GraphService: Topology-aware Constructor for Large-scale Graph Applications

XINBIAO GAN, National University of Defense Technology, Changsha, China

Graph-based services are becoming integrated into everyday life through graph applications and graph learning systems. While traditional graph processing approaches boast excellent throughput with millisecond-level processing time, the construction phase before executing kernel graph operators (e.g., breadth-first search, single-source shortest path) can take up to tens of hours, severely impacting the quality of graph service. Is it feasible to develop a fast graph constructor that can complete the construction process within minutes, or even seconds?

This article aims to answer this question. We present GraphService, a flexible and efficient graph constructor for fast graph applications. To facilitate graph applications with better service, we equip GraphService with a hierarchy-aware graph partitioner based on communication topology as well as a graph topology-aware compression by exploiting a huge number of identical-degree vertices within graph topology. Our evaluation, performed on a range of graph operations and datasets, shows that GraphService significantly reduces communication cost by three orders of magnitude to construct a graph. Furthermore, we tailor GraphService for downstream graph tasks and deploy it on a production supercomputer using 79,024 computing nodes, achieving a remarkable graph processing throughput that outperforms the top-ranked supercomputer on the latest Graph500 list, with construction time reduced by orders of magnitude.

CCS Concepts: • **Mathematics of computing**; • **Theory of computation → Parallel computing models**; **Distributed computing models**;

Additional Key Words and Phrases: Graph construction, graph partitioning, graph representation, sorted graph, graph processing

ACM Reference Format:
Xinbiao Gan. 2025. GraphService: Topology-aware Constructor for Large-scale Graph Applications. *ACM Trans. Arch. Code Optim.* 22, 1, Article 2 (March 2025), 24 pages. https://doi.org/10.1145/3689341

Extension of Conference Paper: A preliminary version of this article "GraphCube: Interconnection Hierarchy-aware Graph Processing" by Gan et al. appeared at the *29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2024 [26]. This extended version makes the following new contributions to the conference paper, providing new insights based on the original paper. (1) We extend our approach to support large-scale graph construction, demonstrating the generality and performance advantages over prior approaches (Section 5.2). (2) We explore a space-time-efficient graph compression to store and index identical-degree vertices, which not only allows for batching and coalescing memory accesses for further boosting graph construction but also reduces memory footprint (Section 5.5). (3) We conduct extensive experiments to demonstrate the effectiveness and efficiency of GraphService. More specially, GraphService achieved the top position in the Graph500 ranking, utilizing up to 77.2K nodes with 17% higher throughput and more than four orders of magnitude reduction in construction time.

This work was supported in part by the National Natural Science Foundation of China (grant nos. 62372455, 62272476, 62032023, 42104078, and 6190241), the PDL Innovation Research Fund (grant no. 2023-JKWPDL-09), and China's National Key Research and Development Program (grant no. 2023YFB3001903).

Author's Contact Information: Xinbiao Gan, National University of Defense Technology, Changsha, Hunan, China; e-mail: xinbiaogan@nudt.edu.cn.

ACM 1544-3973/2025/03-ART2
https://doi.org/10.1145/3689341

1 Introduction

Graph-based services provide an efficient solution for managing the continually growing large-scale applications in both industry and academia [13, 24, 25, 40, 57, 62, 67, 69], for which the **quality of service (QoS)** [67] is essential. A graph encompasses a series of properties used to describe throughput, response time, and other relevant metrics.

As the graph scale (i.e., the number of vertices and edges) explosively grows, large-scale graphs need distributed machines to store and handle. For instance, the Sogou graph [69] contains 271.8 billion vertices and 12.3 trillion edges that need 38,656 nodes during testing [40]; the 155.25K-node Fugaku supercomputer conducts **breadth-first search (BFS)** on a Kronecker graph at scale=42 (having 4.4 trillion vertices and 70.4 trillion edges [9]). No single machine can accommodate and process such big graphs. As such, many real-world and mimic graphs need distributed machines to store and handle them.

While graphs show promise as applications on distributed systems at scale [24, 25, 27, 40, 73] and provide high QoS for many supercomputing users [62, 69], current graph processing systems often prioritize one aspect of QoS: high throughput, which refers to the processing speed of graph algorithms, also known as *graph operators*. However, such criteria do not always address the bottleneck factors that affect the performance of service-oriented graph applications. This is because the complete graph-based service comprises both the graph operator and the graph constructor. Indeed, the graph construction time is usually orders of magnitude higher than the processing time on graph operators. That is primarily because all state-of-the-art graph engines compete in the throughput of graph algorithms and take the processing speed of the graph operators as the sole criterion for graph applications. While this assertion might hold for standard graph processing on small-sized **high-performance computing (HPC)** systems, typically involving at most tens of computing nodes, it does not necessarily apply to distributed graph processing at scale.

In large-scale graph processing, graph construction is often a must, as shown in Figure 1. Graph processing typically involves four stages: (i) loading the graph, during which real-world graphs or synthetic graphs are handled by a graph constructor; (ii) graph preprocessing; (iii) graph construction, which mainly includes graph partition and graph storage; and (iv) graph application. During the preprocessing stage, various tasks are performed, such as counting degrees and sorting graphs [19, 23, 27, 29, 40, 59, 73]. For the graph construction stage, vertices and edges are distributed into many computing nodes by using an advanced graph partitioning algorithm (known as a *partitioner*) such as XTree [29] and stored based on a specified format such as the **compressed sparse row (CSR)** or its variants [9, 15, 23, 59, 66]. Finally, the constructed graphs are passed into graph processing algorithms such as BFS [8, 27], **single-source shortest path (SSSP)** [31, 63], **connected component (CC)** [44, 56], **PageRank (PR)** [7, 58], and **community detection with label propagation (CDLP)** [19, 22]. To provide user-friendly graphic services, we take QoS as the total cost (i.e., Θ) of four graph processing stages for distributed graph applications. Θ(QoS) is a lower-is-better metric for large-scale graph applications on HPC systems.

$$\Theta(\text{QoS}) = \tau_{\text{g}} + \tau_{\text{c}} + \tau_{\text{pre}} + \tau_{loading} \tag{1}$$

Herein, τ_{g} represents the processing time of graph operators, which determines the throughput. τ_{c} represents the processing time of graph construction that can affect the throughput of the graph operators. τ_{pre} and $\tau_{loading}$ are the preprocessing time and loading time of the graph, respectively. Since τ_{pre} can be parallelled in advance and $\tau_{loading}$ can be processed offline, we can approximate Θ(QoS) as follows.

$$\Theta(\text{QoS}) \approx \tau_{\text{g}} + \tau_{\text{c}} \tag{2}$$

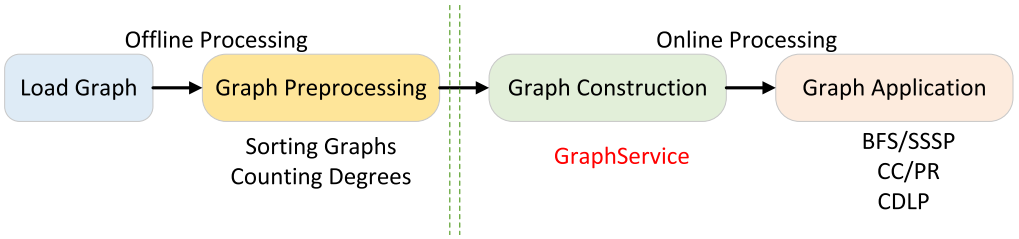


Fig. 1. A typical graph processing pipeline.

According to Equation (2), when processing large-scale graphs on HPC systems, the QoS of distributed graph applications encompasses two aspects, not just throughput. Further, τ_c is the bottleneck of the Θ(QoS) because it is often 3 $\sim$ 5 orders of magnitude higher than τ_g [9, 45]. In detail, communication costs within τ_c exceed 80% due to significant variability among different computing nodes [27, 29, 45]. This trend is expected to become even more pronounced with the increase of computing nodes.

The success of large-scale graph processing systems heavily relies on efficient partitioning algorithms and communication optimization techniques [29, 70, 73]. As such, various graph partitioning strategies have been proposed [6, 20, 29, 30, 32, 33, 42, 43, 45, 50, 51, 55]. Indeed, all graph processing systems utilize some form of graph partitioning to take advantage of the sparsity of the graph data[1] to distribute vertices and edges across computing nodes [27, 29, 40]. However, current advancements in large-scale graph engines have focused on achieving excellent throughput, often at the cost of longer construction times [12, 19, 29, 40, 48, 73]. A key reason is that these engines assume consistent communication overhead between any two nodes. While this assumption may be true for small-sized HPC systems [19, 73], it is not the case for processing big graphs at scale. Recent approaches utilize fine-grained partitioning techniques to exploit graph operators and leverage graph-specific attributes such as vertex distribution to mitigate communication overhead. While important, they only consider the graph-based topologies at best, such as sparsity, and ignore the communication hierarchy of the target HPC system, which severely hampers the graph construction. Although GraphCube [26] takes both the sparsity of the graph and the communication hierarchies into account, it is basically exploited to expedite graph traversal. In contrast, GraphService dedicates to leveraging hierarchical communication domains (in terms of a group of computing nodes attached to the same routing cell [26, 29]) to accelerate graph construction before graph operations (e.g., traversal) ingress. As we will show in this article, state-of-the-art graph processing systems, such as Fugaku [9, 45] and the Wuhan supercomputer [9], struggle with rapid graph construction.

To tackle this problem, we introduce GraphService,[2] which is specifically designed to optimize graph construction in order to minimize the overall online processing time rather than solely concentrating on graph operators. The key idea of GraphService is to (i) develop a hierarchy-aware partitioner that leverages the communication topology of target HPC systems to minimize communication costs and (ii) provide a space-time-efficient graph compression by grouping identical-degree vertices, enabling batch memory access for fast graph construction based on graph topology, where many graph vertices have the same degree [27].

We evaluate GraphService by applying it to representative graph operations across three different HPC systems with varying scales, using up to 79,024 nodes and over 1.2 million processor cores.

