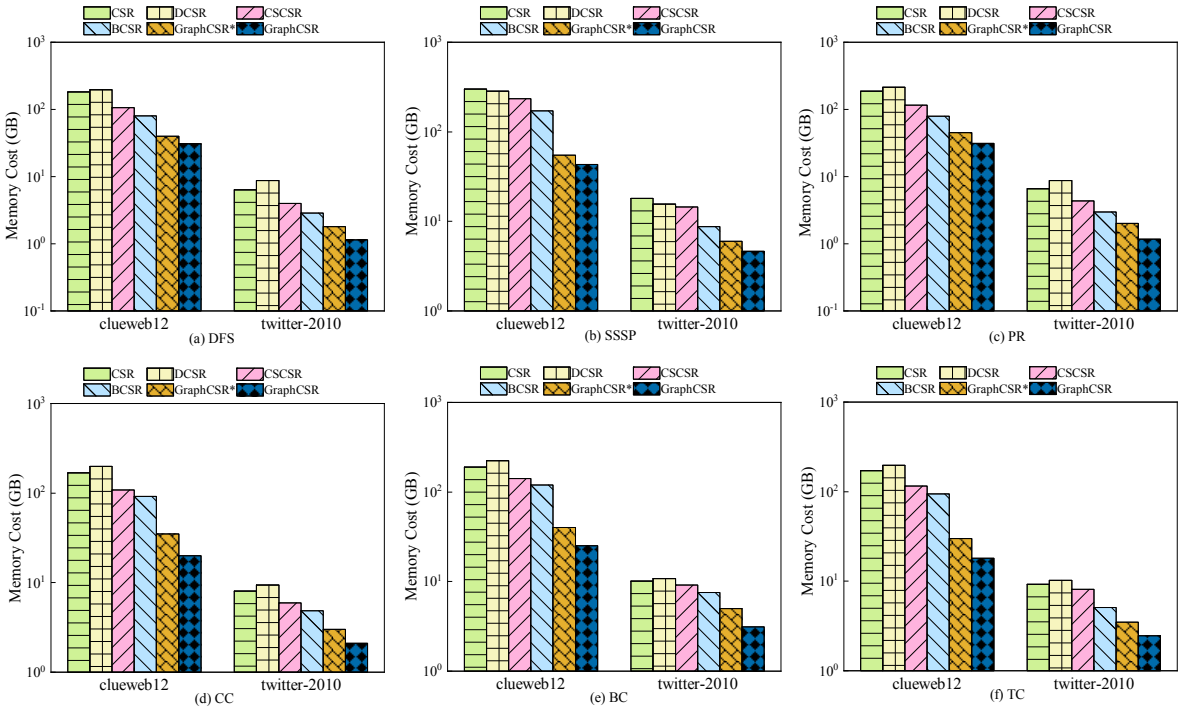

Figure 8: Memory footprint for deploying GraphCSR on two real-world graphsusing Tianhe-Exa with eight nodes available.

