1-2019

GraphMP: I/O-Efficient big graph analytics on a single commodity machine

Peng SUN

Yonggang WEN

Nguyen Binh Duong TA
Singapore Management University, donta@smu.edu.sg

Xiaokui XIAO

GraphMP: I/O-Efficient Big Graph Analytics on a Single Commodity Machine

Peng Sun, Yonggang Wen, Ta Nguyen Binh Duong and Xiaokui Xiao

Abstract—Recent studies showed that single-machine graph processing systems can be as highly competitive as cluster-based approaches on large-scale problems. While several out-of-core graph processing systems and computation models have been proposed, the high disk I/O overhead could significantly reduce performance in many practical cases. In this paper, we propose GraphMP to tackle big graph analytics on a single machine. GraphMP achieves low disk I/O overhead with three techniques. First, we design a vertex-centric sliding window (VSW) computation model to avoid reading and writing vertices on disk. Second, we propose a selective scheduling method to skip loading and processing unnecessary edge shards on disk. Third, we use a compressed edge cache mechanism to fully utilize the available memory of a machine to reduce the amount of disk accesses for edges. Extensive evaluations have shown that GraphMP could outperform existing single-machine out-of-core systems such as GraphChi, X-Stream and GridGraph by up to 30, and can be as highly competitive as distributed graph engines like Pregel+, PowerGraph and Chaos.

Index Terms—Graph Processing, Big Data, Parallel Computing, Vertex-Centric Programming Model

1 Introduction

In the era of "Big Data", many real-world problems, such as social network analytics and collaborative recommendation, can be represented as graph computing problems [1]. Analyzing large-scale graphs has attracted considerable interest in both academia and industry. However, researchers are facing significant challenges in processing big graphs, which contain billions of vertices and hundreds of billions of edges, with popular big data analysis tools like MapReduce [2] and Spark [3], since these general-purpose frameworks cannot leverage inherent interdependencies within graph data and common patterns of iterative graph algorithms for performance optimization [4], [5], [6].

To tackle this challenge, researchers have proposed many dedicated in-memory graph processing systems over multi-core, heterogeneous and distributed infrastructures. These systems usually adopt a vertex-centric programming model (which allows users to think like a vertex when designing parallel graph applications), and should always manage the entire input graph and all intermediate data in memory. Specifically, Ligra [7], Galois [8], GraphMat [9] and Polymer [10] could handle generic graphs with 1-20 billion edges on a single multi-core machine. Some single-machine systems, e.g., [11], [12], [13], [14], [15], [16], [17], [18], [19], could leverage heterogeneous devices, such as graphics processing unit (GPU), field-programmable gate array (FPGA) and Xeon Phi, to scale up graph processing performance.

To process big graphs, which cannot be fully loaded into the memory of a single commodity machine, three types of distributed graph engines could scale out in-memory graph processing to a cluster:

- Pregel-like systems, e.g., [20], [21], [22], [23], [24], assign each vertex and its out-going edges to a machine, and provide interaction between vertices using message passing along edges.
- PowerGraph [25], PowerLyra [26] and GraphX [27] adopt the GAS (Gather-Apply-Scatter) model to improve load balance when processing power-law graphs: they split a vertex into multiple replicas, and parallelize the computation for it on different machines.
- GraphPad [28] and CombBLAS [29] express common graph analyses in generalized sparse matrix-vector multiplication (SpMV) operations, and leverage high-performance computing (HPC) techniques to speed up large-scale SpMV.

However, current in-memory graph processing systems require a costly investment in powerful computing infrastructure to handle big graphs. For example, GraphX needs more than 16TB memory to handle a 10-billion-edge graph [30].

Out-of-core systems, which maintain just a small portion of vertices and/or edges in memory, provide cost-effective solutions for big graph analytics. Single-machine engines, such as GraphChi [31], X-Stream [32], VENUS [33] and GridGraph [34], break the input graph into a set of shards, each of which contains all required information to update its associated vertices. In many cases, an out-of-core graph engine processes all shards in an iteration, and usually uses three stages to execute a shard:

- Loading this shard's associated vertices into memory;
- Processing this shard's edges from disk for updating its associated vertices;
- Writing updated vertices or edges back to disk.

These three stages would generate a huge amount of disk accesses, which may become performance bottleneck [4]. To reduce disk I/O cost, many out-of-core graph computation models have been proposed. Representative examples include the parallel sliding window model (PSW) of

- *Peng Sun, Yonggang Wen and Ta Nguyen Binh Duong are with School of Computer Science and Engineering, Nanyang Technological University, Singapore. Email: {sunp0003, ygwen, donta}@ntu.edu.sg*
- *Xiaokui Xiao is with School of Computing, National University of Singapore, Singapore. Email: xkxiao@nus.edu.sg*

TABLE 1
Existing approaches for large-scale graph processing.

	Single Machine (CPU)				Single Machine (GPU)		Cluster	
Data Storage	**In-Memory**	**Out-of-Core (use HDD if not indicated)**			**In-Memory**	**Out-of-Core**	**In-Memory**	**Out-of-Core**
Approaches	Ligra [7] Galois [8] GraphMat [9] Polymer [10]	GraphChi [31] X-Stream [32] VENUS [33] GridGraph [34]	**GraphMP**	(Use SSD) FlashGraph [35] TurboGraph [36]	Medusa [11] Gunrock [13] MapGraph [14] gGraph [15]	(Use SSD) GTS [16] GGraph [15]	Pregel-like: [20] [21] [22] [23] [24] GAS: [25] [26] [27] SpMV: [28] [29]	(Use HDD) GraphD [37] Chaos [38] Pregelix [39]
Scale (#edges)	1-20 Billion	∼100 Billion	∼100 Billion	∼100 Billion	0.1-4 Billion	4-64 Billion	5-1000 Billion	∼1 Trillion
Speed (#edges/s)	1-2 Billion	5-100 Million	20M-2.2B	20-400 Million	1-7 Billion	∼0.4 Billion	1-7 Billion	5-200 Million
Platform Cost	Medium	Low	Medium	Medium	High	Medium	High	Medium

GraphChi, the edge-centric scatter-gather (ESG) model of X-Stream, the vertex-centric streamlined processing (VSP) model of VENUS, and the dual sliding windows (DSW) model of GridGraph. These approaches try to exploit the sequential bandwidth of hard disks and to reduce the amount of disk accesses. Nonetheless, current out-of-core graph engines still have much lower performance (5-100M edges/s) than that of in-memory graph engines (1-2B edges/s), as shown in Table 1. While Pregelix [39], Chaos [38] and GraphD [37] scale out out-of-core graph processing to multiple machines, their processing performance could not be significantly improved due to the high disk I/O cost.

In this paper, we propose GraphMP, a novel out-of-core graph processing system, to tackle big graph analytics on a single commodity machine[1] based on our previous work [40]. GraphMP employs three techniques to fully utilize available memory resources, and thus significantly reduces disk I/O overhead. First, we design a **vertex-centric sliding window (VSW)** computation model to avoid reading and writing vertices on hard disks. Specifically, GraphMP breaks the input graph's vertices into disjoint intervals. Each interval is associated with a shard, which contains all edges that have destination vertex in that interval. During the computation, GraphMP manages all vertices of a graph in the main memory, slides a window on vertices from disks, and processes edges shard by shard. When processing a specific shard, GraphMP first loads it into memory, then executes user-defined functions on it to update corresponding vertices. Thus, GraphMP does not need to read or write vertices on hard disks until the end of the program, since all of them are stored in memory. Compared to [40], we enhance **selective scheduling** and **compressed edge caching** to further reduce disk I/O overhead. Specifically, with selective scheduling, inactive shards, which would not update any vertices, can be skipped to avoid unnecessary disk accesses. Compressed edge cache mechanism could fully utilize available memory resources to cache as many as shards in memory. If a shard is cached, GraphMP would not access it from hard disks.

As shown in Figure 1, GraphMP can be distinguished from other single-machine graph engines as follows:

- Compared to CPU-based in-memory approaches like GraphMat and Ligra, GraphMP does not need to manage all edges in memory, so that it can handle big graph anlytics beyond the memory limit (real-world graphs usually contain much more edges than vertices).
- Compared to existing HDD-based out-of-core systems like GraphChi and GridGraph, GraphMP requires more memory to manage all vertices for low disk I/O overhead. Most of the time, this is not a problem as a single commodity machine can easily fit all vertices of a big graph into memory. Take PageRank as an example, a graph with 1.1 billion vertices needs about 22.1GB memory to manage all vertex values.
- Compared to graph engines like Gunrock and GTS, which use hetergeneous computation and storage devices (e.g., PCIe/NVMe SSD, GPU, FPGA and Xeon Phi), GraphMP is designed for big graph analytics on a commodity machine with just CPUs and HDDs.

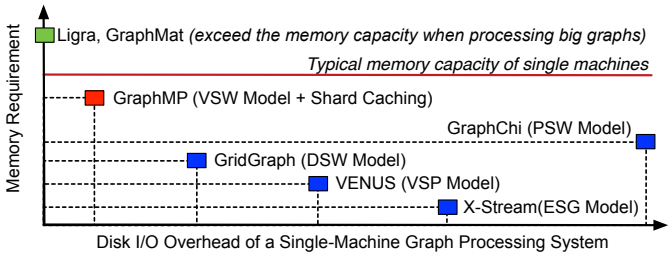


Fig. 1. Compared to in-memory systems like Ligra, GraphMP can handle big graphs on a single machine, since it does not store all graph data in memory. Compared to existing out-of-core systems (e.g., GraphChi, X-Stream, VENUS, GridGraph), GraphMP could fully utilize available memory of a typical server to reduce disk I/O overhead.