[1]Where most vertices of a real-life graph are low-degree vertices with only a small number of edges connected to them [29].
[2]Code available at https://anonymous.4open.science/r/GraphCSR-450E/README.md

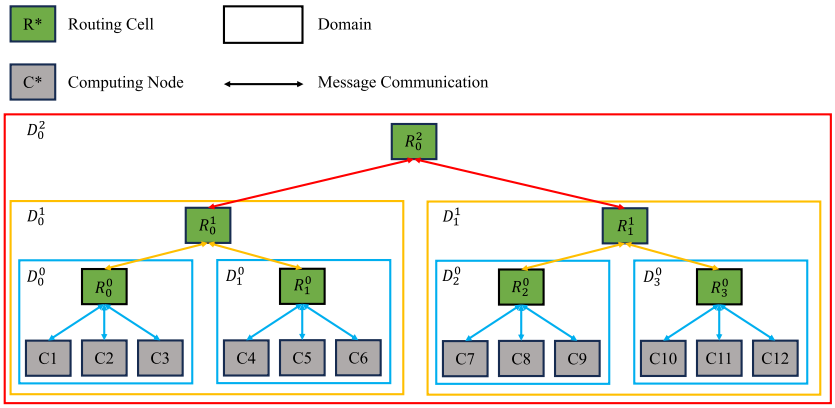


Fig. 2. A hierarchical communication topology with 3-level domains.

We show that GraphService consistently outperforms state-of-the-art graph partitioning methods, CSR-like formats, and graph systems on different graph scales and hardware setups. Specifically, GraphService achieves 162,494 and 23,021 **giga-traversed edges per second (GTEPS)**, respectively, for BFS and SSSP according to the Graph500 specification [9]. These results can be translated to 1.17× and 1.50× improvements, with three orders of magnitude construction time reduction for BFS and SSSP, respectively, compared with the top-ranked systems on the latest Graph500 ranking (November 2023). We also test GraphService on real-world graphs, for which it outperforms three state-of-the-art graph processing engines, Gemini [73], Gluon [18], and GraphScope [19], with a speedup of up to 18.9×.

Conference extension. A preliminary version of this article "GraphCube: Interconnection Hierarchy-aware Graph Processing" by Gan et al. appeared at the *29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2024 [26]. This extended version makes the following new contributions to the conference paper, providing new insights based on the original paper.

— We extend our approach to support large-scale graph construction, demonstrating the generality and performance advantages over prior approaches (Section 5.2).
— We explore a space-time-efficient graph compression to store and index identical-degree vertices, which not only allows for batching and coalescing memory accesses for further boosting graph construction but also reduces memory footprint (Section 5.5).
— We conduct extensive experiments to demonstrate the effectiveness and efficiency of GraphService. More specially, GraphService achieved the top position in the Graph500 ranking, utilizing up to 77.2K nodes with 17% higher throughput and more than 1,218× reduction in construction time.

2 Background

2.1 Communication Topology of HPC

Large-scale HPC systems often implement a hierarchical communication topology [49, 54, 62, 65] through **routing cells (RCS)** to link different computing nodes. In Figure 2, there are 12 computing nodes (denoted as C) and 7 routing cells (denoted as R), which are separated into 7 communication domains, i.e., D_0^0, D_1^0, D_2^0, D_3^0, D_0^1, D_1^1, D_0^2, where D_*^j denotes the j-th level domain and we call D_*^0 the leaf domain. Each communication domain D_i^j is associated with one RC,

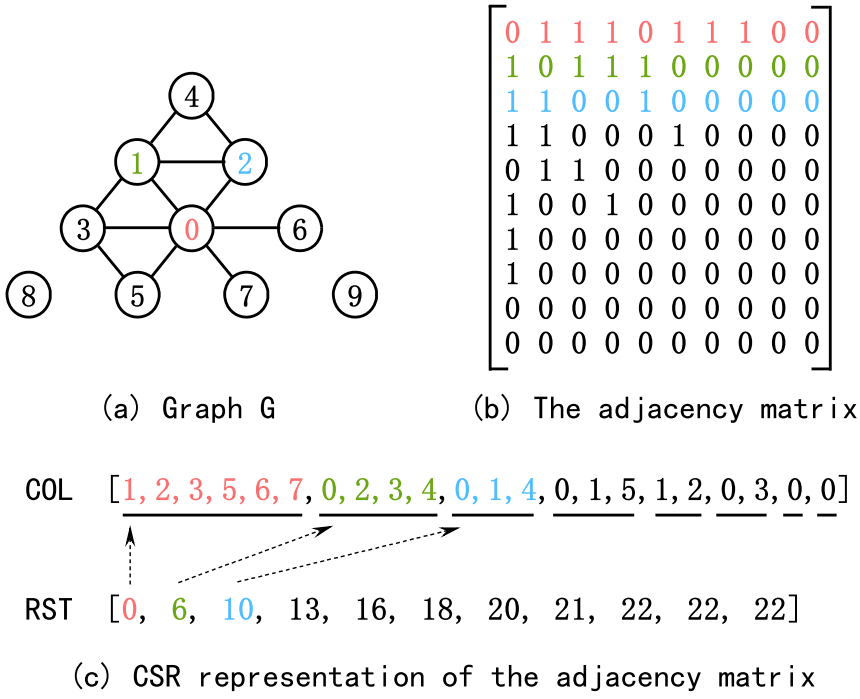


Fig. 3. CSR representation (c) of the graph adjacency matrix (b) for a graph **G** (a).

denoted as R_i^j; thus, the domains can be represented as $D_0^0 = \{R_0^0, C1, C2, C3\}$, $D_1^0 = \{R_1^0, C4, C5, C6\}$, $D_0^1 = \{R_0^1, C1, C2, C3, C4, C5, C6\}$, etc. We can observe that $C1$ and $C4$ are in different leaf domains, i.e., D_0^0 and D_1^0, but they are in the same domain D_0^1, i.e., the first-level domain.

In practice, there are significant differences in the communication latency at different levels of the communication domains. This drives us to orchestrate data layout among communication domains to minimize communication overhead.

2.2 CSR and Graph Distribution

A graph is usually represented using an adjacency matrix. The CSR format is widely used for storing sparse matrices in large-scale graph processing [17, 37, 41]. CSR represents a sparse matrix using three one-dimensional arrays: `val`, `RST`, and `COL`. The `val` array stores the non-zero values in contiguous locations, with the rows packed together. The `RST` array specifies the starting index of each row in the `val` array, while the `COL` array maps each non-zero value to its corresponding column. Graph processing algorithms mostly focus on accessing each vertex and then filtering vertices on demand. For clarity, the `val` array can be omitted for better understanding.

Figure 3 gives an example of how an undirected graph as shown in Figure 3(a) and its adjacency matrix as shown in Figure 3(b) can be represented using the CSR format as shown in Figure 3(c).

Many real-world graphs are sparse and uneven.[3] Prior sparse matrix storage formats [11, 14, 61, 66] essentially distinguish solely between zero and non-zero elements, meticulously recording

[3]This is also known as a small-world model, in which most vertices of a real-life graph are low-degree vertices with only a small number of edges connected to very few high-degree vertices [16, 27].

all non-zero entries of the matrix while concentrating efforts on eliminating 0-degree vertices to enhance memory utilization efficiency. However, GraphService is tailored for graph processing by further compressing the graph adjacency matrix by grouping the same-degree vertices (e.g., $vertices\{2, 3\}$, $\{4, 5\}$, and $\{6, 7\}$ shown in Figure 3). It is worth noting that identical-degree vertices make up the vast majority of real-world and synthetic graphs [27]. As a result, there is lots of room for GraphService to demonstrate its potential.

3 Related Work

Blocksome et al. developed a single-sided communication interface for the IBM Blue Gene/L supercomputer, leading to a threefold improvement in maximum bandwidth [10]. Faraj et al. proposed various enhancements by thoroughly utilizing the interconnection architecture and hardware capabilities of IBM Blue Gene/P, enabling near-peak efficiency in **message passing interface (MPI)** collective communication on the Blue Gene/P system [21]. IBM developed the **LAPI (Low-level Applications Programming Interface)**, which is a low-level, high-performance communication interface available on the IBM RS/6000 SP system [52]. It provides an active message-like interface along with remote memory copy and synchronization functionality. However, the limited set from the LAPI does not compromise on functionality expected on a communication API. Even worse, the topology mapping library and LAPI are designed for IBM supercomputers, resulting in difficulties in adaptation to applications running on general supercomputers, especially for Graph500 testing on other supercomputers.

To improve task mapping on the Blue Gene/L supercomputer, a topology mapping library was incorporated into the BG/L MPI library, enhancing both communication efficiency and application scalability [68]. This library could also offer scalable support for the MPI virtual topology interface. To tackle the significant challenges posed by Blue Gene/Q's extreme parallelism and scale, Kumar et al. developed the **Parallel Active Message Interface (PAMI)**, a communication library designed to maximize the performance of large-scale supercomputing applications [35]. IBM created the LAPI as a high-performance, low-level communication tool for the IBM RS/6000 SP system [52]. This interface provides features akin to active messages, along with capabilities for remote memory copying and synchronization. While LAPI's limited functionality set might appear constrained, it still meets the essential requirements for a communication API. However, the design of both the topology mapping library and the LAPI is specific to IBM supercomputers, which presents challenges when adapting these tools for applications on other general-purpose supercomputers. This issue is particularly pronounced during Graph500 testing on non-IBM systems.

Unlike the communication enhancements tailored for IBM supercomputers, Shida et al. created a specialized MPI library and low-level communication infrastructure for the K supercomputer, leveraging Open MPI and addressing the tofu topology [46, 53]. This approach, while similar in intent to IBM's solution, is specifically designed to optimize performance for the K supercomputer rather than for IBM's systems. To explore how MPI communication affects Graph500 performance, Li et al. performed a comprehensive examination of MPI Send/Recv and MPI-2 RMA, uncovering several performance bottlenecks. They then introduced an advanced, scalable design for Graph500 utilizing MPI-3 RMA, which enhanced GTEPS and achieved a twofold speed increase on the TACC Stampede Cluster [36]. Despite these advancements, translating these insights and optimizations into a universal communication library applicable to the Graph500 benchmark remains a challenge.