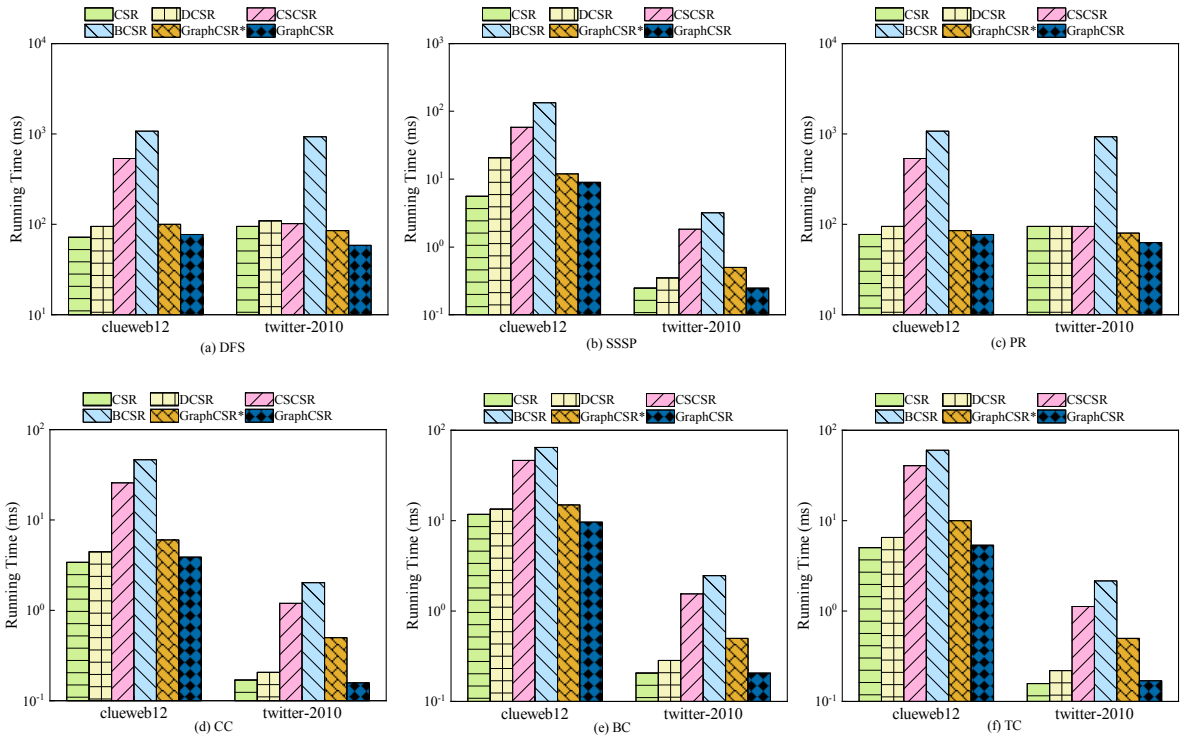

Figure 9: Runtime for deploying GraphCSR on two real-world graphs using Tianhe-Exa with eight nodes available.

Table 2: GraphCSR-based Graph500 BFS v.s. Fugaku (the latest Graph500 top-ranked supercomputer)

System	#Nodes	RAM (GB)	Storage format	GTEPS
Tianhe-Exa	79,024	1,264,384	GraphCSR	224,139.15
Fugaku	152,064	5,087,232	BCSR [45]	166,029

this memory advantage, Figure 6(b) demonstrates that GraphCSR also achieves superior computational performance in SpMV operations across a variety of large-scale graph datasets. The key lies in the reduced data movement of GraphCSR and the streamlined access pattern based on the aggregation of *fusion*. By eliminating excessive structural overhead and maintaining a contiguous and compact layout, GraphCSR enables faster traversal of neighboring vertices and more direct indexing of nonzero entries during the multiplication process.

5.3 GraphCSR with Batching

Current state-of-the-art sparse formats prioritize space reduction, often compromising decompression efficiency. As such, the key performance bottleneck is the achievement of improved decompression efficiency while substantially reducing space requirements. As such, we propose GraphCSR, which improves space efficiency by grouping vertices of the same degree and compressing ones of the low degree through fusion. This design enables batched and coalesced memory accesses, reduces storage overhead, accelerates decompression, and enhances overall graph processing performance.

Consequently, we demonstrate the improvement of the batching performance of GraphCSR with batching access (i.e., GraphCSR w/. batching) significantly outperforms GraphCSR without batching, that is, GraphCSR w/o. batching, as shown in Figure 7. It is worth noting that batching can not only boost the performance of a synthetic graph (e.g., Kron-31) but also advance the performance of real-world graphs, validating its versatility. In addition, performance improvements become prevalent by varying the size of the graph. Both clueweb12 and Kron−31 have more vertices and edges than twitter-2010, since there is more space for batching access, demonstrating good scalability.

5.4 GraphCSR-based Graph500 Ranking

GraphCSR has been successfully deployed on the Tianhe-Exa supercomputer, scaling up to 79,024 nodes, each equipped with 16 GB of RAM. Under this configuration, running a distributed Graph500 BFS benchmark with GraphCSR achieved a record-breaking performance of 224,139.15 GTEPS. This result represents a 13.52% improvement over the current top-ranked system, Fugaku, which reaches 166,029 GTEPS using more than 152,000 nodes and four times the total memory capacity.