We implement GraphMP using C++ and OpenMP, which is available at https://github.com/cap-ntu/Graphee, Extensive evaluations on a testbed have shown that GraphMP performs much better than existing single-machine out-of-core approaches, and has competitive performance to popular in-memory and distributed solutions. When running PageRank, single source shortest path (SSSP) and weakly connected components (CC) on a graph with 1.1 billion vertices, which is the largest web graph dataset could be downloaded from http://law.di.unimi.it/datasets.php, GraphMP can outperform GraphChi, X-Stream and GridGraph by a factor of up to 30.

The rest of the paper is structured as follows. In section 2, we present the system design of GraphMP. Section 3 gives quantitative comparison between our approach with other graph processing systems. The evaluation results are detailed in Section 4. We conclude the paper in section 5.

1. It is common for current commodity single machine to have more than 64GB memory. For example, a single EC2 M4 instance can have up to 256GB memory. In this work, GraphMP is deployed on a Dell R720 server with two Intel Xeon E5-2620 processors (12 cores in total), 128GB memory and 4x4TB HDDs (RAID5).

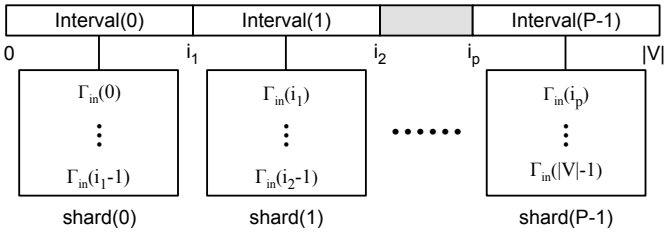


Fig. 2. The input graph's vertices are divided into P intervals. Each interval is associated with a shard, which stores all edges that have destination vertex in that interval. GraphMP structures all edges of a shard in key-value pairs $(id(v), \Gamma_{in}(v))$, and stores them in the Compressed Sparse Row format.

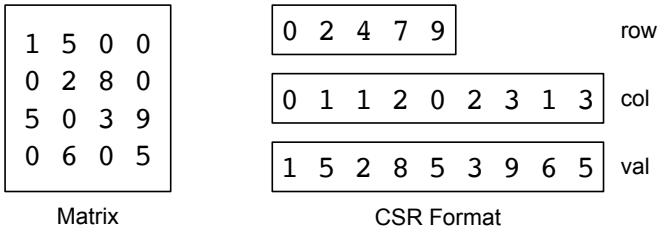


Fig. 3. An example of using the CSR format to represent a sparse matrix.

2 System Design

In this section, we introduce the system design of GraphMP, and show how GraphMP handles big graph analytics in a single machine with vertex-centric sliding window (VSW) model, selective scheduling and compressed edge caching.

2.1 Notations

Graph $G = (V, E)$ contains $|V|$ vertices and $|E|$ edges. Each vertex v has a unique ID $id(v)$, a value $val(v)$, an incoming adjacency list $\Gamma_{in}(v)$, an outgoing adjacency list $\Gamma_{out}(v)$, and a boolean field $active(v)$. During the computation, $val(v)$ may be updated, and $active(v)$ indicates whether $val(v)$ is updated in the last iteration. If vertex $u \in \Gamma_{in}(v)$, u is an incoming neighbor of v, and (u, v) is an in-edge of v. If $u \in \Gamma_{out}(v)$, u is an outgoing neighbor of v, and (v, u) is an out-edge of v. $d_{in}(v) = |\Gamma_{in}(v)|$ and $d_{out}(v) = |\Gamma_{out}(v)|$ are the in-degree and out-degree of v, respectively. Let $val(u, v)$ denote the edge value of (u, v). In this work, if G is a unweighted graph, $val(u, v) = 1, \forall (u, v) \in E$.

2.2 Graph Sharding and Data Processing

Before vertex-centric computation, GraphMP breaks the input graph into P shards in data processing stage. As shown in Figure 2, the vertices of $G = (V, E)$ are divided into P disjoint intervals. Each interval is associated with a shard, which stores all edges that have destination vertex in that interval. For example, in Figure 2, `shard(1)` contains all edges with destination vertex v, where $i_1 \leq id(v) \leq i_2 - 1$. In this example, i_1 is the start vertex id of `shard(1)`, and $i_2 - 1$ is its end vertex id. Vertex intervals are chosen with two policies:

- Any shard can be completely loaded into memory;
- Each shard tries to contain a similar number of edges for workload balance during computation.

In this work, each shard approximately contains 20 millions edges, so that a single shard roughly needs 80MB memory. Users can use other vertex intervals for specific applications or graph data sets.

GraphMP groups edges in a shard by their destination, and manage them as a sparse matrix in Compressed Sparse Row (CSR) format. One edge is treated as a non-zero entry of the sparse matrix. The CSR format of a sparse matrix contains a `row` array, a `col` array, and a `val` array. Specifically, the `col` array stores all non-zero entries' column indices in row-major order. The `val` array contains corresponding nonzero values. The `row` array records each row's start point in `col` and `val` array. Figure 3 shows an example of using the CSR format to represent a 4-row sparse matrix. In this example, `row[3]` = 7, `row[4]` = 9. It means that the last row of the matrix contains 2 nonzero entires, whose column indices are stored in `col[7]` and `col[8]`. The corresponding values can be accessed from `val[7]` and `val[8]`. This work maps the edges of a shard as a sparse matrix in CSR format. For example, in Figure 2, `shard(1)` can be mapped as a sparse matrix with $i_2 - i_1$ rows and $|V|$ columns. In CSR, the `col` array stores all edges' column indices in row-major order, and the `row` array records each vertex's adjacency list distribution. If the input graph is an unweighted graph, there is no need to construct the `val` array, since all edges have the same weight. In `shard(1)` of Figure 2, the incoming adjacency list of vertex v ($i_1 \leq id(v) \leq i_2$-1) can be accessed from:

$$\{col[row[id(v) - i_1]], \cdots, col[row[id(v) + 1 - i_1] - 1]\}.$$

In addition to edge shards, GraphMP creates two metadata files. First, a property file contains the global information of the represented graph, including the number of vertices, edges and shards, and the vertex intervals. Second, a vertex information file stores several arrays to record the information of all vertices. It contains an array to record all vertex values (which can be the initial or updated values), an in-degree array and an out-degree array to store each vertex's in-degree and out-degree, respectively.

Algorithm 1: Compute Vertex Intervals

```
shard_id = 0, vertex_id = 0, edge_num = 0
shard[shard_id].start_vertex_id = 0
while vertex_id < |V| do
    edge_num += Γin(V_vertex_id)
    if edge_num > threshold_edge_num then
        shard[shard_id].end_vertex_id = vertex_id - 1
        shard_id += 1
        shard[shard_id].start_vertex_id = vertex_id
        edge_num = Γin(V_vertex_id)
    vertex_id += 1
shard[shard_id].end_vertex_id = |V| - 1
```

GraphMP uses three steps to preprocess a graph. In the first step, GraphMP scans the graph to record each vertex's in-degree. With this information, GraphMP computes each shard's associated vertex interval based on Algorithm 1. In this method, *threshold_edge_num* denotes the max number of edges a shard could contain, which is user defined and should be no greater than the graph's max in-degree. With

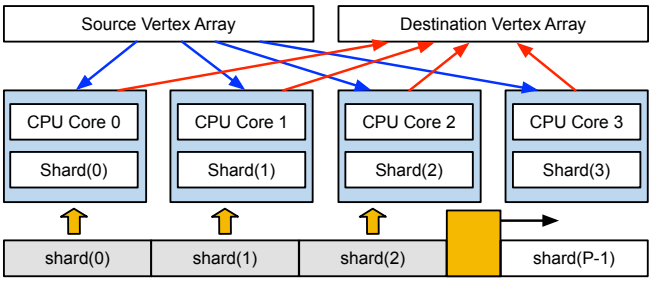


Fig. 4. The VSW computation model. GraphMP slides a window on vertices, and makes each CPU core process a shard at a time. When processing a shard, a CPU core continually pulls required vertex values from memory, and pushes updated ones to another array in memory.

this simple method, GraphMP quickly divides input vertices into a set of intervals, and each shard is allocated with a start vertex id and an end vertex id. This method also guarantees that each shard is small enough to be loaded into memory, and tries to let each shard contain a similar number of edges. In the second step, GraphMP sequentially reads graph edges from disk, and appends each edge to a shard file based on its destination and computed vertex intervals. In the third step, GraphMP transforms all shard files to the CSR format, and persists them on disk. After these three steps, all edges are actually sorted and grouped by their destination vertices.

After the data preprocessing, GraphMP is ready to perform vertex-centric computation based on the VSW model. GraphMP only needs to perform data preprocessing one time, then could execute any graph applications using the same partitioned data. As a comparison, GraphChi needs to preprocess input graph again before executing a new type of a graph application. For example, GraphChi cannot use the same partitioned graph for PageRank to run SSSP [31].