4 Motivation and Overview

4.1 Motivation Example

As a motivation, we consider the construction time released in the latest Graph500 rankings [9], as shown in Tables 1 and 2. According to Graph500 specification [9], the GTEPS is the ranking

Table 1. The Latest Graph500 BFS Top-Ranked Supercomputer with Construction Time C_Time (/second) and the Processing Time of Kernel Graph Algorithm (/second)

#Rank	System	GTEPS (#Scale)	C_Time	BFS_time
1	Supercomputer Fugaku	138,867 (42)	2,388.66	1.9
2	Wuhan Supercomputer	115,357.6 (41)	16,284.8	3.3

Table 2. The Latest Graph500 SSSP Top-Ranked Supercomputer with Construction Time C_Time (/second) and the Processing Time of Kernel Graph Algorithm (/second)

#Rank	System	GTEPS (#Scale)	C_Time	SSSP_time
1	Wuhan Supercomputer	15,335.9 (41)	16285	0.436
2	Pengcheng Cloudbrain-II	11,529.7 (40)	57218	0.655

metric for Graph500, where #scale means the inputting graph has $2^{\#scale}$ vertices and $16 \times 2^{\#scale}$ edges. The processing time of kernel graph algorithms can be estimated as $BFS/SSSP\_time = GTEPS/\left(16 * 2^{\#scale}\right)$ since the total traversed edges in practice are very close to the edges of the input graph [9, 27, 29].

Graph construction issues. It is evident that even the supercomputers Fugaku and Wuhan Supercomputer[4] achieve top-ranked performance, completing the BFS kernel in 1.9/3.3 seconds, yet they consume 2,388.66 seconds (approximately 40 minutes) [1, 9, 45] and 16,284.8 seconds (equivalent to 4.52 hours) [1], respectively, for BFS ingress listed in Table 1. Similarly, the top-ranked Wuhan Supercomputer and Pengcheng Cloudbrain-II[5] respectively take milliseconds to process the SSSP kernel while consuming 16,285 seconds (equivalent to 4.52 hours) and 57,218 seconds (equivalent to 15.89 hours) to construct graphs before entering SSSP, as shown in Table 2.

Although these systems win top-ranked throughout when processing graph operators, they wait up to tens of hours to ingress. It severely conflicts with the QoS according to Equation (2) and damages the experience of large-scale HPC systems.

Breakdown. Figure 4 illustrates communication and computation, including synchronization times per testing scale for Graph500 BFS. Communication time is split into the time of calling *send/recv*, *allgatherv*, and *alltoallv*. As the number of parallel nodes increases, communication overhead grows, while computation time decreases due to enhanced parallelism in graph processing. However, increased node counts can result in communication becoming a significant bottleneck in the overall program execution time. The total communication time, i.e., the accumulative time of *send/recv*, *allgatherv*, *alltoallv* and *synchronization*, can be up to 87% when running large-scale BFS on 65,536 computing nodes on the latest Tianhe supercomputer [28, 29, 38]. The main reasons are that (i) current systems ignore communication hierarchies, resulting in expensive communication costs and (ii) there is substantial irregular memory access and prior solutions failing to leverage access coalesce.

[4]The Fugaku and Wuhan Supercomputer are the top 2 in the latest Graph500 list in BFS ranking [9].
[5]The Wuhan Supercomputer and Pengcheng Cloudbrain-II are the top 2 in the latest Graph500 list in SSSP ranking [9].

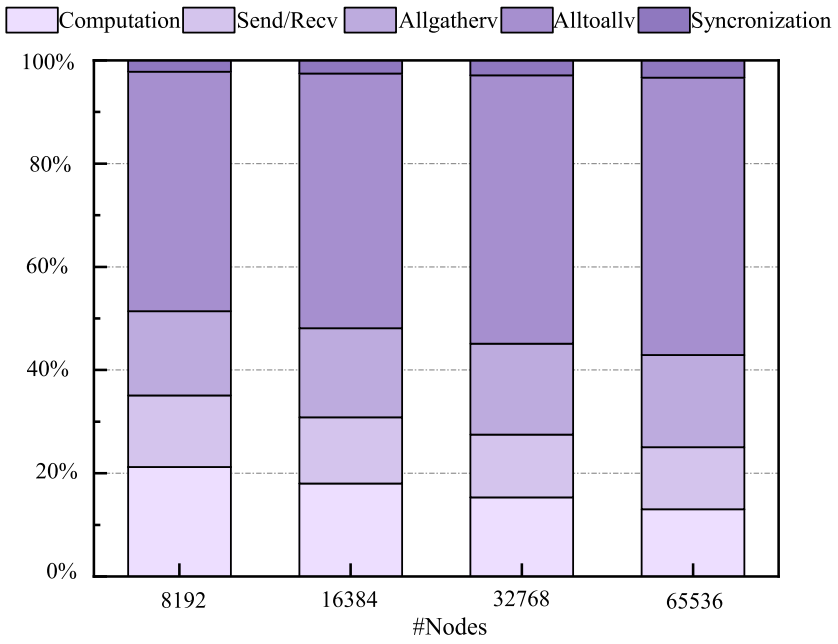


Fig. 4. BFS communication and computation breakdown.

Data layout. Figure 5(a) shows a graph with four vertices, $G = (a, b, c, d)$, and Figure 5(b) shows a hierarchical topology with three communication domains, $T = [D_1, D_2, D_3]$, where each domain D_i contains two CPUs. All the CPUs are the same, but the communication costs among them are different. For example, the intra-domain communication is about $0.1\mu s$, but inter-domain communication is increased by almost 10 times for one layer. Thus, the communication cost between CPUs 1 and 2 is $0.1\mu s$, between CPUs 1 and 3 is $1\mu s$, and between CPUs 1 and 5 is $10\mu s$. Given that one CPU can accept one graph vertex for computing, traditional methods are used to balance the workload for each domain, as shown in Figure 5(c): $a \rightarrow 1, b \rightarrow 3, c \rightarrow 5, d \rightarrow 2$. According to the communication costs mentioned above, the total communication cost is $1\mu s(a : b) + 10\mu s(a : c) + 1\mu s(b : d) = 12\mu s$. This is a sub-optimal assignment. If we are aware of the hierarchical topology of the target HPC system, we can come up with a better solution, which is shown in Figure 5(d): $a \rightarrow 1, b \rightarrow 3, c \rightarrow 2, d \rightarrow 4$. To this end, the final communication cost is $1\mu s(a : b) + 0.1\mu s(a : c) + 0.1\mu s(b : d) = 1.2\mu s$. By doing so, communication costs can be reduced by orders of magnitude. This is a motivation of GraphService to orchestrate data layout for expediting graph construction.

Coalescing access. In addition to communication topology boosting graph construction, CSR transformation can also batch memory access to further advance graph construction. Many real-world and mimic graphs are sparse and exhibit a skewed distribution, where many vertices have the same degree [25]. However, CSR-like formats only differ zero values from non-zero elements and then record all non-zero entries of the matrix. Specifically, there is a key regarding the row index array, `RST`, when using CSR to store the adjacency matrix of a sorted graph. Vertices with the same edge degree have adjacent lists stored in a contiguous manner in the `COL` array. We further observe that identical-degree vertices constitute a large proportion of real-life graphs, implying that they take up most of the storage space. Figure 6 shows how the vertices distribute in two real-world graphs, clueweb12 and twitter2010. Whereas clueweb12 owns 978,408,098 vertices and

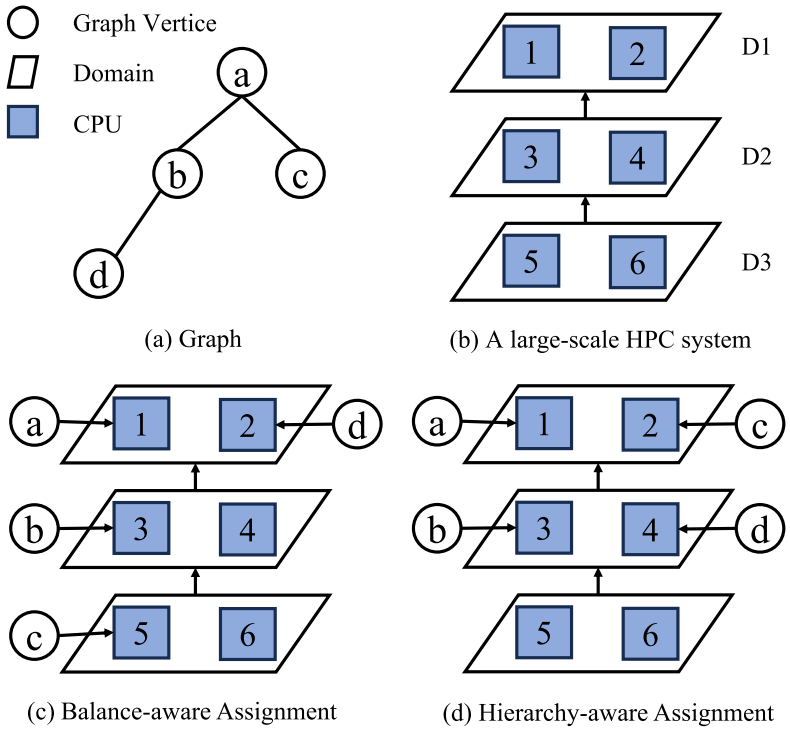


Fig. 5. Balance-aware vs. hierarchy-aware placement.

has a sparsity (i.e., non-zero degree vertices ratio) of 90.5%, twitter2010 owns 41,652,230 vertices and has a sparsity of 86.7%. In Figure 6, we show that accumulative 1-degree (i.e., in-degree and out-degree) vertices are up to 49.57% and 16.03% of the total vertices, respectively, for clueweb12 and twitter2010. Interestingly, the same-degree distribution does not vary significantly with the sparsity of the graph though the number of zeros decreases as the graph gets denser. More details of the vertices distribution can be referred to TianheGraph [27]. This gives us an extra dimension of motivation to develop GraphService for batching and coalescing memory accesses with memory reduction.