As shown in Table 2, this comparison highlights the remarkable efficiency of our system, with higher throughput achieved using nearly half the number of nodes and significantly less memory. Importantly, both experiments were conducted on graphs of the same scale, generated using the Graph500 generator with an edge factor of 42 (Kron-42), yielding graphs with approximately 4.4 trillion vertices and 70.4 trillion edges.

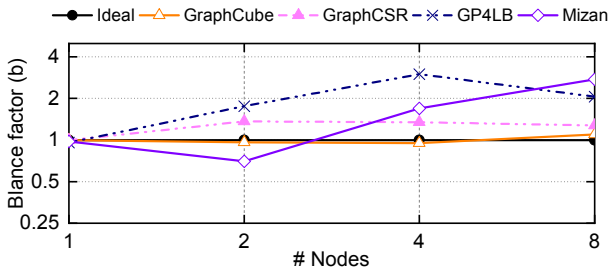

Figure 10: Loading balance factors when executing BFS using Tianhe-Exa with 8-node available. The closer the factor to 1, the more evenly distributed load is.

These results provide strong empirical evidence for the scalability and resource efficiency of GraphCSR. They demonstrate that the proposed GraphCSR not only supports high-performance graph traversal at extreme scales but also enables supercomputing platforms to deliver world-leading performance under tighter memory and compute constraints, making it a practical solution for modern large-scale graph processing workloads.

5.5 GraphCSR for Small-sized Machines

Although motivated by many-node supercomputers, GraphCSR can also behave well on smaller number of nodes even a single-alone machine. First, we compare GraphCSR with different CSR-like formats within a 8-node subset of Tianhe-Exa using various graph traversal algorithms, such as DFS, SSSP, PR, CC, BC, and TC, as shown in Figure 8 and Figure 9. The results show that GraphCSR outperforms prior CSR-like formats on all data sets and evaluation metrics. GraphCSR surpasses all CSR formats and saves at most 89.2% and 71.9% (average 77.3% and 65.5%) of space compared to naive CSR and BCSR, respectively. And refer to Figure 9, GraphCSR outperforms all CSR formats and offers speeds up to 19.3 times (average 14.4×) while running each popular graph algorithm. It is worth noticing that because real-world graphs are typically not hypersparse graphs, DCSR requires more memory than BCSR or even vanilla CSR. Correspondingly, our approach is highly stable when dealing with both hypersparse and non-hypersparse graphs since we are not solely relying on the number of 0-degree vertices as DCSR does.

Further, we examine GraphCSR with with state-of-the-art graph systems including GraphCube [23], Mizan [31] and DGP4LB (i.e., dynamic graph partitioning scheme to support load balance in distributed graph environments [14]) with load balance optimization. Figure 10 presents a comparison of load balance factors for GraphCube, Mizan and DGP4LB by running BFS on a graph (i.e. Kron-27) generated by Graph500 using 8 Tianhe-Exa. The balance factor of GraphCSR is still second to the state-of-the-art GraphCube [23] that is nearly optimal. where the load balance factor (b), is defined as

