CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution

Xiangyu Zhi
Xiao Yan*
Department of Computer Science and Engineering, Southern University of Science and Technology
zhixy2021@mail.sustech.edu.cn
yanx@sustech.edu.cn

Bo Tang
Ziyao Yin
Department of Computer Science and Engineering, Southern University of Science and Technology
tangb3@sustech.edu.cn
yinzy2020@mail.sustech.edu.cn

Yanchao Zhu
Minqi Zhou
Gauss Department, Huawei Company
zhuyanchao2@huawei.com
zhouminqi@huawei.com

ABSTRACT

Many systems are designed to run graph algorithms efficiently in memory but they achieve only cache efficiency or work efficiency. We tackle this fundamental trade-off in existing systems by designing CoroGraph, a system that attains both cache efficiency and work efficiency for in-memory graph processing. CoroGraph adopts a novel *hybrid execution model*, which generates update messages at vertex granularity to prioritize promising vertices for work efficiency, and commits updates at partition granularity to share data access for cache efficiency. To overlap the random memory access of graph algorithms with computation, CoroGraph extensively uses *coroutine*, i.e., a lightweight function in C++ that can yield and resume with low overhead, to prefetch the required data. A suite of designs are incorporated to reap the full benefits of coroutine, which include prefetch pipeline, cache-friendly graph format, and stop-free synchronization. We compare CoroGraph with five state-of-the-art graph algorithm systems via extensive experiments. The results show that CoroGraph yields shorter algorithm execution time than all baselines in 18 out of 20 cases, and its speedup over the best-performing baseline can be over 2x. Detailed profiling suggests that CoroGraph achieves both cache efficiency and work efficiency with a low memory stall and a small number of processed edges.

PVLDB Reference Format:
Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, Minqi Zhou. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. PVLDB, 17(4): 891 - 903, 2023.
doi:10.14778/3636218.3636240

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBGroup-SUSTech/corograph.

1 INTRODUCTION

Graphs are ubiquitous in many domains such as social media [13, 15], e-commerce [7], finance [1, 14], and biology [22]. Algorithms are executed on these graphs to support various applications, e.g., community identification [2, 45], recommendation [26], fraud detection [4], and drug discovery [20]. Examples of popular graph algorithms include PageRank (PR), single source shortest path (SSSP), weakly connected component (WCC), and k-core [43]. Many frameworks are designed to execute graph algorithms efficiently, e.g., Ligra [46], Galois [40], Gemini [50], GraphIt [49], and GPOP [31]. In this paper, we consider in-memory graph algorithm frameworks that work on a single machine with multiple cores and focus on reducing algorithm execution time.

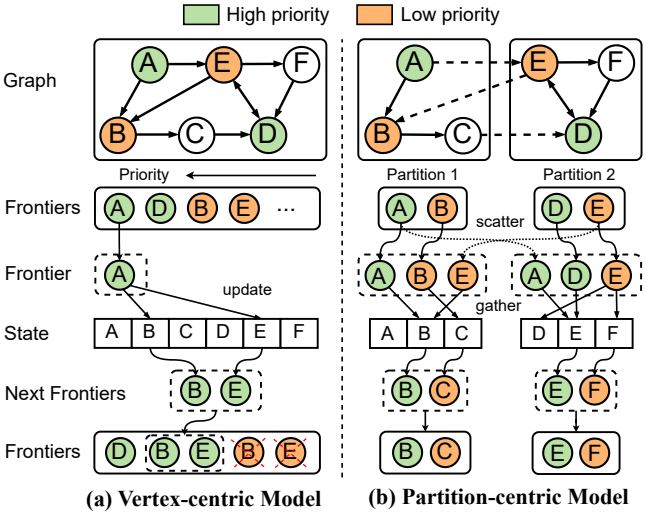

Figure 1: The partition-centric and vertex-centric execution models in existing systems (best viewed in color).

1.1 Motivation and Problem

Graph algorithms generally adopt the following pattern: each vertex v in the graph is associated with a vertex state $s[v]$; there are some active vertices called *frontiers*; algorithm updates the neighbors[1] of each frontier, and these neighbors may become new frontiers. Existing graph algorithm frameworks optimize either the *work efficiency* or *cache efficiency* of graph algorithm execution.

Work efficiency is measured by the number of vertex state update operations conducted during graph algorithm execution [33, 40].

*Xiao Yan is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636240

[1]To be precise, we mean the vertex states of the neighbors for a frontier. For conciseness, we use vertex to denote vertex state when the context is clear.

Since each update is conducted along an edge (i.e., from frontier to target vertex), work efficiency is also quantified by the number of processed edges [33]. Existing systems (e.g., Galois) achieve work efficiency with a *vertex-centric* execution model. The observation is that frontiers make different contributions to algorithm progress, and thus work-efficient graph algorithms (e.g., SSSP, k-core, and WCC) determine a priority value for each frontier to prioritize promising frontiers that accelerate algorithm progress. As shown in Figure 1(a), the vertex-centric model uses a queue to manage the frontiers, and the highest-priority frontier (e.g., A) is obtained from the queue each time. The neighbors of the dequeued frontier (e.g., B and E) are updated, and if a neighbor meets the condition specified by the algorithm after update, it is inserted into the frontier queue, and its lower-priority instances will be pruned.