Lessons learned. The experimental results demonstrate the limitations of current graph processing engines in constructing graphs although they give excellent throughput. It also highlights the need to build an ultra-fast constructor, including a communication hierarchy-aware partitioner and a space-time-efficient graph compression.

4.2 Preliminaries

Definition 4.1 (Computing Node (CN)). A **computing node (CN)**, denoted as C, represents a kind of device in the exascale cluster that is responsible for message computing.

Definition 4.2 (Routing Cell (RC)). An RC, denoted as R, represents a kind of device in the large-scale HPC systems that is responsible for transferring messages between CNs, which can be further classified into leaf RC and high-level RC. A leaf RC is the lowest level of RC, which is responsible for the lowest level of message communication for CNs. High-level RC is responsible for message communication among RCs.

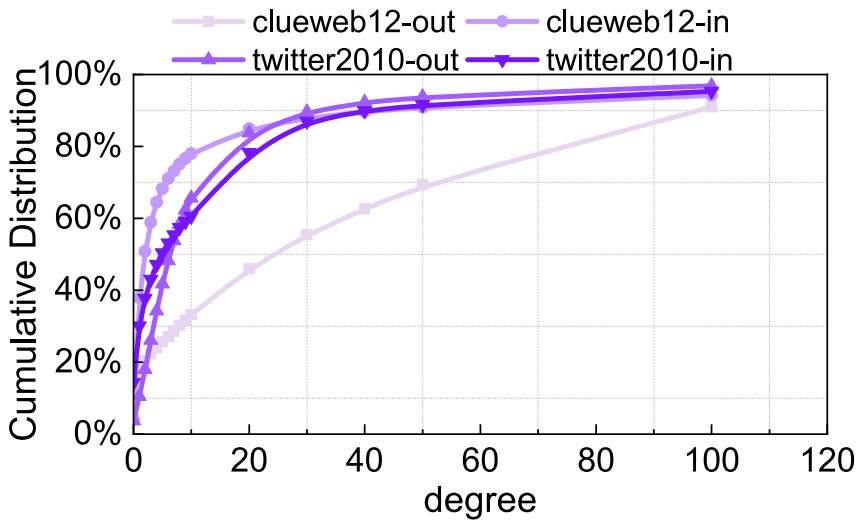


Fig. 6. Degree cumulative distribution of clueweb12 and twitter-2010, such that graph-in/out represents the in-degree or out-degree of the graph.

Definition 4.3 (Communication Domain). A communication domain, D, is a set of CNs that are attached to the same RC, which can be represented as $D = \{R, C1, C2, \cdots, Cn\}$, where n is the total number of computing nodes in D, and R is the routing node that all the CNs are connected with. Similar to RC, the communication domain can also be further classified into leaf domain and high-level domain. A leaf domain is a set of CNs attached to a leaf RC. For example, in Figure 2, leaf domain $D_0^0 = \{R_0^0, C1, C2, C3\}$ because $C1, C2, C3$ are attached to R_0^0, of which the parent domain (i.e., high-level domain) includes a set of CNs attached to the same high-level RC. For example, $D_0^1 = \{R_0^1, C1, C2, C3, C4, C5, C6\}$ since all the CNs within this domain are attached to R_0^1.

Definition 4.4 (Communication Cost). The communication cost between two CNs, Ci and Cj, can be defined as $ComCost(Ci, Cj) = cost^{intra} + \sum_{k=1}^{h} cost_k^{inter}$, where $cost^{intra}$ is the intra-domain communication cost, $cost_k^{inter}$ is the inter-domain communication cost between domain D_*^{k-1} and D_*^k, and h is the lowest domain level that Ci and Cj are both in.

In practice, the communication costs differ a lot among intra- and inter-domain communications. For example, the intra-domain communication cost can be 1 unit,[6] i.e., $cost^{intra} = 1\ \mathcal{U}$, the inter-domain communication cost between D_*^0 and D_*^1 can be $10\ \mathcal{U}$.

Definition 4.5 (Graph). A graph is denoted as $\mathbf{G} = (V, E)$, where V is a set of vertices, $v_i \in V$ represents the i-th node, and $N = |V|$ is the total number of vertices in $\mathbf{G}$; $E \subseteq V \times V$ represents a set of edges, $e_i = (v_j, v_k) \in E$ denotes a connection from v_j to v_k.

Definition 4.6 (Graph Partitioning). An s-cut graph partition of graph $\mathbf{G}$ can be defined as $P_s(\mathbf{G}) = \{\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_s\}$, where s is the total number of graph partitions, $\mathbf{G}_i$ is the i-th subgraph, $\mathbf{G}.V = \bigcup_{i=1}^{s} \mathbf{G}_i.V$, and $\mathbf{G}.E = \bigcup_{i=1}^{s} \mathbf{G}_i.E$.

Given a graph $\mathbf{G} = (V, E)$ and a large-scale hierarchical system (i.e., **Exa**), we can split and distribute $\mathbf{G}$ into a total of s subgraphs $P_s(\mathbf{G})$. An ultra-fast constructor aims to orchestrate the $P_s(\mathbf{G})$

[6] A $\mathcal{U}$ may be one microsecond, millisecond, or second depending on the target system.

into different CNs for graph operators ingressing for the best QoS by minimizing the construction time. It can be formulated as follows.

$$\Theta(\text{QoS}) \approx \min\left\{\tau_g + \tau_c\right\} \tag{3}$$

Owing to $\tau_g \ll \tau_c$ [29, 40, 60], the construction time (i.e., τ_c) is dominated by communication cost as shown in Figure 4 so that Equation (4) can be approximated as follows.

$$\begin{aligned} &\min \sum_{i=1}^{N}\sum_{j=1}^{N} ComCost(v_i, v_j), \\ &\text{subject to } v_i, v_j \in \textbf{Exa}.\text{CNs}, \end{aligned} \tag{4}$$

where N is the total number of vertices in **G** and **Exa**.CNs refers to the set of CNs belonging to **Exa**. $ComCost(v_i, v_j)$ is the message communication cost between v_i and v_j, which are distributed into CNs equipped in the **Exa**.

5 Methodology

5.1 Overview of GraphService

GraphService is designed to build an ultra-fast graph constructor before the kernel operators ingress for large-scale distributed graph-based applications. This is accomplished by orchestrating data layout based on a communication hierarchy-aware partitioner (Section 5.2), with the aim of minimizing communication latency. To further advance graph construction, a space-time-efficient graph compression is built in GraphService by grouping identical-degree vertices (Section 5.5), aiming to batch memory access along with memory reduction.

Implementation. We implement GraphService as a library in around 10K lines of C/C++ code tailored for ARM and x86 architectures. It provides **application programming interfaces (APIs)** for common graph operations, including those evaluated in this work.

5.2 Hierarchy-Aware Partitioner

Due to the significantly faster communication within lower-level domains compared with higher-level ones, GraphService explicitly accounts for variations in communication latencies across different communication domains. It ensures that the graph's locality adapts to the hierarchical communication domains of target HPC systems. Our key idea is to assign high-degree vertices and their clustered vertices to the same communication domain. That is because there are frequent communications among high-degree vertices, and placing high-degree vertices within the same domain can significantly improve locality and advance performance.

5.3 Vertex Clustering

Like most graph partitioning methods [12, 19, 27, 29, 73], we begin by extracting the spatial locality of subgraphs from the input graph. This can be achieved by multiple methods, such as XTree [29], core subgraph [12], and vertex clustering. In this work, we opt for vertex clustering due to its low time and space complexity, but our partitioning algorithm (detailed in Section 5.4) can work with other methods as well. Vertex clustering is based on an observation that in real-world graphs, only a small number of vertices have a high number of edges. These high-degree vertices and their reachable neighbors will likely be frequently accessed together. Our method differs from previous works in that we strive to position vertices in the same cluster nearby within the communication domains, thereby minimizing communication overhead.

In the graph preprocessing phase, we sort graph vertices by edge degree and group them into a "vertex cluster" using Algorithm 1. Each cluster is centered around a designated "center vertex",

ALGORITHM 1: Graph Vertex Clustering Algorithm

Input: vertex s and clustering distance threshold h
Output: $\mathfrak{C}$: a neighboring graph vertex cluster for v_0

```
1  Function Clustering(v_0, h)
2      if (h < 1) then
3          return C;
4      end
5      if !visited[v_0] then
6          C ← C ∪ {v_0};
7          visited[v_0] = 1;
8      end
9      for each u ∈ adjacent(v_0) do
10         Clustering(u, h − 1);
11     end
12     return C
13 end
```

v_0, with members consisting of vertices reachable from v_0 within a given number of hops. The iterative clustering process starts by selecting the highest-degree unvisited vertex as v_0 and adding its unvisited neighbors to the $\mathfrak{C}$ cluster. We then recursively add unvisited vertices at the next level to $\mathfrak{C}$ until the distance threshold, h, is less than 1 (line 2). After generating clusters for a given v_0, we repeat the process and move on to the next unvisited vertex with the highest edge degree. We continue until all vertices have been visited at least once, except for isolated ones with no neighbor.

We generate up to three clusters for each unvisited center vertex, v_0, with distance thresholds of 1-hop, 2-hop, and 3-hop, though users can adjust this threshold. Each cluster contains vertices at a specific distance from v_0, with smaller thresholds resulting in stronger vertex locality around v_0. Because only a small fraction of vertices have a high edge degree and we only add unvisited vertices to a new cluster, we generate a small number of vertex clusters, typically less than 15.