2.3 Vertex-Centric Sliding Window Computation

GraphMP slides a window on vertices, and processes edges shard by shard on a single machine with N CPU cores, as shown in Figure 4 and Algorithm 2. During the computation, GraphMP maintains two vertex arrays in memory until the end of the program: `SrcVertexArray` and `DstVertexArray`. The `SrcVertexArray` stores latest vertex values, which are the input of the current iteration. Updated vertex values are written into the `DstVertexArray`, which are used as the input of the next iteration. GraphMP uses OpenMP to parallelize the computation (line 3 of Algorithm 2): each CPU core processes a shard at a time. When processing a specific shard, GraphMP first loads it into memory (line 6), then executes user-defined vertex-centric functions, and writes the results to the `DstVertexArray` (line 7-8). Given a vertex, if its value is updated, we call it an active vertex. Otherwise, it is inactive. After processing all shards, GraphMP records all active vertices in a list (line 9). This list could help GraphMP to avoid loading and processing inactive shards in the next iteration (line 5), which would not generate any updates (detailed in Section 2.4). The values of `DstVertexArray` are used as the input of next iteration (line 10). The program terminates if it does not generate any active vertices (line 2).

Algorithm 2: Vertex-Centric Sliding Window Model

```
1  init (src_vertex_array, dst_vertex_array)
2  while active_ratio > 0 do
3      # pragma omp parallel for num_threads(N)
4      for shard ∈ all_shards do
5          if active_vertex_ratio > threshold_active_ratio or
             Bloom_filter[shard.id].has(active_vertices) then
6              load_to_memory(shard)
7              for v ∈ shard.associated_vertices do
8                  dst_vertex_array[v.id] ← update(v,
                     src_vertex_array)
9      active_vertices = {vertices with updated values}
10     src_vertex_array ← dst_vertex_array
11     active_ratio ← |active_vertices| / vertex_num
```

Users need to define two functions for a particular application: `Init` and `Update`. Specifically, the `Init` function takes `SrcVertexArray` and `DstVertexArray` as inputs,

```
Init(SrcVertexArray, DstVertexArray),
```

and initialize the values of all vertices. The `Update` function accepts a vertex and `SrcVertexArray` as inputs,

```
Update(InputVertex, SrcVertexArray),
```

and should return two results: an updated vertex value which should be stored in `DstVertexArray`, and a boolean value to indicate whether the input vertex updates its value. Specifically, this function allows the input vertex to pull the values of its incoming neighbors from `SrcVertexArray` along the in-edges, and uses them to update its value.

We implement three popular graph applications (PageRank, SSSP and CC) using `Init` and `Update` in Algorithm 3. PageRank is an algorithm used to measure the importance of website pages. SSSP is used to find shortest paths from a source vertex to all other vertices in the graph. CC can detect whether any two vertices of the graph are connected to each other by paths. In PageRank, the vertex value type is `Double` to store the rank value of a vertex. The graph is initialized before the first iteration, and the value of each vertex is $1/vertex\_num$ (line 3-4). All vertices are set to be active in the initialization phase (line 5). During the iterative computation, each vertex accumulates vertex values along its in-edges into sum (line 8-9), and sets its own rank value to $0.15/vertex\_num + 0.85 * sum$ (line 10). The two hyper-parameters "0.15" and "0.85" are adopted from Google [20], which help the algorithms converges smoothly. In SSSP, the vertex value type is `Long` to store the minimum distance from the source vertex (for example, vertex 0). Before the first iteration, the source vertex initializes its value to zero, and other vertex values are initialized to ∞ (line 14-18). Only the source vertex is set to be active in the initialization phase (line 19). During the computation, each vertex connects its neighbor vertices along in-edges (line 22-23), and tries to find a shorter path to the source vertex (line 24). When running CC on undirected graphs, the vertex value type is `Long` to store the subgraph ID. If two vertices have the same subgraph ID, they are connected to each other by paths. Before the first iteration, the value of each vertex is initialized to its vertex ID (line 28-29). All vertices are

Algorithm 3: PageRank, SSSP and CC in GraphMP

```
1  // vertex_value (Double) is the rank value
2  Function PR_Init(Array src_vertex, Array dst_vertex)
3      for i ∈ range(num_vertex) do
4          src_vertex[i] = dst_vertex[i] = 1 / num_vertex
5      active_vertices = {all vertices}
6  Function PR_Update(Vertex v, Array src_vertex)
7      sum = 0
8      for e ∈ v.incoming_neighbours do
9          sum += src_vertex[e.source] / e.source.out_deg
10     updated_value = 0.15 / num_vertex + 0.85 * sum
11     return updated_value
12 // vertex_value (Long) is the distance to the source vertex
13 Function SSSP_Init(Array src_vertex, Array dst_vertex)
14     for i ∈ range(num_vertex) do
15         if i == source_vertex.id then
16             src_vertex[i] = dst_vertex[i] = 0
17         else
18             src_vertex[i] = dst_vertex[i] = ∞
19     active_vertices = {source_vertex}
20 Function SSSP_Update(Vertex v, Array src_vertex)
21     d = ∞
22     for e ∈ v.incoming_neighbours do
23         d = min (src_vertex[e.source] + (e,u).val, d)
24     updated_value = min (d, v.value)
25     return updated_value
26 // vertex_value (Long) is the vertex group id
27 Function CC_Init(Array src_vertex, Array dst_vertex)
28     for i ∈ range(num_vertex) do
29         src_vertex[i] = dst_vertex[i] = i
30     active_vertices = {all vertices}
31 Function CC_Update(Vertex v, Array src_vertex)
32     subgraph_id = ∞
33     for e ∈ v.incoming_neighbours do
34         subgraph_id = min (src_vertex[e.source],
               subgraph_id)
35     updated_value = min (subgraph_id, v.value)
36     return updated_value
```

set to be active in the initialization phase (line 30). During the computation, each vertex checks the subgraph ID of its neighbors, and overwrites its own subgraph ID with the max vertex ID received from its neighbors. This continues until convergence (line 33-35).

When using multiple CPU cores to process graph shards in parallel, GraphMP does not require locks or atomic operations. This property could improve the graph processing performance considerably. As shown in Figure 4 and Algorithm 2 (line 3), in each iteration, GraphMP uses one CPU core to process a shard for updating its associated vertices, and could process N shards in parallel when having N GPU cores. Given a vertex v, `SrcVertexArray[v.id]` may be accessed as input by multiple CPU cores at the same time. Due to GraphMP's graph sharding strategy (specifically all in-edges of a vertex are managed in the same shard), `DstVertexArray[v.id]` is computed from edges located in a single shard, and could be written by a single CPU core in each iteration. Therefore, there is no need to use locks or atomic operations to avoid data inconsistency issues on `SrcVertexArray` and `DstVertexArray`. As a comparison, GridGraph should use an atomic operation to process each edge, since multiple cores may simultaneously update the same vertex [34].

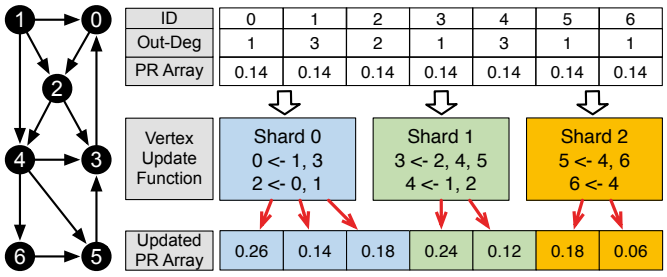


Fig. 5. An example of the first iteration of PageRank on GraphMP.

In Figure 5, we show an example of how GraphMP runs PageRank. The input graph is partitioned into three shards, each of which contains two vertices and their adjacency lists. At the beginning of PageRank, all vertex values are initiated to be $1/num\_vertex = 0.14$. GraphMP slides a window on vertices, and lets each CPU core process a shard at a time. When processing shard 0 on a CPU core, GraphMP pulls the values of vertex 1, 3 from `SrcVertexArray`, then use them to compute the updated value for vertex 0, and writes it to `DstVertexArray[0]`. After processing all 3 shards, GraphMP uses the values of `DstVertexArray` to replace the values of `SrcVertexArray`, and starts the next iteration if there are any active vertices.

2.4 System Optimizations

GraphMP employs two optimization techniques (specifically selective scheduling and compressed edge caching) to further reduce the disk I/O overhead and improve the graph processing performance.

2.4.1 Selective Scheduling

For many graph applications, such as PageRank, SSSP and CC, a lot of vertices converge quickly and would not update their values in the rest iterations. Given a shard, if all source vertices of its associated edges are inactive, it is an inactive shard. An inactive shard would not generate any updates in the following iteration. Therefore, it is unnecessary to load and process these inactive shards.

To solve this problem, we leverage Bloom filters to detect inactive shards, so that GraphMP could avoid unnecessary disk accesses and processing. A Bloom filter is a memory-efficient data structure, which can rapidly test whether an element is a member of a set by using multiple hash functions. GraphMP manages a Bloom filter for each shard to record the source vertices of its edges. When processing a shard, GraphMP uses its Bloom filter to check whether it contains any active vertices. If yes, GraphMP would continue to load and process the shard. Otherwise, GraphMP would skip it. For example, in Figure 5, when the sliding window is moved to shard 2, its Bloom filter could tell GraphMP whether vertex 4, 6 have changed their values in the last iteration. If there are no active vertices, the sliding window would

TABLE 2
Compression ratio and processing throughput per CPU core.