$$b = (\Theta\left(W^i\_load\right)/\Theta\left(A\_load\right))/N \tag{2}$$

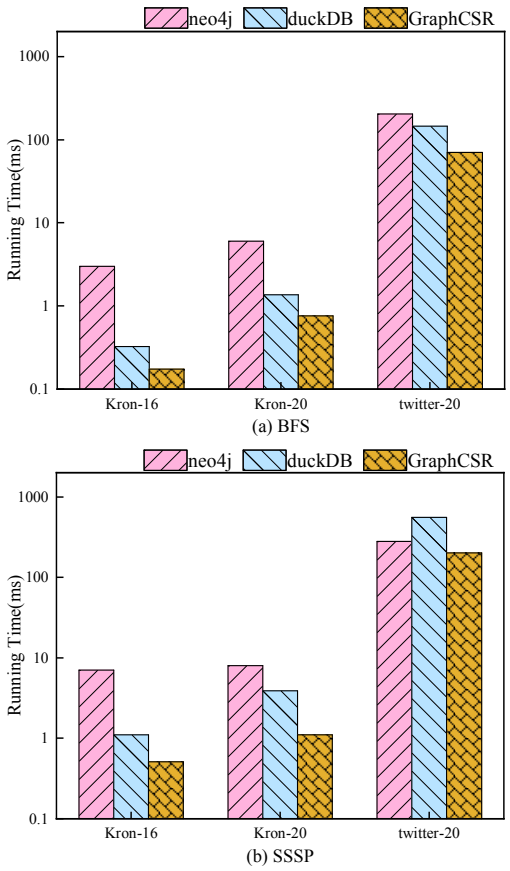

Figure 11: Comparions between GraphCSR and Neo4j, and DuckDB by running BFS and SSSP.

where $\Theta(W^i\_load)$ and $\Theta(A\_load)$ represent the working loads at node i and the average loads among all the running nodes, respectively, N is the number of running nodes.

Finally, we compare GraphCSR standard graph systems like Neo4j or DuckDB which offer optimized graph querying operators such as BFS and SSSP using a geneal single machine as listed in Table 1. Figure 11. demonstrates that GraphCSR significantly outperforms Neo4j or DuckDB in all graph operators. That is because both grouping identical vertices and batching access coordinately improve space and time efficiency, especially for sorted graphs.

5.6 Preprocessing Overhead

GraphCSR requires a one-time preprocessing step in which the input graph is sorted by degree of the vertex. This operation is performed only once per dataset and remains valid for all subsequent graph processing tasks. Extensive evaluations show that the preprocessing overhead of GraphCSR increases modestly with increasing graph scale. Even at a system scale of 512 compute nodes, the total preprocessing time remains low, only 2.07 seconds, highlighting the efficiency and scalability of the format in distributed environments.

Although graph sorting may appear costly, it is a one-time preprocessing step commonly shared across various graph workloads such as traversal, aggregation, and subgraph analysis. Modern graph systems, including Gemini [51], Shentu [38], GraphScope [17], Gluon [16], and GraphCube [23], have adopted sorted inputs as standard practice. Moreover, recent advances in high performance computing, such as vectorization and pipelining [22, 24], have significantly reduced the sorting overhead. GraphCSR is designed to align with these trends, leveraging sorted graphs to amplify space and performance benefits without introducing additional preprocessing costs, making it a practical and compatible solution for modern graph analytics pipelines.

GraphCSR supports SpMV and SpMM operations specifically optimized for graph analytics, but it is not ideally suited for general sparse matrix computations due to the vertex reordering it employs, which complicates direct index access common in standard CSR implementations. For general-purpose sparse matrix tasks, we recommend converting GraphCSR back to the native CSR format to fully leverage existing highly optimized CSR-based kernels. Existing methods such as CSR5 [39] and SMASH [30] focus primarily on parallelization and hardware-specific optimizations in small-scale matrices. However, these approaches do not address the significant memory overhead challenges encountered in large-scale graph processing workloads. Since memory often limits performance more than computation in HPC, GraphCSR 's memory-efficient design combined with reversible CSR conversion provides an effective trade-off between storage savings and computational compatibility, addressing key challenges in large-scale graph analytics.

6 CONCLUSION

We have presented GraphCSR, a CSR-like sparse storage format designed to optimize memory utilization for distributed large-scale graph processing. GraphCSR exploits the prevalence of low-degree vertices in real-world graphs by grouping vertices with identical degrees and combining complementary degree pairs, significantly reducing the storage footprint of the adjacency matrix. This organization enables GraphCSR to support efficient batch processing and coalesced memory access, leading to improved memory efficiency and higher throughput in graph processing tasks. We evaluated GraphCSR through theoretical analysis and extensive empirical experiments, comparing it with representative sparse matrix storage formats. Extensive evaluations show that GraphCSR consistently achieves significant reductions in memory consumption while simultaneously improving processing throughput compared to the baseline storage formats. Consequently, GraphCSR presents a promising approach for handling large-scale datasets in high-performance computing environments.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under grant Nos. 6205208, 62421002, 62372455 and 62272476, partially funded by the China National Key RD Program (No.2022YFB4501400), partially supported by the PDL innovation research fund under grant No. 2023-JKWPDL-09, partially supported by China's National Key Research and Development Program under grant No. 2023YFB3001903.