Essentially, we use the distance threshold, h, to determine the proximity of adjacent vertices to the center vertex, v_0. We start by considering vertices at a distance of 1 from v_0, and form a neighboring cluster consisting of these vertices. We then consider vertices at a distance of 2 from v_0 to create a new cluster. We continue this process, forming clusters for vertices at increasing distances from v_0 until all vertices have been assigned to a cluster. Once we have formed the vertex clusters, we assign them to communication domains based on the center vertex's edge degree and the aggregated edge degree of the cluster by starting a cluster whose center vertex has the highest degree.

5.4 Topology-Aware Graph Partitioning

Algorithm 2 outlines partitioning and distributing graph vertices to computing nodes. The algorithm operates on a list of allocated computing nodes, denoted by $\mathbb{N}$, which contains the node IDs. The distribution algorithm considers the locality of graph and communication differences across communication hierarchies.

Our partitioning algorithm (Algorithm 2) first builds a communication hierarchy for the target hierarchical systems, grouping computing nodes into communication domains according to the interconnection hierarchy of the target HPC systems and getting the total levels of communication hierarchy (lines 1-5). Next, we remove isolated vertices from the vertex list ($\mathbf{V}$) that lack connections to other vertices, storing them in a separate list $\tilde{\mathbf{V}}$. The remaining vertices in $\mathbf{V}$ are distributed among computing nodes using the `Partitioning` function (lines 9-12). The `Partitioning` function selects the vertex in $\mathbf{V}$ with the highest degree and utilizes the given vertex clusters (explained

ALGORITHM 2: Communication Hierarchy-Aware Partitioning Algorithm

Input: Sorted vertex list, V, a list of computing nodes, $\mathbb{N}$, clustering distance threshold, h

```
// Build communication tree and return tree height
```

1 Organize computing nodes into communication tree with hierarchical communication domains according to the communication topology of target systems;
2 H← $hierarchy(\mathbb{N})$
3 i = H
4 **while** $i \geq 0$ **do**
5 grouping the communication domains as $\mathcal{D}^{(H-i)}$ from bottom to up according to Figure 2;
6 i=i-1;
7 **end**
8 Move isolated vertices from **V** to $\tilde{\mathbf{V}}$
9 Set all vertices to be unvisited
10 **while** $!empty(V)$ **do**
`// Remove the highest degree vertex`
11 $v \leftarrow V.dequeue()$
12 **Partitioning**(v)
13 **end**
14 **Function** `Partitioning(`*vertex* v`)`
15 $\mathfrak{C}_v$=[]
16 **for** $(i = 1; i \leq h; i++)$ **do**
`// Calling Algorithm 1.`
17 $\mathfrak{C}_v \leftarrow$ **Clustering**(v, i)
18 **end**
19 **while** $!empty(\mathfrak{C}_v)$ **do**
20 $i = 0$;
21 **while** $i \leq H$ **do**
22 $\mathfrak{C}_{max} \leftarrow \max(\mathfrak{C}_v)$
`// Recursively distribute vertices to the communication tree.`
23 share $\mathfrak{C}_{max}$ among nodes attached to $D^{(i)}$ evenly; i=i+1;
24 **end**
25 $\mathfrak{C} \leftarrow$ remaining reachable vertices from v with a distance $> h$
26 distribute vertices in $\mathfrak{C}$ into all nodes attached to D^0 evenly
27 **end**
28 **end**

in Section 5.3) to group nearby vertices. Vertices within a vertex cluster $\mathfrak{C}$ are assigned to computing nodes recursively to adapt to the target communication hierarchies. This method prioritizes node placement within the same communication domain or domains at the same level. To partition the graph based on vertex distance, we set a threshold, denoted as h (see Section 5.3). We then generate a list of h clusters, $\mathfrak{C}_v = \mathfrak{C}_v^1, \mathfrak{C}_v^2, \ldots, \mathfrak{C}_v^h$, where each cluster $\mathfrak{C}_v^i$ corresponds to a distance of i $(1 \leq i \leq h)$ from the highest-degree vertex v that has not been processed yet (lines 25-27). When distributing vertices in each cluster to computing nodes, GraphService recursively starts from the lowest available level of the communication hierarchy and moves to a higher level only if the current level's resources cannot hold all vertices in $\mathfrak{C}$.

5.5 Topology-Aware Graph Compression

GraphService builds upon the classical CSR format but employs a folding method to group vertices with the same degree into a single starting offset in the RST array. GraphService will classify the initial RST array into two parts with a key parameter Thr, which is a threshold of vertex degree.

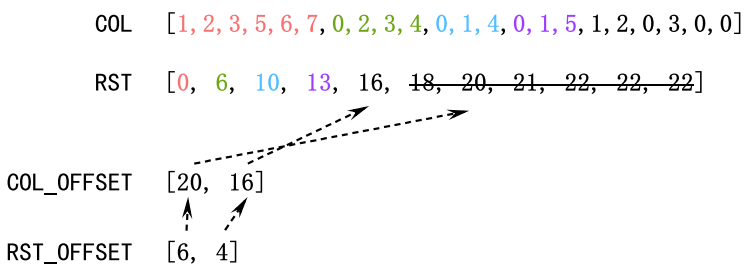


Fig. 7. Using GraphService (Thr = 2) to represent the graph adjacency matrix of Figure 3(b).

Vertices located before and after Thr will have specified indexing methods separately. While simple, this folding scheme is effective for graph processing by batching identical-degree vertices, thereby coalescing memory accesses and further enhancing graph construction speed in large-scale graphs.

5.6 GraphService Sparse Format

To facilitate GraphService, we tailor CSR to enhance large-scale graph storage, which introduces two additional arrays, RST_OFFSET and COL_OFFSET, and a hyperparameter, denoted as Thr. Thr is a threshold that determines whether a vertex is low degree and should be folded. Vertices in RST are classified into two parts by Thr; thus, only a small number of vertices with degrees exceeding Thr are stored and indexed similarly to standard CSR. Otherwise, vertices with degrees no more than Thr are expressed in RST_OFFSET and COL_OFFSET. The RST_OFFSET array stores the minimal IDs of vertices with edge degrees between 1 to Thr while the COL_OFFSET array stores the COL offset values with respect to the vertices in RST_OFFSET. In this way, the low-degree vertices can be traced through the RST_OFFSET and COL_OFFSET arrays. For high-degree vertices whose edge degrees are great than Thr, we store them in the standard CSR RST and COL arrays. Note that we opt not to compress high-degree vertices in GraphService. This decision stems from the observation that high-degree vertices are typically relatively rare in real-world graphs and are often accessed frequently during graph processing [12, 27, 29, 40, 73]. Therefore, compressing them may incur additional runtime overhead, which could outweigh the benefits of compression.

5.7 Graph Storage

We use Figure 7 to illustrate how GraphService represents a graph adjacency matrix for the example given earlier in Figure 3. For illustration, we set the edge degree threshold parameter, Thr, to 2. This means that vertices with edge degrees greater than 2 are considered high-degree vertices and will be stored in standard CSR, whereas vertices with degrees equal to or less than 2 are considered low-degree vertices and will be stored in the RST_OFFSET and COL_OFFSET arrays by referring to COL.

Algorithm 3 outlines how GraphService encodes high-degree and low-degree vertices. Specifically, the RST_OFFSET array stores the minimal IDs of N-degree vertices for $N \in [1, Thr]$, with the edge degree ordered in an ascending manner. For example, in Figure 7, RST_OFFSET[0] records the minimal ID of vertices with degree = 1 among vertices with ID = 6 and ID = 7, storing only one vertex with ID = 6 for all 1-degree vertices. Similarly, RST_OFFSET[1] records the minimal ID of vertices with degree = 2 (i.e., degree = Thr) among vertices with ID = 4 and ID = 5, storing only one vertex with ID = 4 for all 2-degree vertices. For others in the RST_OFFSET array, the same rule applies. Correspondingly, COL_OFFSET[0] records the starting offset in COL for the vertex with minimal ID among 1-degree vertices (i.e., the vertex with ID = 6), whereas COL_OFFSET[1] records the starting offset in COL for the vertex with minimal ID among 2-degree vertices and, thus, a

Table 3. Synthetic/Real Graph Data Used in Our Evaluation

G. Scale	Edge factor	#Vertices	#Edges	#Comp. Nodes
Real-life graphs				
Kron-26	16	64 M	1 B	1
Kron-28	16	256 M	4 B	4
Kron-30	16	1 B	16 B	16
Kron-32	16	4 B	64 B	64
Kron-34	16	16 B	256 B	256
Kron-36	16	64 B	1 Tri	1,024
Kron-37	16	128 B	2 Tri	2,048
Kron-38	16	256 B	4 Tri	4,096
Kron-41	16	2 Tri	32 Tri	79,024
	clueweb12 [3]	987 M	42.6 B	16
	twitter2010 [2]	4.2 M	1.5 B	16
	USA road [5]	23.9 M	58.8 M	16

ALGORITHM 3: GraphService Constructing Algorithm

Input: RST of standard CSR
Thr // threshold of vertex degree
Output: RST_OFFSET
COL_OFFSET

```
1 for v_c ∈ RST in parallel do
2     degree = RST [v_c+1] - RST [v_c]
3     if 0 < degree ≤ Thr then
4         if v_c ≤ RST_OFFSET[degree-1] then
5             RST_OFFSET[degree-1]=v_c
6             COL_OFFSET[degree-1]=RST[v_c]
7
```

successive adjacent vertex set {1, 2} for the vertex with ID = 4, and {0, 3} for the next 2-degree vertex with ID = 5, until all 2-degree vertices would be directly accessed without repeated calculations. In contrast, high-degree vertices with degrees greater than Thr are stored in a standard CSR. Overall, this approach optimizes the storage of low-degree vertices while still allowing high-degree vertices to be stored using the standard CSR approach. To this end, GraphService not only saves space but also enables batch memory access since it merely needs one entry for many identical-degree vertices that are continuously stored in a sorted graph.