	Compression Ratio			Throughput (MB/s)		
	snappy	zlib-1	zlib-3	snappy	zlib-1	zlib-3
Twitter	1.75	2.78	3.22	870	55	46
UK-2007	1.89	3.71	4.54	947	58	53
UK-2014	1.96	4.34	5.26	903	65	50
EU-2015	1.96	4.35	5.88	890	62	56

	Size (GB) (CSV)	Size (GB) (raw)	Size (GB) (snappy)	Size (GB) (zlib-1)	Size (GB) (zlib-3)
Twitter	24	6.5	3.7	2.3	2
UK-2007	94	23	12	6.2	5
UK-2014	874	196	100	45	37
EU-2015	1700	362	185	80	62

skip shard 2, since it cannot not update vertex 5 or 6 after the processing.

GraphMP only enables selective scheduling when the ratio of active vertices is lower than a threshold. If the active vertex ratio is high, nearly all shards contain at least one active vertex. In this case, GraphMP wastes a lot of time on detecting inactive shards, and would not reduce any unnecessary disk accesses. As shown in Algorithm 2 Line 5, GraphMP starts to detect inactive shards when the active vertex ratio is lower than a threshold. In this paper, we use 0.001 as the threshold according to our experiment data. Users can choose a better value for specific applications.

2.4.2 Compressed Edge Caching

We design a cache system in GraphMP to reduce the amount of disk accesses for edges. The VSW computation model requires storing all vertices and shards under processing in the main memory. These data would not consume all available memory resources of a single machine. For example, given a server with 24 CPU cores and 128GB memory, when running PageRank on a graph with 1.1 billion vertices, GraphMP uses 21GB memory to store all data, including `SrcVertexArray`, `DstVertexArray`, the out-degree array, Bloom filters, and the shards under processing. It motivates us to build an in-application cache system to fully utilize available memory to reduce the disk I/O overhead. Specifically, when GraphMP needs to process a shard, it first searches the cache system. If there is a cache hit, GraphMP can process the shard without disk accesses. Otherwise, GraphMP loads the target shard from disk, and leaves it in the cache system if the cache system is not full.

GraphMP can compress cached shards to improve the amount of cached edge shards and further reduce disk I/O cost. Table 2 shows that popular compressors can efficiently reduce the size of graph datasets. We use four real-world graphs as inputs: Twitter, UK-2007, UK-2014 and EU-2015. Section 4 gives more detail about these four graph datasets. We see that zlib-3 could compress EU-2015 by a factor of 5.88. While GraphPS needs additional decompression time, the edge cache system still provides higher throughput than hard disks. For example, snappy can decompress an edge shard at a rate of 903MB/s using a single CPU core. In contrast, we can only achieve up to 310MB/s sequential disk read speed with RAID5, and the available disk bandwidth is shared by all CPU cores.

In this work, we use two compressors (snappy and zlib), and consider 5 cache modes:

- Cache-0: Use system page cache without edge cache.
- Cache-1: Cache uncompressed edge shards.
- Cache-2: Cache shards compressed by snappy.
- Cache-3: Cache shards compressed by zlib-1.
- Cache-4: Cache shards compressed by zlib-3.

GraphMP can automatically select the most suitable cache mode, considering disk I/O and decompression cost. When having limited memory, it is crucial to select compressors with high compression ratio for low disk I/O overhead. If the memory is large, caching shards with low compression ratio can reduce decompression overhead without increasing disk I/O overhead. Let C denote the memory size of the cache system, S is the input graph's size, and γ_i is the estimated compression ratio of cache mode-i. GraphMP selects minimal i constrained by $S/\gamma_i \leq C$. If no mode satisfies this constraint, GraphMP uses mode-4 with highest compression ratio. In this case, GraphMP caches as many shards as possible in memory, and reads other shards from disk during computation. In this work, $\gamma_0 = 1, \gamma_1 = 2, \gamma_2 = 4, \gamma_3 = 5$, according to Table 2.

3 Theoretical Comparison

We compare our proposed VSW model with four popular graph computation models: the parallel sliding window model (PSW) of GraphChi, the edge-centric scatter-gather (ESG) model of X-Stream, the vertex-centric streamlined processing (VSP) model of VENUS and the dual sliding windows (DSW) model of GridGraph. All systems partition the input graph into P shards or blocks, and run applications using N CPU cores. Let C denote the size of a vertex record, and D is the size of one edge record. For fair comparison, we assume that the neighbors of a vertex are randomly chosen, and the average degree is $d_{avg} = |E|/|V|$. We disable selective scheduling, so that all system should process all edges in each iteration. We use the amount of data read and write on disk per iteration, and the memory usage as the evaluation criteria. Table 3 summarizes the analysis results.

3.1 The PSW Model of GraphChi

Under PSW, GraphChi splits the vertices V of graph $G = (V, E)$ into P disjoint intervals. For each interval, GraphChi associates a shard, which stores all the edges that have destination in the interval. Edges are stored in the order of their source. Unlike GraphMP where each vertex can access the values of its neighbors from `SrcVertexArray`, GraphChi accesses such values from the edges. Thus, the data size of each edge in GraphChi is $(C + D)$ [33]. In addition, GraphChi stores updated vertex values in a single file as flat array of user-defined type. For each iteration, GraphChi uses three steps to process a shard: (1) loading its associated vertices, in-edges and out-edges from disk into memory; (2) updating the vertex values; and (3) writing updated vertices and edges to disk. In step (1), GraphChi loads each vertex (which incurs $C|V|$ data read), and accesses in-edges and out-edges of each vertex (which incurs $2(C + D)|E|$ data read). In step (3), GraphChi writes updated vertices into disk (which incurs $C|V|$ data write), and writes each edge

TABLE 3
Analysis of graph computation models. C is the size of a vertex value, D is the size of an edge value, P is the number of partitioned shards or blocks of a graph, d_{avg} denotes the graph's average degree, $\delta \approx (1 - e^{-d_{avg}/P})P$, θ denotes GraphMP's cache hit ratio and $0 \leq \theta \leq 1$.

Category	PSW (GraphChi)	ESG (X-Stream)	VSP (VENUS)	DSW (GridGraph)	VSW (GraphMP)
Data Read	$C\|V\| + 2(C+D)\|E\|$	$C\|V\| + (C+D)\|E\|$	$C(1+\delta)\|V\| + D\|E\|$	$C\sqrt{P}\|V\| + D\|E\|$	$\theta D\|E\|$
Data Write	$C\|V\| + 2(C+D)\|E\|$	$C\|V\| + C\|E\|$	$C\|V\|$	$C\sqrt{P}\|V\|$	0
Memory Usage	$(C\|V\| + 2(C+D)\|E\|)/P$	$C\|V\|/P$	$C(2+\delta)\|V\|/P$	$2C\|V\|/\sqrt{P}$	$2C\|V\| + ND\|E\|/P$
Preprocessing I/O Cost	$(C+5D)\|E\|$	$2D\|E\|$	$4D\|E\|$	$6D\|E\|$	$5D\|E\|$

twice (which incurs $2(C+D)|E|$ data write) if the computation updates edges in both directions. With the PSW model, the data read and write in total are both $C|V|+2(C+D)|E|$. In step (2), GraphChi needs to keep $|V|/P$ vertices and their in-edges, out-edges in memory for computation. The memory usage is $(C|V| + 2(C+D)|E|)/P$.

GraphChi uses 3 steps to divide a graph into P shards: (1) counting the in-degree of each vertex (which incurs $D|E|$ data read) and dividing vertices into P intervals , (2) writing each edge to a temporary scratch file of the owning shard (which incurs $D|E|$ data read and $D|E|$ data write); and (3) sorting edges and writing each file in compact format (which incurs $D|E|$ data read and $(C+D)|E|$ data write). The total I/O cost of the preprocessing is $(C+5D)|E|$.

3.2 The ESG Model of X-Stream

X-Stream splits the input graph's vertices into P partitions, each of which could fits in high-speed memory. Furthermore, X-Stream assigns edges to P partitions, such that the edge list of a partition consists of all edges whose source vertex is in the partitions vertex set. Then, X-Steam processes the graph one partition at a time with two phases under the ESG model. In phase (1), when processing a graph partition, X-Stream first loads its associated vertices into memory, and processes its out-edges in a streaming fashion: generating and propagating updates (the size of an update is C) to corresponding values on disk. In this phase, the size of data read is $C|V| + D|E|$, and the size of data write is $C|E|$. In phase (2), X-Stream processes all updates and uses them to update vertex values on disk. In this phase, the size of data read is $C|E|$, and the size of data write is $C|V|$. With the ESG model, the data read and write in total are $C|V| + (C+D)|E|$ and $C|V| + C|E|$, respectively. X-Stream only needs to keep the vertices of a partition in memory, so the memory usage is $C|V|/P$.

X-Stream needs one step for data preprocessing. Specifically, before the computation, it reads edges from disks in sequence, and append them to corresponding files on disks. X-Stream does not need to sort edge lists or convert the data structure during preprocessing. Thus, the I/O cost of the preprocessing is $2D|E|$.