REFERENCES

[1] 2021. https://graph500.org/. (2021).

[2] 2022. https://law.di.unimi.it/webdata/twitter-2010/. (2022).

[3] 2022. https://lemurproject.org/clueweb12/. (2022).

[4] Yasir Arfat, Rashid Mehmood, and Aiiad Albeshri. 2018. Parallel Shortest Path Graph Computations of United States Road Network Data on Apache Spark. *Social Informatics and Telecommunications Engineering* (2018), 323–336.

[5] Cigdem Aslay, Laks VS Lakshmanan, Wei Lu, and Xiaokui Xiao. 2018. Influence maximization in online social networks. In *Proceedings of the eleventh ACM international conference on web search and data mining*. 775–776.

[6] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[7] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[8] Huanqi Cao, Yuanwei Wang, Haojie Wang, Heng Lin, Zixuan Ma, Wanwang Yin, and Wenguang Chen. 2022. Scaling graph traversal to 281 trillion edges with 40 million cores. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 234–245.

[9] Fabio Checconi and Fabrizio Petrini. 2014. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 425–434.

[10] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 32.

[11] Xinhai Chen, Peizhen Xie, Lihua Chi, Jie Liu, and Chunye Gong. 2018. An efficient SIMD compression format for sparse matrix-vector multiplication. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4800.

[12] Yuedan Chen, Guoqing Xiao, and Wangdong Yang. 2020. Optimizing partitioned CSR-based SpGEMM on the Sunway TaihuLight. *Neural Computing and Applications* 32, 10 (2020), 5571–5582.

[13] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. 2009. On compressing social networks. *SIGKDD* (2009), 219–228.

[14] Dojin Choi, Jinsu Han, Jongtae Lim, Jinsuk Han, Kyoungsoo Bok, and Jaesoo Yoo. 2021. Dynamic graph partitioning scheme for supporting load balancing in distributed graph environments. *IEEE Access* 9 (2021), 65254–65265.

[15] Hoang-Vu Dang and Bertil Schmidt. 2012. The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus. *Procedia Computer Science* 9 (2012), 57–66.

[16] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 752–768.

[17] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*. 1433–1445.

[19] Xinbiao Gan. 2025. GraphService: Topology-aware constructor for large-scale graph applications. *ACM Transactions on Architecture and Code Optimization* 22, 1 (2025), 1–24.

[20] Xinbiao Gan, Tiejun Li, Qiang Zhang, Guang Wu, Bo Yang, Chunye Gong, Jie Liu, and Kai Lu. 2025. GraphCSR: A Space and Time-Efficient Sparse Matrix Representation for Web-scale Graph Processing. In *Proceedings of the ACM on Web Conference 2025*. 763–771.

[21] Xinbiao Gan, Tiejun Li, Qiang Zhang, Bo Yang, Xinhai Chen, and Jie Liu. 2024. SuperCSR: A Space-Time-Efficient CSR Representation for Large-scale Graph Applications on Supercomputers. In *Proceedings of the 53rd International Conference on Parallel Processing*. 158–167.

[22] Xinbiao Gan, Qian Tang, Feng Xiong, Shijie Li, Bo Yang, and Tiejun Li. 2024. TianheStar: Orchestrating SSSP Applications on Tianhe Supercomputer. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 534–542.

[23] Xinbiao Gan, Guang Wu, Shenghao Qiu, Feng Xiong, Jiaqi Si, Jianbin Fang, Dezun Dong, Chunye Gong, Tiejun Li, and Zheng Wang. 2024. GraphCube: Interconnection hierarchy-aware graph processing. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 160–174.

[24] Xinbiao Gan, Yiming Zhang, Ruibo Wang, Tiejun Li, Tiaojie Xiao, Ruigeng Zeng, Jie Liu, and Kai Lu. 2021. TianheGraph: Customizing Graph Search for Graph500 on Tianhe Supercomputer. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 941–951.