6 Experimental Evaluation

6.1 Experimental Workloads

Our main evaluation is performed on the BFS algorithm defined in the Graph500 benchmark [9]. Graph500 is the *de facto* standard for assessing a computer system's capability for graph processing [27, 39, 40, 47, 60]. It provides a graph generator to generate synthetic graphs that mimic real-world graph structures. This tool takes two parameters, a graph factor, and an $edge\_factor$. Given a graph size m and an $edge\_factor$ n, it generates a graph (i.e., $Kron-m$) of 2^m vertices and $n \times 2^m$ edges. Unless stated otherwise, we use the Graph500 default $edge\_factor$ of 16. For our evaluation, we vary the graph factor between 26 and 41 to generate graphs of different scales for testing our approach on various hardware setups. Table 3 lists the synthetic graphs and real-world graphs used in our evaluation.

Table 4. Hardware Systems Used in Our Evaluation

System	Max. #nodes used	CPU	RAM per node	Top-level bandwidth
Tianhe-Exa	79,024	16-core FT-2000 ARMv8 CPU @ 2.2 GHz	16 G	200 Gbps
WuzhenLight	1,024	64-core HG2 7285H (AMD x86 ISA) CPU @ 2.5 GHz	256 G	100 Gbps
Intel Cluster	512	12-core Intel Xeon CPU @ 2.93 GHz	64 G	160 Gbps

In addition to the synthetic graph data generated by Graph500, we also evaluate GraphService on two public real-world graphs [2, 3], including clueweb12 (with 987 million vertices and 42.6 billion edges) [3], and twitter-2010 (with 41.7 million vertices and 1.47 billion edges) [2]. We also assess other graph processing operations on these datasets, including SSSP, PR, CC, BC, and TC.

6.2 Evaluation Platforms

To evaluate the portability of GraphService, we apply it to three cutting-edge systems with different CPU architectures using 512 nodes to 79,024 nodes. Table 4 lists the three HPC systems used in our testing and the maximum number of computing nodes. Each node on the WuzhenLight has two HG2 32-core CPUs at 2.5 GHz that are compatible with the AMD x64 instruction set. Each node on Tianhe-Exa has a Phytium 16-core CPU at 2.0 GHz. The last one is configured with a 512-node Intel Xeon CPU at 2.93 GHz. These three systems run a customized Linux operating system with Linux kernel v9.3.0. We use MPICH 10.2.0 for the MPI and libgomp 4.5 for OpenMP and compile the benchmark using GCC 10.2.0 with "-O3" as the compiler option.

6.3 Competing Baselines

We compare GraphService to five representative graph partitioners: 2D decomposition [29], LDG [50], Par-METIS (the parallel version of METIS [32]), CLUGP [34, 71], and TopoX [70] for validating the communication hierarchy-aware partitioner. Moreover, we compare GraphService to six state-of-the-art sparse storage formats: native CSR, DCSR [11], COO (Coordinate list) [17, 37, 41], CSCSR (Coarse index + Skip list) [14], BCSR (Bitmap-based sparse matrix representation) [61], and CSR5 [41]. We also compare GraphService against Gemini [73] and GraphScope [19], two state-of-the-art graph processing engines, using the engineer-tuned algorithm implementations provided by these frameworks.

6.4 Benchmarking Graph500

We deployed the full implementation of GraphService to benchmark Graph500 BFS and SSSP on Tianhe-Exa. In our experiments, we used 79,024 computing nodes (1,264,384 cores) for BFS and 8,192 nodes (131,072 cores) for SSSP. Our implementation and evaluation fully comply with the Graph500 specification.

The latest Graph500 ranking, published in November 2023, places Fugaku and Wuhan Supercomputer as the top performers for BFS and SSSP, respectively. However, GraphService on Tianhe-Exa successfully outperforms these top-ranking systems for both benchmarks with three orders of magnitude lower construction time, as listed in Table 5. This notable improvement can be attributed to the communication hierarchy-aware partitioner and space-time-efficient graph compression techniques integrated into GraphService, which significantly enhances the performance of BFS and SSSP.

6.5 Compare with Baseline Partitioners

Figure 8 reports the time spent on constructing the graph. Generally, as the size of the graph and the number of computing nodes increases, the construction time also grows. However, we observe that GraphService has the lowest overhead compared with other baselines. In contrast, LDG, which

Table 5. The Latest Graph500 BFS/SSSP Ranking with Construction Time — C_Time (/second)

#Rank	BFS/SSSP	System	GTEPS	C_Time
1	BFS	Supercomputer Fugaku	138,867	2,388.66
–	BFS	Tianhe-Exa	162,494	22.61
1	SSSP	Wuhan Supercomputer	15,335.9	16,285
–	SSSP	Tianhe-Exa	23,021	13.37

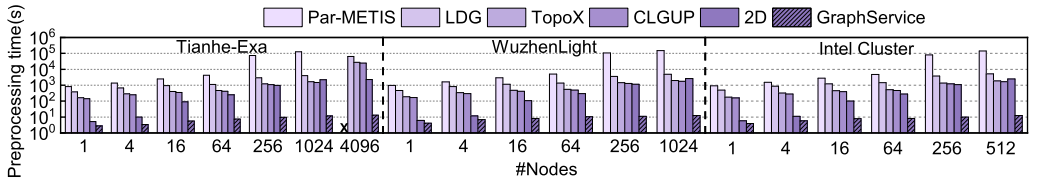


Fig. 8. Construction time before BFS ingress (***lower is better***).

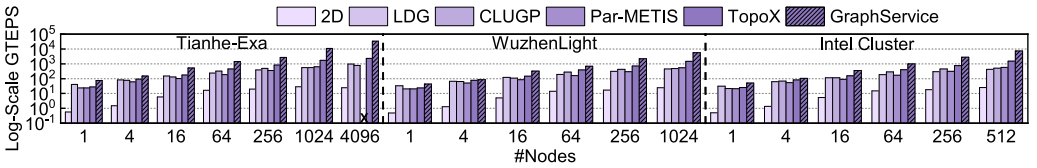
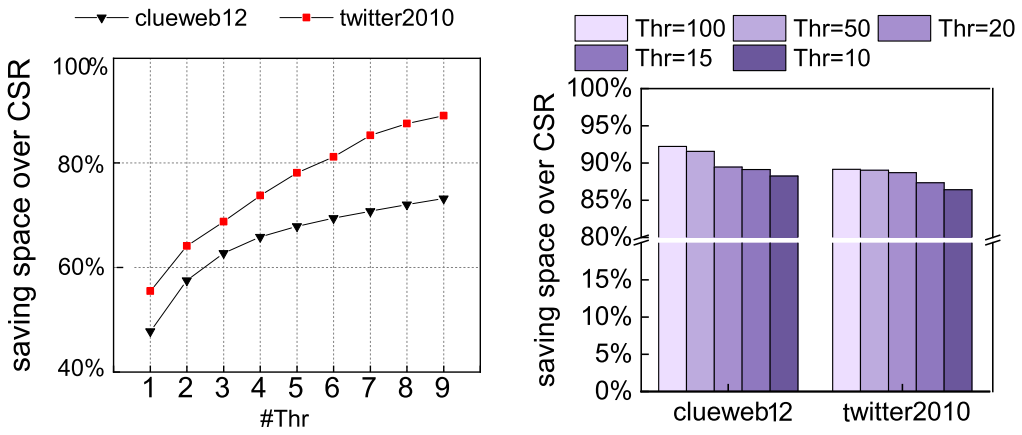


Fig. 9. BFS throughput given by different partitioners (***higher is better***).

requires significant refactoring of the input graph, incurs 4, 971.83× longer construction time with the 4,096-node available on Tianhe-Exa than that of GraphService. Note that (i) ParMETIS is a general partitioner without taking the structure of the graph into account, resulting in poor performance in processing large-scale graphs. (ii) More importantly, communication time will become the main bottleneck of the graph processing but existing partitioners (such as ParMETIS) fail to exploit communication hierarchies at large scales, which suggests that developing a topology-aware GraphService is crucial to scale graph processing.

Figure 9 compares the throughput of GraphService to five graph partitioning methods. The experiment used up to 4,096 Tianhe-Exa nodes to execute BFS. Some methods led to a runtime error (marked as **X**). GraphService outperforms all baselines, particularly as the number of computing nodes increases. For instance, when processing a graph scale of 38 using 4,096 Tianhe-Exa nodes, GraphService delivers 33,490.17 GTEPS, 9.7× and 28.7× improvements over TopoX and CLUGP, respectively. We also obtain similar results on SSSP, PR, CC, and CDLP, where GraphService respectively gives 27.2×, 29.1×, 25.6×, and 19.7× throughput improvements over the best-performing baseline when using 4,096 Tianhe-Exa nodes. This is because GraphService significantly enhances graph distribution, making subsequent communication during graph computation more regular and expediting graph tasks. Note that, while GraphService is motivated by Fat-tree, the underlying methodologies of our work have been generalized across different interconnect topologies, such as 3-D Torus, and Dragonfly. Both are respectively equipped in WuzhenLight and Intel Cluster, and evaluated using GraphService.



(a) Thr from 1 to 9 (b) Thr from 10 to 100

Fig. 10. Sensibility of *Thr* in graph clueweb12 and twitter2010, in which saving space = $\frac{|M(CSR)-M(SuperCSR)|}{M(CSR)} \times 100\%$, M(X) represents the memory cost of format X (i.e., CSR or GraphService).

6.6 Tuning *Thr* for Graph Compression

In this subsection, we will take the real-world graphs, including clubweb12 and twitter2010, to examine the impact of the hyperparameter *Thr* on the performance of GraphService. In addition, by digging deeper, we demonstrate how to further fine-tune the *Thr* for fast graph processing.