3.3 The VSP Model of VENUS

VENUS evenly splits $|V|$ vertices into P disjoint intervals. Each interval is associated with a g-shard (which stores all edges with destination in that interval), and a v-shard (which contains all vertices appear in that g-shard). For each iteration, VENUS processes g-shards and v-shards sequentially in three steps: (1) loading a v-shard into the main memory, (2) processing its corresponding g-shard in a streaming fashion, (3) writing updated vertices to disk. In step (1), VENUS needs to process all edges once, which incurs $D|E|$ data read. In step (3), all updated vertices are written to disk, so the data write is $C|V|$. According to Theorem 2 in [41], each vertex interval contains $|V|/P$ vertices, and each v-shard contains up to $|V|/P + (1 - e^{-d_{avg}/P})|V|$ entries. Therefore, the data read and write are $C(1+\delta)|V| + D|E|$ and $C|V|$ respectively, where $\delta \approx (1 - e^{-d_{avg}/P})P$. VENUS needs to keep a v-shard and its updated vertices in memory, so the memory usage is $C(2+\delta)|V|/P$.

VENUS uses two steps for preprocessing. Since VENUS evenly splits the set of vertices into P disjoint intervals, there is no need to count the degree of each vertex first. Therefore, VENUS reads the input graph data sequentially, adds each encountered edge into a buffer according to its destination, and writes the sorted edges into an intermediate file when an buffer is full. The second step performs a k-way merge on all intermediate files resulted from the first step to construct required data structure. Thus, edges are grouped by their destination. The I/O cost of the preprocessing is $4D|E|$.

3.4 The DSW Model of GridGraph

GridGraph groups the input graph's $|E|$ edges into a "grid" representation. Specifically, the $|V|$ vertices are divided into $\sqrt{P}$ equalized vertex chunks and $|E|$ edges are partitioned into $\sqrt{P} \times \sqrt{P}$ blocks according to the source and destination vertices. Each edge is placed into a block using the following rule: the source vertex determines the row of the block, and the destination vertex determines the column of the block. GridGraph processes edges block by block. GridGraph uses 3 steps to process a block in the i-th row and j-th column: (1) loading the i-th source vertex chunk and the j-th destination vertex chunk into memory; (2) processing edges in a streaming fashion for updating the destination vertices; and (3) writing the destination vertex chunk to disk if it is not required by the next block. After processing a column of blocks, GridGraph reads $|E|/\sqrt{P}$ edges and $|V|$ vertices, and writes $|V|/\sqrt{P}$ vertices to disk. The data read and write are $C\sqrt{P}|V| + D|E|$ and $C\sqrt{P}|V|$, respectively. During the computation, GridGraph needs to keep two vertex chunks in memory, so the memory usage is $2C|V|/\sqrt{P}$.

GridGraph needs three steps for data processing based on the provided program. In the first step, GridGraph reads edges sequentially, calculates the block that an edge belongs to, and appends the edge to the corresponding block file. To improve I/O throughput, GridGraph combines the generated $\sqrt{P} \times \sqrt{P}$ block files into a column-oriented file and a row-oriented file in step 2 and step 3. The I/O cost of the preprocessing is $6D|E|$.

3.5 The VSW Model of GraphMP

GraphMP keeps all source and destination vertices in the main memory during the vertex-centric computation. Therefore, GraphMP would not incur any disk write for vertices in each iteration until the end of the program. In each iteration, GraphMP should use N CPU cores to process P edge shards in parallel, which incurs $D|E|$ data read. Since GraphMP uses a compressed edge cache mechanism, the actual size of data read of GraphMP is $\theta D|E|$, where $0 \leq \theta \leq 1$ is the cache miss ratio. During the computation, GraphMP manages $|V|$ source vertices (which are the input of the current iteration) and $|V|$ destination vertices (which are the output the current iteration and the input of the next iteration) in memory, and each CPU core loads $|E|/P$ edges in memory. The total memory usage is $2C|V| + ND|E|/P$. As discussed in Section 2, GraphMP needs three steps for data preprocessing, and its I/O cost is $5D|E|$.

3.6 Discussion

PSW, ESG, VSP, DSW and VSW adopt a similar way to process large-scale graphs on a single machine: they partition a big graph into small shards or blocks, and uses limited computation resources to sequentially process these small graph shards. As detailed in this section, each graph computation model has its own graph sharding policy, data structure and vertex-centric computation flow, which could significantly affect the I/O overhead of graph applications. As shown in Table 3, the VSW model of GraphMP could reduce the amount of data reads and writes on disks than other models at the cost of higher memory usage. More specifically, VSW manages all vertices in memory and only needs to access edges from hard disks during the computation. With the help of compressed edge caching, VSW further reduces the amount of data reads by caching a portion of edge shards in memory. As a comparison, PSW, ESG, VSP and DSW should read both vertices and edges from disks at each iteration. Additionally, VSW could directly update vertex values in memory without writing those data to disks. As a compression, VSP and DSW should frequently write updated vertices to disks. PSW and ESG need to update on-disk edges since they use a single data structure to manage both edges and vertices. In Section IV, we use experiments to show that a single commodity machine could provide sufficient memory for processing big graphs with the VSW model. Also, GraphMP has similar data preprocessing I/O cost with other graph engines.

4 Performance Evaluations

We evaluate the performance of GraphMP using a Dell R720 server with three applications (PageRank, SSSP, CC) and four directed graph datasets. The physical machine contains two Intel Xeon E5-2620 CPUs, 128GB memory, 4x4TB HDDs (RAID5). Table 4 shows the information of used datasets: Twitter, UK-2007, UK-2014 and EU-2015. Twitter is a social network graph crawled in 2010 [42], showing connections between twitter users. UK-2007 and UK-2014 are two web graphs crawled in 2007 and 2014 respectively, showing links between web pages in the .uk domain. EU-2015 is a web graph crawled in 2015, showing page links in European

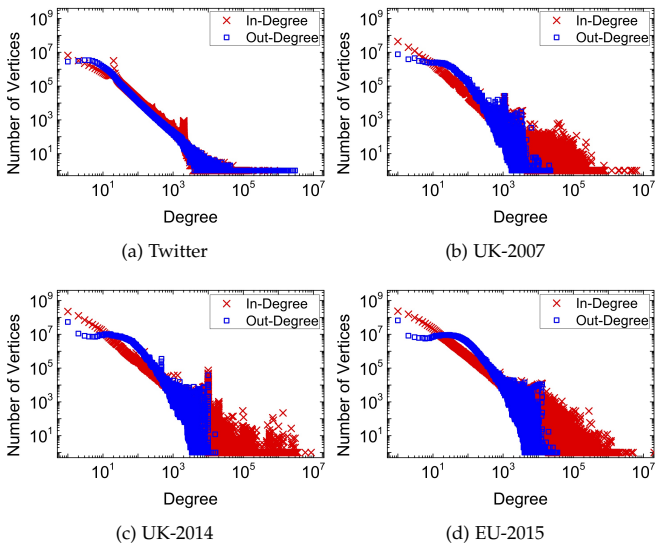


Fig. 6. The in-degree and out-degree distribution of used graph datasets. All four graphs are power-law graphs: most vertices have relatively few neighbors while a few have many neighbors.

TABLE 4
Graph datasets used in the experiments[1].

Dataset	Vertex Num	Edge Num	Avg Deg	Max Indeg	Max Outdeg	Size (CSV)
Twitter	42M	1.5B	35.3	0.7M	770K	25GB
UK-2007	134M	5.5B	41.2	6.3M	22.4K	93GB
UK-2014	788M	47.6B	60.4	8.6M	16.3K	0.9TB
EU-2015	1.1B	91.8B	85.7	20M	35.3K	1.7TB

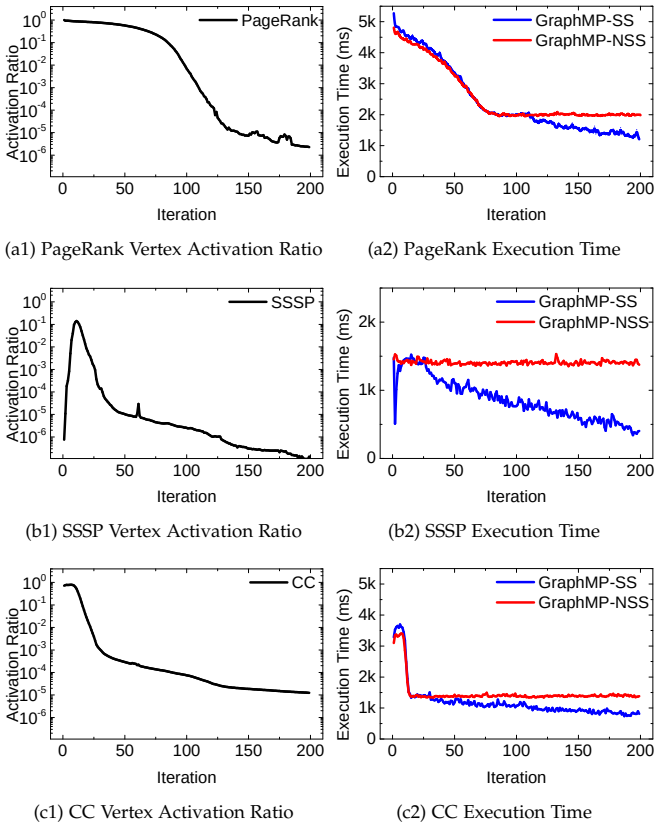
[1] All datasets is public on http://law.di.unimi.it/datasets.php.