[25] Xinbiao Gan, Yiming Zhang, Ruigeng Zeng, Jie Liu, Ruibo Wang, Tiejun Li, Li Chen, and Kai Lu. 2022. Xtree: Traversal-based partitioning for extreme-scale graph processing on supercomputers. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2046–2059.

[26] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.

[27] http://graph500.org/. 2021. The Graph 500 List. https://graph500.org/ Last accessed 03 March 2022.

[28] Konstantin Isupov, Ivan Babeshko, and Alexander Krutikov. 2021. Implementation of multiple precision sparse matrix-vector multiplication on CUDA using ELLPACK format. In *Journal of Physics: Conference Series*, Vol. 1828. IOP Publishing, 012013.

[29] Keita Iwabuchi, Hitoshi Sato, Yuichiro Yasui, Katsuki Fujisawa, and Satoshi Matsuoka. 2014. NVM-based hybrid BFS with memory efficient data structure. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 529–538.

[30] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 600–614.

[31] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European conference on computer systems*. 169–182.

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media? *Www '10 Proceedings of International Conference on World Wide Web* (2010), 591–600.

[33] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 238–252.

[34] Yishui Li, Peizhen Xie, Xinhai Chen, Jie Liu, Bo Yang, Shengguo Li, Chunye Gong, Xinbiao Gan, and Han Xu. 2020. VBSF: a new storage format for SIMD sparse matrix–vector multiplication on modern processors. *The Journal of Supercomputing* 76 (2020), 2063–2081.

[35] Zhe Li, Chengkun Wu, Yishui Li, Runduo Liu, Kai Lu, Ruibo Wang, Jie Liu, Chunye Gong, Canqun Yang, Xin Wang, et al. 2023. Free energy perturbation–based large-scale virtual screening for effective drug discovery against COVID-19. *The international journal of high performance computing applications* 37, 1 (2023), 45–57.

[36] Yongsub Lim, Won-Jo Lee, Ho-Jin Choi, and U Kang. 2015. Discovering large subsets with high quality partitions in real world graphs. In *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*. IEEE, 186–193.

[37] Heng Lin, Xiongchao Tang, Bowen Yu, Youwei Zhuo, Wenguang Chen, Jidong Zhai, Wanwang Yin, and Weimin Zheng. 2017. Scalable graph traversal on sunway taihulight with ten million cores. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 635–645.

[38] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. 2018. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 706–716.

[39] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[40] Yongchao Liu and Bertil Schmidt. 2015. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 82–89.

[41] David Mizell and Kristyn Maschhoff. 2009. Early experiences with large-scale Cray XMT systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–9.

[42] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[43] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhisa Sato. 2020. Performance Evaluation of Supercomputer Fugaku using Breadth-First Search Benchmark in Graph500. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 408–409.

[44] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. 2016. Extreme scale breadth-first search on supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 1040–1047.

[45] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. 2017. Efficient breadth-first search on massively parallel and distributed-memory machines. *Data Science and Engineering* 2, 1 (2017),

22–35.

[46] Norases Vesdapunt and Hector Garcia-Molina. 2016. Updating an Existing Social Graph Snapshot via a Limited API. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 1693–1702.

[47] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. 2022. Nebula Graph: An open source distributed graph database. *arXiv preprint arXiv:2206.07278* (2022).

[48] Gan Xinbiao, Tan Wen, and Liu Jie. 2021. Bidirectional-Bitmap Based CSR for Reducing Large-Scale Graph Space. *Journal of Computer Research and Development* 58, 3 (2021), 458.

[49] Yuichiro Yasui and Katsuki Fujisawa. 2015. Fast and scalable NUMA-based thread parallel breadth-first search. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 377–385.

[50] R. Zafarani and H. Liu. 2009. Social Computing Data Repository at ASU [http://socialcomputing.asu.edu]. *Informatics and Decision Systems Engineering* (2009).

[51] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[52] Lei Zou, Lei Chen, and M Tamer Özsu. 2009. Distance-join: Pattern match query in a large graph database. *Proceedings of the VLDB Endowment* 2, 1 (2009), 886–897.

[53] Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. 2011. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment* 4, 8 (2011), 482–493.