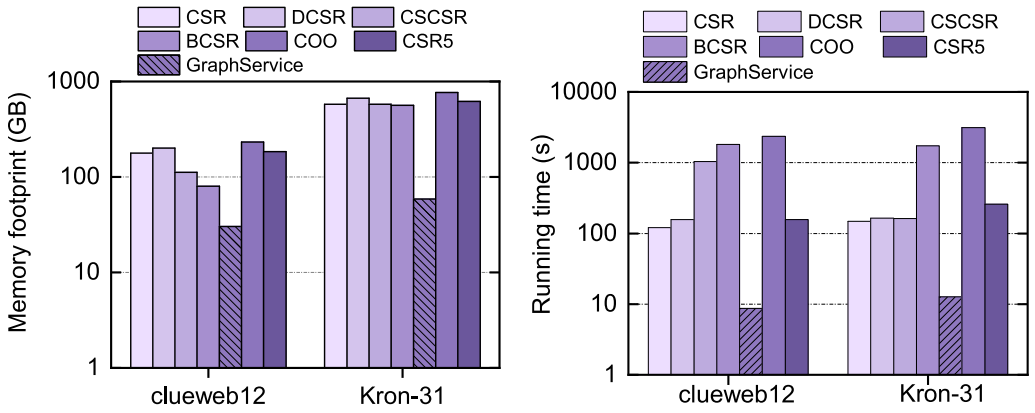
The selection policy for *Thr* should be highly based on the graph's vertex distribution. As we show in Figure 6, low-degree vertices account for a high fraction in the real-world graphs, whereas DCSR-mentioned hypersparse graphs are uncommon. Thus, our insight is to make *Thr* cover most of the low-degree vertices. For example, vertices with $degree \in [1, 9]$ hold more than 78% for the provided real-world graphs (see Figure 6). We highly recommend users set up their own *Thr* range and evaluate the sensibility of *Thr* across different scales of graphs according to the graph degrees' distribution as shown in Figure 10.

We evaluate the GraphService's performance by carefully tuning the $Thr \in \{10, 15, 20, 25\}$. We also list the results of $Thr \leq 9$ (see Figure 10(a)) to prove that based on the degree distribution in Figure 6, every increase in *Thr* brings obvious benefits, and the overall yield is linear. On the other hand, Figure 10(b) shows that when $Thr > 9$, further changes in *Thr* have little effect on its performance with the same graph. The largest performance gap would be around 5% when we conduct different *Thr* on clueweb12 and twitter2010. So far, we may draw the following conclusions.

(i) Majorities of the graphs have a large scale of N-degree vertices, such that $N \leq 10$. In this case, $Thr = 9$ gains significant benefits. It is strongly recommended that, prior to the meticulous adjustment of the *Thr*, researchers should refer to the graph's degree distribution.
(ii) Although a larger *Thr* may give a better GraphService performance, GraphService is overall *Thr*-oblivious when $Thr > 10$.

6.7 Preprocessing Overhead

Owing to requiring a sorted graph as input, GraphService includes a built-in sorting module configured to accommodate various types of input graphs without the need for manual sorting beforehand. Next, GraphService performs a further reindexing of each vertex by incorporating the RST and COL array, which incurs preprocessing overhead. This preprocessing is a *one-off* cost and employed by many famous graph systems [16, 18, 19, 27, 29, 30, 40, 73]. Experiments show that



(a) Memory cost (b) Constructioon time

Fig. 11. The memory and construction time of GraphService against other sparse formats when running BFS on graphs.

the overhead of GraphService preprocessing will get slightly higher but still acceptable when the graph gets bigger (2.07 s while scaling to 512 nodes). We have noticed that many great works of vertex sorting in graph-parallel processing systems have been provided, such as [15, 19, 27, 30, 40, 64, 73]. GraphService is developed for better servicing graph applications and can be integrated seamlessly with existing graph preprocessing approaches, potentially gaining additional benefits from them.

6.8 Compare with Other Sparse Storage Formats

Figure 11(a) reports the memory footprint comparison across different sparse formats for BFS. GraphService outperforms all the other CSR-like formats, saving more than 90% memory space over most of the CSRs, especially up to 99.8% of space against the CSCSR.

We also evaluate GraphService on a real-world social graph clueweb12 [3] and report both the memory usage and runtime of BFS in Figure 11. Figure 11(a) shows that GraphService has the smallest memory cost over other storage formats (saving average memory). In Figure 11(b), GraphService also shows the fastest runtime against others (yielding average runtime speedup).

6.9 Scalability

In this section, we evaluate the scalability of GraphService by applying it to BFS running with different numbers of nodes on Tianhe-Exa [72], WuzhenLight [4], and Intel cluster listed in Table 4. Figure 12 reports how the normalized GTEPS changes as we increase the number of computing nodes, with one single node serving as the normalization baseline. We observe a consistent increase in GTEPS for GraphService as the number of computing nodes increases, suggesting that GraphService-based BFS exhibits good scalability.

6.10 GraphService for Real-World Graphs

We conduct experiments with GraphService on large real-world graphs using 64 Tianhe-Exa nodes across four communication domains. Figure 13 compares GraphService with Gemini and GraphScope, both of which offer engineer-optimized implementations for the test algorithms. Gemini could not execute some test cases (marked as **X**) and does not support CDLP. GraphService's partitioning approach leads to shorter preprocessing times than Gemini and GraphScope,

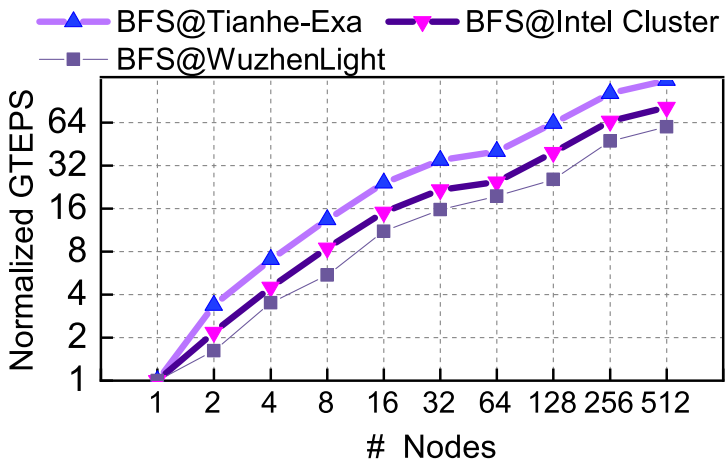
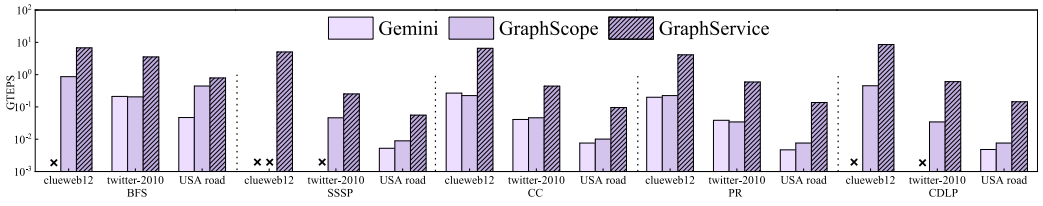
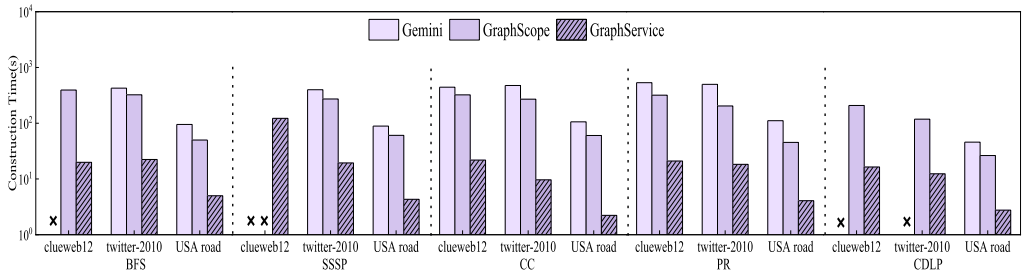


Fig. 12. GraphService's scalability on various HPC systems.



(a) Throughput



(b) Construction time (***lower-is-better***)

Fig. 13. Performance comparison on real-world graph and typical graph operators using 64-node on Tianhe-Exa.

which need graph repartitioning for load balancing [19, 73]. GraphService consistently outperforms Gemini and GraphScope in all test cases during the graph computation stage, achieving a speedup of up to 18.92× over GraphScope.

7 Conclusion and Future Work

We have presented GraphService, a groundbreaking framework tailored for constructing large-scale graph applications. GraphService revolutionizes performance by innovating hierarchical communication topologies and strategically grouping vertices based on identical degrees within the graph structure. This approach positions GraphService as the leading solution and significantly reduces construction time before graph algorithm execution.

Extensive evaluations affirm that GraphService consistently surpasses baseline methodologies, delivering unparalleled graph processing throughput. Moreover, it achieves a remarkable reduction in graph construction time by an astounding factor of 4,971.83, thus setting a new standard for efficiency in graph application development. Looking ahead, GraphService promises to redefine the landscape of graph computing. Future iterations will focus on enhancing scalability, integrating advanced optimization techniques, and expanding compatibility with diverse graph structures. By continuing to push the boundaries of performance and usability, GraphService is poised to empower developers and researchers alike with unprecedented capabilities in managing and analyzing large-scale graphs.

References

[1] Graph500. 2021. Retrieved from https://graph500.org/

[2] Laboratory for Web Algorithmics. 2022. twitter-2020. Retrieved from https://law.di.unimi.it/webdata/twitter-2010/

[3] The Lemur Project. The ClueWeb12 Dataset. 2022. Retrieved from https://lemurproject.org/clueweb12/

[4] 2022. Retrieved from https://www.laitimes.com/en/article/85ga_86m5.html

[5] DIMACS. 2010. 9th DIMACS Implementation Challenge – Shortest Paths. http://www.diag.uniroma1.it//challenge9/download.shtml

[6] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: An experimental study. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1590–1603.

[7] Arvind Arasu, Jasmine Novak, Andrew Tomkins, and John Tomlin. 2002. PageRank computation and the structure of the web: Experiments and algorithms. In *Proceedings of the 11th International World Wide Web Conference, Poster Track*. 107–117.

[8] Nicolas Aspert, Volodymyr Miz, Benjamin Ricaud, and Pierre Vandergheynst. 2019. A graph-structured dataset for Wikipedia research. In *Companion Proceedings of the 2019 World Wide Web Conference*. 1188–1193.