Union countries. EU-2015 is our largest graph dataset, containing 1.1 billion vertices and 91.8 billion edges. If we store the raw graph in the CSV format, EU-2015 is a 1.7TB file. All datasets are real-word power-law graphs. As shown in Figure 6, in all four graphs, most vertices have relatively few neighbors while a few have many neighbors. Since we run CC on undirected graphs, we need to convert the input directed graphs into undirected graphs, and use undirected graphs as the input of CC.

In this section, we first evaluate the effect of GraphMP's selective scheduling and compressed edge caching. Then, we compare the performance of GraphMP with GraphMat, which is a single-machine in-memory graph system. Next, we compare the performance of GraphMP with three popular single-machine out-of-core engines: GraphChi, X-Stream and GridGraph. Finally, we compare the performance of GraphMP with three distributed in-memory graph engines (Pregel+, PowerGraph and PowerLyra) and two distributed out-of-core systems (GraphD and Chaos). We set up aforementioned distributed engines on 9 R720 servers connected by 10Gbps network. Each server has the same configuration with the server used to run single-machine graph engines.

4.1 Effect of Selective Scheduling

In this set of experiments, we enable selective scheduling in GraphMP-SS, so that it can use Bloom filters to detect and skip inactive shards. In GraphMP-NSS, we disable selective

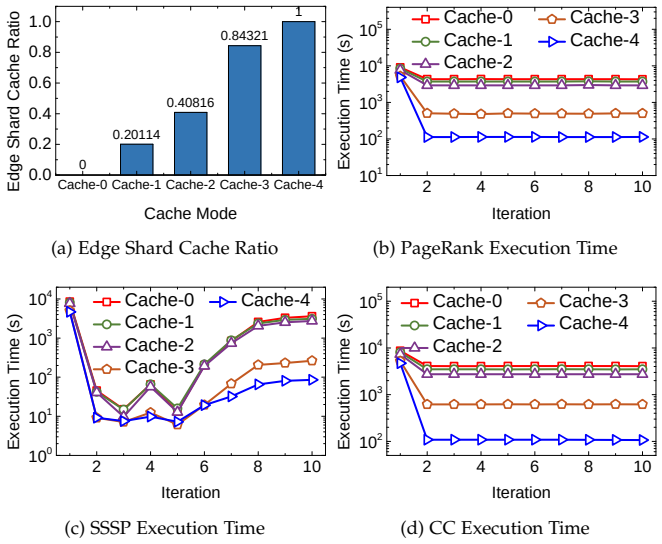


(a1) PageRank Vertex Activation Ratio (a2) PageRank Execution Time

(b1) SSSP Vertex Activation Ratio (b2) SSSP Execution Time

(c1) CC Vertex Activation Ratio (c2) CC Execution Time

Fig. 7. Effect of GraphMP's selective scheduling mechanism. GraphMP-SS enables selective scheduling. GraphMP-NSS disables the selective scheduling mechanism. We use UK-2007 as the input, and run PageRank, SSSP and CC on a single machine. The vertex activation ratio denotes the number of active vertices of an iteration.



(a) Edge Shard Cache Ratio (b) PageRank Execution Time

(c) SSSP Execution Time (d) CC Execution Time

Fig. 8. Effect of GraphMP's compressed edge caching. We use EU-2015 as input, run PageRank, SSSP and CC on GraphMP with different cache modes, and capture the execution time of first 10 iterations.

scheduling, so that it should process all shards in each iteration. To see the effect of GraphMP's selective scheduling, we run PageRank, SSP and CC on UK-2007 using GraphMP-SS and GraphMP-NSS, and compare their performance. Figure 7 shows that GraphMP's selective scheduling could improve the processing performance for all three applications.

As shown in Figure 7 (a1), when running PageRank on UK-2007, many vertices converge quickly. After the 110-th iteration, less than 0.1% of vertices update their values in an iteration. After that iteration, GraphMP-SS enables selective scheduling, and it continually reduces the execution time of an iteration. In particular, GraphMP-SS only uses 1.2 seconds to execute the 200-th iteration. As a comparison, GraphMP-NSS roughly uses 2 seconds per iteration after the 110-th iteration. In this case, selective scheduling could improve the processing performance of a single iteration by a factor of up to 1.67, and improve the overall performance by 5.8%.

From Figure 7 (b1) and (b2), we find that SSSP benefits a lot from GraphMP's selective scheduling mechanism. In this experiment, GraphMP updates more than 0.1% of vertices in a few iterations. Therefore, GraphMP-SS could continuously reduce the computation time from the 15-th iteration, and uses 0.4 seconds in the 200-th iteration. As a comparison, GraphMP-NSS roughly uses 1.4 seconds per iteration. In this case, GraphMP's selective scheduling mechanism could speed up the computation of an iteration by a factor of up to 2.86, and improve the overall performance of SSSP by 50.1%.

GraphMP's selective scheduling mechanism is enabled after the 31-th iteration of CC, as shown in Figure 7 (c1) and (c2). GraphMP-SS begins to outperform GraphMP-NSS from that iteration. GraphMP-SS uses 0.8 seconds in the 200-th iteration, and GraphMP-SS uses 1.4 seconds. In this case, GraphMP's selective scheduling mechanism reduces the computation time of an iteration by a factor of up to 1.75, and improves the overall performance of CC by 9.5%.

4.2 Effect of Compressed Edge Caching

To see the effect of GraphMP's compressed edge caching, we run PageRank, SSP and CC on EU-2015 using GraphMP with different cache modes, and compare their performance.

As shown in Figure 8(a), when using compressors with higher compression rate, GraphMP could cache more edge shards in memory. Specifically, GraphMP (Cache-0) could cache about 20% of edge shards in memory without data compressing. In the same testbed, GraphMP (Cache-3) could cache about 84.3% of edge shards and reduce disk I/O cost by using zlib-1 to compress cached edge shards. GraphMP (Cache-4) could cache all edge shards by compressing edge shards with zlib-3. In this case, there is even no disk I/O cost after loading all edge shards into memory.

From Figure 8(b), (c) and (d), we can see that GraphMP's compressed edge caching could significantly improve graph processing performance. Since GraphMP should access all graph data from disk for filling edge cache and constructing Bloom filters during the first iteration, it takes more time to complete this iteration than others. Figure 8(b) shows that GraphMP roughly takes 48650 seconds to complete the first 10 iterations of PageRank with cache-0. The corresponding values of cache-1, cache-2, cache-3 and cache-4 are 42075, 34077, 9678 and 5868 seconds, respectively. In this case, GraphMP's compressed edge caching can speed up PageRank by 8.3. When running SSSP, cache-1, cache-2, cache-3 and cache-4 could speed up the application by 1.1, 1.2,

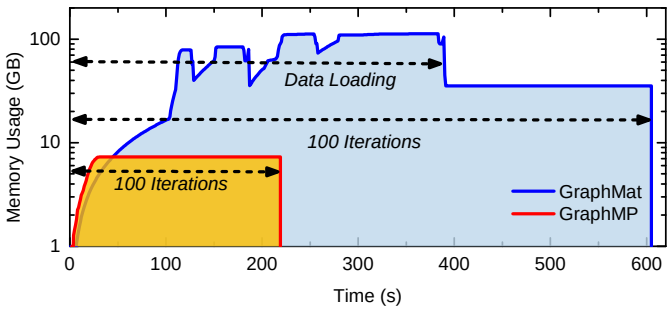


Fig. 9. Performance comparison between GraphMP and GraphMat. In this experiment, we run PageRank on the Twitter dataset.

3.3 and 3.8 respectively, compared to cache-0, as shown in Figure 8(c). In this experiment, GraphMP has less execution time from iteration 2 to iteration 6 due to selective scheduling: only a small portion of shards with active vertices are processed. As shown in Figure 8(d), CC also benefits from compressed edge caching. More specifically, cache-1, cache-2, cache-3 and cache-4 could speed up CC by a factor of 1.2, 1.4, 4.2 and 8.1 respectively, when compared to cache-0.

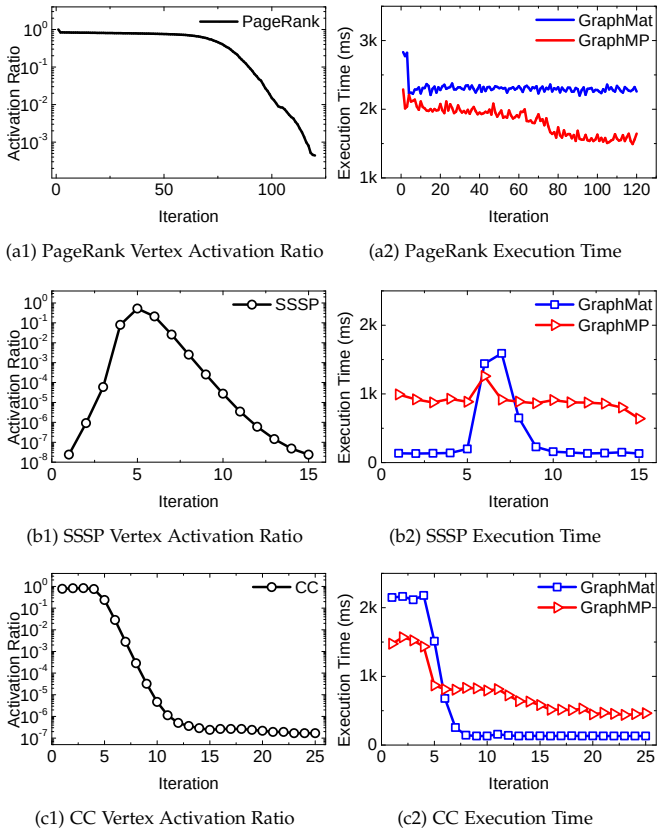
4.3 GraphMP vs. GraphMat

We compare the performance of GraphMP with GraphMat, which is an in-memory graph processing system. GraphMat maps vertex-centric programs to sparse vector-matrix multiplication (SpMV) operations, and leverages sparse linear algebra libraries and techniques to improve the performance of large-scale graph computation. The results show that GraphMP has competitive performance to GraphMat.

GraphMat could not process UK-2007, UK-2014 and EU-2015 in our machine with 128GB memory. At the beginning of a graph application, GraphMat loads all vertices and edges into memory, and manages them with required data structures. As shown in Figure 9, when running PageRank on Twitter, GraphMat uses up to 122GB memory for data loading. When processing UK-2007, UK-2014 and EU-2015 in our machine, GraphMat can easily crash caused by out-of-memory. As a comparison, GraphMP could efficiently process all 4 datasets in a single machine.

From Figure 9, we can find that GraphMat's data loading phases use 390 seconds to load the Twitter dataset into memory. Since GraphMat does not require the input graph's edges to be ordered, there is an expensive sorting process to build required data structures for SpMV during the data loading phase. As a comparison, GraphMP uses a separated data preprocessing stage to sort and group the input graph's edges, and could reuse these data in different applications. Thus, GraphMP uses 7.3GB memory (including Bloom filters and edge cache) to run PageRank on Twitter, and takes about 30 seconds to complete the first iteration, which contains the compressed edge cache filling time and Bloom filter construction time. In addition, GraphMP needs additional 340.2 seconds for data preprocessing.

Since GraphMP performs costly edge sorting in a separated data preprocessing stage and GraphMat sorts edges at the beginning of an application, we compare the performance of GraphMat with GraphMP in two cases. In the first



(a1) PageRank Vertex Activation Ratio (a2) PageRank Execution Time

(b1) SSSP Vertex Activation Ratio (b2) SSSP Execution Time

(c1) CC Vertex Activation Ratio (c2) CC Execution Time

Fig. 10. Performance comparison between GraphMP and GraphMat to run PageRank, SSSP and CC on the Twitter dataset. Vertex activation ratio denotes the ratio of active vertices of an iteration. In this figure, the first iteration's execution time does not include data loading time or initialization time for fair comparison.

case, we do not consider data loading overhead for both systems. In GraphMat, we start to measure the execution time when all vertices and edges are loaded into memory and all data structures are constructed. In GraphMP, we do not consider the time used for filling edge cache and constructing Bloom filters. Figure 10 shows vertex activation ratio and running time of each iteration when running PageRank, SSSP and CC on Twitter. We can see that GraphMat takes 28 seconds to complete the first 120 iterations of PageRank, and GraphMP uses 22 seconds. GraphMat takes 1.3 seconds to complete the first 15 iterations of SSSP, while GraphMP needs 9.9s. When running CC, GraphMat takes 1.5 seconds to complete the first 25 iterations, while GraphMP uses 2.1 seconds. In the second case, we consider data loading and data preprocessing overhead in both systems. Specifically, we add data preprocessing time of GraphMP to the total execution time. In this case, GraphMat takes 418 seconds for running PageRank, while GraphMP uses 366 seconds. GraphMat takes 349 seconds to run SSSP, while GraphMP needs 361s. When running CC in our testbed, GraphMat takes 382 seconds, while GraphMP uses 373 seconds.

4.4 GraphMP vs. GraphChi, X-Stream and GridGraph

In this set of experiments, we compare the performance of GraphMP with three out-of-core graph processing systems: GraphChi, X-Stream and GridGraph. We do not use VENUS,

TABLE 5
Performance comparison between GraphMP with other systems (application: PageRank; time unit: minutes; time collection: first 10 iterations).

Dataset	Single-Machine Out-of-Core			Distributed In-Memory			Distributed Out-of-Core		GraphMP	
	GraphChi	X-Stream	GridGraph	Pregel+	PowerGraph	PowerLyra	GraphD	Chaos	NoCache	Cache
Twitter	7.35	22.62	2.73	1.15	0.92	0.78	2.02	3.64	0.76	0.67
UK-2007	18.71	148.27	7.57	5.49	3.22	2.55	13.38	14.11	2.99	2.90
UK-2014	473.33	1413.35	675.73	-	-	-	543.82	389.90	336.45	46.36
EU-2015	970.67	2856.78	1162.63	-	-	-	2267.43	751.09	797.98	94.48

TABLE 6
Performance comparison between GraphMP with other systems (application: SSSP; time unit: minutes; time collection: first 10 iterations).

Dataset	Single-Machine Out-of-Core			Distributed In-Memory			Distributed Out-of-Core		GraphMP	
	GraphChi	X-Stream	GridGraph	Pregel+	PowerGraph	PowerLyra	GraphD	Chaos	NoCache	Cache
Twitter	21.35	7.66	14.35	0.17	1.11	0.75	0.26	2.66	0.59	0.51
UK-2007	64.45	31.23	38.07	1.29	2.72	2.49	1.36	4.25	2.49	2.35
UK-2014	647.58	697.43	507.63	-	-	-	589.64	371.08	261.04	46.50
EU-2015	1627.43	1478.75	514.62	-	-	-	2120.22	627.23	320.00	77.19

TABLE 7
Performance comparison between GraphMP with other systems (application: CC; time unit: minutes; time collection: first 10 iterations).

Dataset	Single-Machine Out-of-Core			Distributed In-Memory			Distributed Out-of-Core		GraphMP	
	GraphChi	X-Stream	GridGraph	Pregel+	PowerGraph	PowerLyra	GraphD	Chaos	NoCache	Cache
Twitter	21.23	11.78	16.67	1.57	1.39	1.06	2.66	4.85	0.60	0.55
UK-2007	61.15	115.94	35.82	7.81	4.24	4.07	16.12	18.79	2.82	2.79
UK-2014	635.97	1628.00	533.63	-	-	-	466.60	414.76	219.92	44.31
EU-2015	1553.70	2691.37	867.45	-	-	-	2172.95	735.02	451.74	91.25

since it is not open source. We run PageRank, SSSP and CC on Twitter, UK-2007, UK-2014 and EU-2015, and record their processing time of the first 10 iterations and memory usage. GraphMP-C denotes the system with compressed edge caching, and GraphMP-NC denotes the system without compressed edge caching. For fair comparison and simplicity, the first iteration's execution time of each application includes data loading and initialization time.

Table 5, 6 and 7 show the execution time of each iteration with different systems, datasets and applications. We could observe that GraphMP can considerably improve the graph processing performance, especially when dealing with big graphs. In Graph-C, the performance gain comes from three contributions: VSW model, selective scheduling, and compressed edge caching. When running PageRank on EU-2015, GraphMP-NC could outperform GraphChi, X-Steam and GridGraph by 1.21x, 3.58x and 1.46x, respectively. If we enable compressed edge caching, GraphMP-C further improves the processing performance, and outperforms GraphChi, X-Steam and GridGraph by 10.28x, 30.27x and 12.32x to run PageRank on EU-2015, respectively. When running SSSP, only a small part of vertices may update their values in an iteration. With selective scheduling, GraphMP-NC and GraphMP-C could skip loading and processing inactive shards to reduce the disk I/O overhead and processing time. GraphMP-NC could respectively outperform GraphChi, X-Steam and GridGraph by 5.09x, 4.62x and 1.61x for running SSSP on EU-2015. GraphMP's compressed edge caching mechanism could further reduce disk I/O overhead, and reduce the processing time. Thus, GraphMP-C could outperform GraphChi, X-Steam and GridGraph by 21.08x, 19.16x and 6.67x for running SSSP on EU-2015, respectively. When running CC on EU-2015, GraphMP-NC could outperform GraphChi, X-Steam and GridGraph by 3.42x, 5.96x and 1.92x, respectively. This performance gain is due to the VSW computation model with less disk I/O overhead. If we enable compressed edge caching, GraphMP-C respectively outperforms GraphChi, X-Steam and GridGraph by 17.02x, 29.49x and 9.51x to run CC on EU-2015.

TABLE 8
Preprocessing time comparison between GraphChi, GridGraph, X-Stream and GraphMP (time unit: minutes).