[9] Graph500 benchmark BFS. 2023. The Graph 500 List. Retrieved November 3, 2023 from https://graph500.org/?page_id=1240

[10] M. Blocksome, C. Archer, T. Inglett, P. McCarthy, M. Mundy, J. Ratterman, A. Sidelnik, B. Smith, G. Almási, J. Castanos, D. Lieber, J. Moreira, S. Krishnamoorthy, V. Tipparaju, and J. Nieplocha. 2006. Design and implementation of a one-sided communication interface for the IBM eServer blue gene. *ACM/IEEE SC 2006 Conference (SC'06)* (2006), 54–54.

[11] 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.

[12] 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.

[13] Zhenhua Chang, Ding Ding, and Youhao Xia. 2021. A graph-based QoS prediction approach for web service recommendation. *Applied Intelligence* (2021), 1–15.

[14] 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.

[15] R. Chen, J. Shi, Y. Chen, and H. Chen. 2015. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *European Conference on Computer Systems* (2015), 1–15.

[16] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[17] 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.

[18] 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.

[19] 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.

[20] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1261–1274.

[21] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels. 2009. MPI collective communications on the blue Gene/P supercomputer: Algorithms and optimizations. In *2009 17th IEEE Symposium on High Performance Interconnects*. 63–72. https://doi.org/10.1109/HOTI.2009.12

[22] Per Fuchs, Domagoj Margan, and Jana Giceva. 2023. Sortledton: A universal graph data structure. *ACM SIGMOD Record* 52, 1 (2023), 17–25.

[23] Pablo Fuentes, Mariano Benito, Enrique Vallejo, José Luis Bosque, Ramón Beivide, Andreea Anghel, Germán Rodríguez, Mitch Gusat, Cyriel Minkenberg, and Mateo Valero. 2017. A scalable synthetic traffic model of Graph500 for computer networks analysis. *Concurrency and Computation: Practice and Experience* 29, 24 (2017), e4231.

[24] Xinbiao Gan, Jiaqi Guo, Peilin Guo, Guang Wu, Jiaqi Si, Songzhu Mei, Cong Liu, and Tiejun Li. 2023. GraphMedia: Communication-balanced graph searching for billion-scale social media access. In *Proceedings of the 31st ACM International Conference on Multimedia*. 8984–8993.

[25] Xinbiao Gan, Guang Wu, Cong Liu, Jiaqi Si, Xuguang Chen, Bo Yang, and Tiejun Li. 2022. TianheQueries: Ultra-fast and scalable graph queries on Tianhe supercomputer. In *2022 IEEE 24th International Conference on High Performance Computing & Communications; 8th International Conference on Data Science & Systems; 20th International Conference on Smart City; 8th International Conference on Dependability in Sensor, Cloud & Big Data Systems & Application (HPC-C/DSS/SmartCity/DependSys)*. IEEE, 1153–1158.

[26] 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.

[27] 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.

[28] 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* (2021), 1–1. https://doi.org/10.1109/TPDS.2021.31007852

[29] 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'22)*. IEEE, 2046–2059.

[30] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 17–30.

[31] Muhammad Imran, Gábor E. Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Fast datalog evaluation for batch and stream graph processing. *World Wide Web* 25, 2 (2022), 971–1003.

[32] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

[33] George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. Parmetis: Parallel graph partitioning and sparse matrix ordering library. (1997).

[34] Deyu Kong, Xike Xie, and Zhuoxu Zhang. 2022. Clustering-based partitioning for large web graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE'22)*. IEEE, 593–606.

[35] S. Kumar, A. Mamidala, Daniel Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burow. 2012. PAMI: A parallel active message interface for the blue Gene/Q supercomputer. *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), 763–773.

[36] Mingzhe Li, Xiaoyi Lu, S. Potluri, Khaled Hamidouche, J. Jose, K. Tomko, and D. Panda. 2014. Scalable Graph500 design with MPI-3 RMA. *2014 IEEE International Conference on Cluster Computing (CLUSTER)* (2014), 230–238.

[37] 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.

[38] Z. Li, C. Wu, and Y. Li. 2021. FEP-based large-scale virtual screening for effective drug discovery against COVID-19. In *International Conference on High Performance Computing, Networking, Storage, and Analysis*.

[39] 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'17)*. IEEE, 635–645.

[40] 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.

[41] 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.

[42] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2009. Pregel: A system for large-scale graph processing. *SIGMOD* (2009), 135–146.

[43] Ruben Mayer and Hans-Arno Jacobsen. 2021. Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints. In *Proceedings of the 2021 International Conference on Management of Data*. 1289–1302.
[44] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph structure in the web—revisited: A trick of the heavy tail. In *Proceedings of the 23rd International Conference on World Wide Web*. 427–432.
[45] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhisa Sato. [n.d.]. Performance of the supercomputer Fugaku for breadth-first search in Graph500 benchmark.([n. d.]).
[46] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and M. Sato. 2020. Performance evaluation of supercomputer Fugaku using breadth-first search benchmark in Graph500. *2020 IEEE International Conference on Cluster Computing (CLUSTER)* (2020), 408–409.
[47] 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'20)*. IEEE, 408–409.
[48] Masahiro Nakao, Koji Ueno, Katsuki Fujisawa, Yuetsu Kodama, and Mitsuhisa Sato. 2021. Performance of the supercomputer Fugaku for breadth-first search in Graph500 benchmark. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings 36*. Springer, 372–390.
[49] Md Nahid Newaz, Hua Ming, Sayan Ghosh, Joshua Suetterlein, and Nathan R. Tallent. [n.d.]. Simulating application agnostic process assignment for graph workloads on dragonfly and fat tree topologies. ([n. d.]).
[50] Joel Nishimura and Johan Ugander. 2013. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1106–1114.
[51] Anil Pacaci and M. Tamer Özsu. 2019. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*. 1375–1392.
[52] G. Shah, J. Nieplocha, J. Mirza, Chulho Kim, R. Harrison, R. Govindaraju, K. Gildea, Paul DiNicola, and C. A. Bender. 1998. Performance and experience with LAPI—a new high-performance communication library for the IBM RS/6000 SP. *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing* (1998), 260–266.
[53] Naoyuki Shida, S. Sumimoto, and Atsuya Uno. 2012. MPI library and low-level communication on the K computer.
[54] Hyogi Sim, Awais Khan, and Sudharshan S Vazhkudai. 2020. An analysis of system balance and architectural trends based on Top500 supercomputers. (2020).
[55] George M. Slota, Cameron Root, Karen Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. 2020. Scalable, multi-constraint, complex-objective graph partitioning. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2789–2801.
[56] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *Proceedings of the 11th ACM International Conference on Web Search and Data Mining*. 540–546.
[57] Tina Esther Trueman, P. Narayanasamy, and J. Ashok Kumar. 2022. A graph-based method for ranking of cloud service providers. *The Journal of Supercomputing* 78, 5 (2022), 7260–7277.
[58] Sotiris Tsioutsiouliklis, Evaggelia Pitoura, Panayiotis Tsaparas, Ilias Kleftakis, and Nikos Mamoulis. 2021. Fairness-aware PageRank. In *Proceedings of the Web Conference 2021*. 3815–3826.
[59] Koji Ueno and Toyotaro Suzumura. 2012. Highly scalable graph search for the Graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. 149–160.
[60] 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'16)*. IEEE, 1040–1047.
[61] 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.
[62] Ruibo Wang, Kai Lu, Juan Chen, Wenzhe Zhang, Jinwen Li, Yuan Yuan, Pingjing Lu, Libo Huang, Shengguo Li, and Xiaokang Fan. 2020. Brief introduction of TianHe exascale prototype system. *Tsinghua Science and Technology* 26, 3 (2020), 361–369.
[63] Yingheng Wang, Yaosen Min, Xin Chen, and Ji Wu. 2021. Multi-view graph contrastive representation learning for drug-drug interaction prediction. In *Proceedings of the Web Conference 2021*. 2921–2933.
[64] Wikipedia. 2021. Fugaku (Supercomputer). Retrieved September 20, 2021 from https://en.wikipedia.org/wiki/Fugaku_(supercomputer)
[65] Jeremiah J. Wilke and Joseph P. Kenny. 2020. *Opportunities and Limitations of Quality-of-Service (QoS) in Message Passing (MPI) Applications on Adaptively Routed Dragonfly and Fat Tree Networks*. Technical Report. Sandia National Lab (SNL-CA), Livermore, CA.

[66] 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.
[67] Dongjin Yu, Yu Liu, Yueshen Xu, and Yuyu Yin. 2014. Personalized QoS prediction for web services using latent factor models. In *2014 IEEE International Conference on Services Computing*. IEEE, 107–114.
[68] H. Yu, I. Chung, and J. Moreira. 2006. Topology mapping for blue Gene/L supercomputer. *ACM/IEEE SC 2006 Conference (SC'06)* (2006), 52–52.
[69] Yiming Zhang, Kai Lu, and Wenguang Chen. 2021. Processing extreme-scale graphs on China's supercomputers. *Commun. ACM* 64, 11 (2021), 60–63.
[70] Yiming Zhang, Haonan Wang, Menghan Jia, Jinyan Wang, Dong sheng Li, Guangtao Xue, and K. Tan. 2020. TopoX: Topology refactorization for minimizing network communication in graph computations. *IEEE/ACM Transactions on Networking* 28 (2020), 2768–2782.
[71] Zhuoxu Zhang and Zezhong Ding. 2022. Streaming graph clustering for graph partition. In *2022 IEEE 5th International Conference on Automation, Electronics and Electrical Engineering (AUTEEE'22)*. IEEE, 880–884.
[72] Li Zhe, Wu Chengkun, Li Yishui, et al. 2021. FEP-based large-scale virtual screening for effective drug discovery against COVID-19 [J/OL]. (2021).
[73] 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, November 2–4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

Received 16 May 2024; revised 22 July 2024; accepted 8 August 2024