	Graphchi	Gridgraph	X-Stream	GraphMP
Twitter	11.08	4.83	3.38	5.67
UK-2007	45.42	23.98	14.20	20.93
UK-2014	453.07	422.02	130.41	313.18
EU-2015	1031.02	766.03	218.37	523.41

Table 8 shows the data preprocessing time of GraphChi, GridGraph, X-Stream and GraphMP. We use provided data preprocessing programs of GraphChi and GridGraph, and use C++ to implement a new data preprocessing engine for X-Stream, since X-Steam provides a Python script for data preprocessing with poor performance. In this experiment, all input graphs are stored in CSV format. From Table 8, we can find that GraphMP would not introduce much data preprocessing cost. When dealing with EU-2015, X-Stream offers the best performance: it only uses 218.37 minutes to split the input graph into partitions with required format. GraphChi, GridGraph and GraphMP take 1031.02, 766.03 and 523.41 minutes to preprocess EU-2015, respectively.

Figure 11 shows the memory usage of each graph processing system to run PageRank. We can see that GraphMP-NC uses more memory than GraphChi, X-Stream and GridGraph, since it keeps all source and destination vertices in memory. For example, when running PageRank on EU-

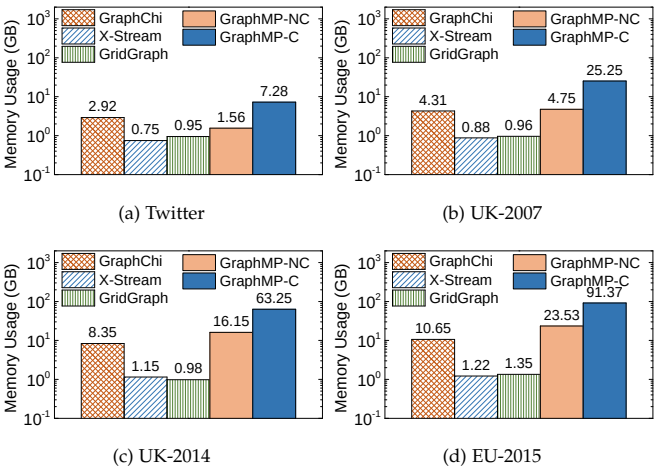


Fig. 11. Memory usage of 5 graph processing systems to run PageRank on Twitter, UK-2007, UK-2014 and EU-2015. We disable compressed cache mechanism in GraphMP-NC, and enable it in GraphMP-C.

2015, GraphChi, X-Stream and GridGraph only use 10.65GB, 1.22GB and 1.35GB memory, respectively. The corresponding value of GraphMP-NC is 23.53GB. GraphChi, X-Stream and GridGraph are designed for graph processing at scale on a single PC rather than a commodity server or a cloud instance. Even though our machine has 128GB memory, these systems cannot efficiently utilize them. If we enable compressed edge caching, GraphMP-C uses 91.37GB memory to run PageRank on EU-2015. In this case, GraphMP-C uses about 68GB as cache. Due to compression techniques, GraphMP-C could store all 91.8 billion edges in the cache using 68GB memory. Thus, there are no disk accesses for edges during the computation after the first iteration. While GraphMP-C needs additional time for shard decompression, it can still considerably improve the processing performance due to the reduced disk I/O overhead.

4.5 GraphMP vs. Distributed Graph Engines

We compare the performance of GraphMP with three distributed in-memory graph engines (Pregel+, PowerGraph and PowerLyra) and two distributed out-of-core approaches (GraphD and Chaos). We set up aforementioned distributed graph engines on 9 servers connected by 10Gbps network. Each server has the same hardware and software configuration with the server used to run GraphMP. The cluster totally has 1.15TB memory and 18 physical CPUs (108 cores, 216 threads). In the experiments, we enable selective scheduling and compressed edge caching in GraphMP.

Table 5, 6 and 7 show that GraphMP can be as highly competitive as distributed graph processing systems. When running PageRank on UK-2007, GraphMP outperforms Pregel+ and PowerGraph by 1.89x and 1.11x, respectively. In the same case, PowerLyra outperforms GraphMP by 1.13x. Compared to GraphD and Chaos, GraphMP could speed up PageRank on UK-2007 by a factor of 4.61 and 4.86. When running SSSP on UK-2007, Pregel+ and GraphD offer better performance than GraphMP, since they could avoid processing inactive vertices at the level of vertex. As a comparison, GraphMP could only perform selective scheduling at the level of edge shards. Compared to Chaos, GraphMP could speed up SSSP on UK-2007 by a factor of 1.81. When running CC on UK-2007, GraphMP could outperform Pregel+, PowerGraph, PowerLyra, GraphD and Chaos by 2.79x, 1.52x, 1.46x, 5.78x and 6.74x, respectively. Note that these distributed graph engines have 9x more resources than GraphMP.

Due to memory limitation (even though the cluster has more than 1TB memory), Pregel+, PowerGraph and PowerLyra crash when processing UK-2014 and EU-2015. GraphD and Chaos can cope with UK-2014 and EU-2015 from disks. As a comparison, GraphMP can efficiently handle UK-2014 and EU-2015 using just a single machine. With compressed edge caching, GraphMP can manage all edges in a single machine's memory. Therefore, GraphMP avoids costly disk I/O operations, and offers higher performance than GraphD and Chaos. Compared to GraphD, GraphMP could speed up the processing performance by a factor of 23.99, 27.46 and 23.81 to run PageRank, SSSP and CC on EU-2015. Compared to Chaos, the corresponding speedup ratios are 7.95, 8.13 and 8.06, respectively.

5 Conclusion

In this paper, we tackle the challenge of big graph analytics on a single machine. Existing out-of-core approaches have poor performance due to the high disk I/O overhead. To address this problem, we propose a new out-of-core graph processing system named GraphMP. GraphMP partitions the input graph into small shards, each of which could be fully loaded into memory and contains a similar number of edges. Edges with the same destination vertex appear in the same shard. We use three techniques to improve the graph processing performance by reducing the disk I/O overhead. First, we design a vertex-centric sliding window computation model to avoid reading and writing vertices on disk. Second, we propose selective scheduling to skip loading and processing unnecessary shards on disk. Third, we use compressed edge caching to fully utilize the available memory resources to reduce the amount of disk accesses for edges. With these three techniques, GraphMP could efficiently support big graph analytics on a single commodity machine. Extensive evaluations show that GraphMP could outperform GraphChi, X-Stream and GridGraph by up to 30, and can be as highly competitive as distributed graph engines like Pregel+, PowerGraph and Chaos.

GraphMP is designed for graph processing on normal machines or cloud instances without special hardwares. If large flash memory or non-volatile memory is deployed, one may use systems like FlashGraph or Mosaic. When having big memory (for example, in supercomputers), one may use in-memory systems like GraphMat. If high-speed network and big memory machines are available, one may use distributed graph engines like Pregel+ or PowerGraph.

References

[1] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial," *IEEE Access*, vol. 2, pp. 652–687, 2014.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[4] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[5] V. Kalavri, V. Vlassov, and S. Haridi, "High-level programming abstractions for distributed graph processing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 2, pp. 305–324, 2018.

[6] P. Sun, Y. Wen, T. N. B. Duong, and X. Xiao, "Graphh: High performance big graph analytics in small clusters," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017, pp. 256–266.

[7] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM Sigplan Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

[8] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 211–222, 2007.

[9] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[10] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 183–193.

[11] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[12] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015, pp. 39–50.

[13] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 11.

[14] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *Proceedings of Workshop on GRAph Data management Experiences and Systems*. ACM, 2014, pp. 1–6.

[15] T. Zhang, J. Zhang, W. Shu, M.-Y. Wu, and X. Liang, "Efficient graph computation on hybrid cpu and gpu systems." *Journal of Supercomputing*, vol. 71, no. 4, 2015.

[16] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim, "Gts: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 447–461.

[17] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 527–543.

[18] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 2014, pp. 25–28.

[19] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 239–252.

[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[21] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[22] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014.

[23] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 22.

[24] C. Zhou, J. Gao, B. Sun, and J. X. Yu, "Mocgraph: Scalable distributed graph processing using message online computing," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 377–388, 2014.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[26] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 1.

[27] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014, pp. 599–613.

[28] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey, "Graphpad: Optimized graph primitives for parallel and distributed platforms," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 313–322.

[29] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[30] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: scaling graph computation to the trillions," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 408–421.

[31] A. Kyrola, G. E. Blelloch, C. Guestrin *et al.*, "Graphchi: Large-scale graph computation on just a pc." in *OSDI*, vol. 12, 2012, pp. 31–46.

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.

[33] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He, "Venus: Vertex-centric streamlined graph computation on a single pc," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1131–1142.

[34] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning." in *USENIX Annual Technical Conference*, 2015, pp. 375–386.

[35] D. M. Da Zheng, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 45–58.

[36] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 77–85.

[37] D. Yan, Y. Huang, M. Liu, H. Chen, J. Cheng, H. Wu, and C. Zhang, "Graphd: Distributed vertex-centric graph processing beyond the memory limit," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 99–114, 2018.

[38] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 410–424.

[39] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big (ger) graph analytics on a dataflow engine," *Proceedings of the VLDB Endowment*, vol. 8, no. 2, pp. 161–172, 2014.

[40] P. Sun, Y. Wen, T. N. B. Duong, and X. Xiao, "Graphmp: An efficient semi-external-memory big graph processing system on a single machine," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017, pp. 276–283.

[41] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*. ACM, 2015, pp. 1307–1317.

[42] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. AcM, 2010, pp. 591–600.