Cache efficiency is quantified by the ratio of the CPU stall time caused by cache misses over the algorithm execution time (called memory bound [10]) [46, 47]. Since the neighbors of a frontier are usually not consecutive in memory, the neighbor update operation of graph algorithms essentially conducts random memory access. Random access causes both *read amplification* and *cache miss*[2], which result in high memory bound. Existing systems (e.g., GPOP) improve cache efficiency with a *partition-centric* execution model. In particular, the vertex states are organized into partitions that fit in L2 or L3 cache, and processing is conducted at partition granularity. As shown in Figure 1(b), the partition-centric model involves a *scatter* phase and a *gather* phase. In the scatter phase, the frontiers of each partition is copied to the partitions where its neighboring vertices reside. For instance, the neighbors of frontier A span two partitions, so A is copied to both partitions. In the gather phase, each partition collects the states from the scatter phase, updates the vertices within the partition, and gets the frontier for the next iteration. By ensuring that the working partition resides in cache, the partition-centric model avoids memory stall and achieves high cache efficiency.

Table 1 shows the trade-off between cache efficiency and work efficiency in existing graph frameworks. In particular, GPOP has low memory stall but poor work efficiency. This is because its partition-centric model batches computation by partition to share data access and cannot follow the priority order to process individual frontiers. Galois conducts less work but has high memory stall. This is because its vertex-centric model processes each frontier individually and cannot share data access among the frontiers as in the partition-centric model. This fundamental trade-off leads us to the following question–*is it possible to architect a graph algorithm framework that achieves both cache and work efficiency?*

1.2 Our Solution: CoroGraph

Our initial design is to reduce memory stall for the vertex-centric model with a *coroutine-based prefetch pipeline*. In particular, software prefetch allows programs to specify the data to load [9], and coroutines are lightweight functions that can yield and resume with low overhead [19]. In our pipeline, processing tasks are conducted by coroutines, which issue prefetch instructions, switch to the computation tasks of other coroutines, and resume for computation after the required data are fetched to cache. This design overlaps memory access and computation but the performance is unsatisfactory for two reasons. First, memory stall dominates execution time for vertex-centric model (over 70% for Galois in Table 1), and thus the gain is limited even if memory access and computation are perfectly overlapped (e.g., max speedup for Galois is $100/70 \approx 1.43$). Second, the threads have data conflicts when processing different frontiers in parallel because the frontiers share common neighbors, and these data conflicts invalidate prefetched data.

Table 1: Execution statistics of representative graph frameworks and our CoroGraph. The algorithm is SSSP, and the graph is Livejournal. GPOP targets cache efficiency while Galois targets work efficiency. For both memory bound and processed edges (i.e., # of edges), smaller is better.

System	SSSP		
	Memory bound	# of edges (M)	Time (ms)
Ligra	65.3%	569	1270
Gemini	55.2%	573	954
GraphIt	60.5%	145	782
GPOP	32.9%	604	594
Galois	70.2%	89	513
CoroGraph	**28.3%**	**68**	**262**

To tackle the problems above, CoroGraph adopts a novel *hybrid execution model* that combines the benefits of partition-centric and vertex-centric execution. In particular, like vertex-centric execution, the frontiers are managed by a priority queue and processed in the order of their contributions to algorithm progress, resulting in good work efficiency. Meanwhile, like partition-centric execution, updates to vertex states are committed at partition granularity. To accommodate the two mismatching patterns, the frontiers are processed in a scatter phase that generates update messages while the actual updates are conduct in a gather phase. A single thread is assigned to process all updates to a partition, which results in good cache efficiency by sharing data access among update operations. The thread-to-partition update pattern also benefits prefetch by eliminating the data conflicts that invalidate prefetched data.

Beside the hybrid execution model, we tailor a suite of optimizations in CoroGraph. First, we use the aforementioned coroutine-based prefetch pipeline to overlap memory access with computation. Second, we propose a lightweight but effective strategy to switch between prefetch and direct data access because prefetch becomes slower than direct access as a thread gradually warms the cache by processing a partition. Third, since the threads are blocked when waiting to access shared data structures, we adopt a flexible synchronization strategy, which allows the threads to proceed with other tasks instead of blocking. Besides, we also adjust the graph storage format to reduce cache miss and consider the NUMA architecture of modern processors when scheduling the tasks and configuring the prefetch pipeline.

We experiment with 4 popular graph algorithms (i.e., SSSP, k-core, PR, and WCC) and compare our CoroGraph with 5 sate-of-the-art graph algorithm frameworks (i.e., Ligra, Gemini, GPOP, Galois,

[2]Read amplification means that a float or integer vertex state is fetched via 64-byte cache line load while cache miss happens because the automatic hardware prefetch of CPU only loads consecutive data.

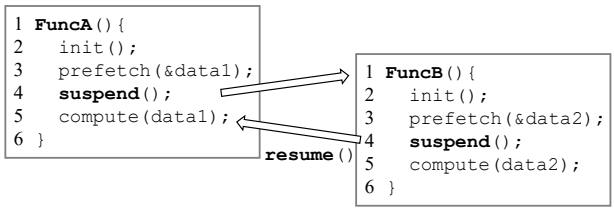

Figure 2: An illustration of coroutine execution.

and GraphIt). The results show that CoroGraph yields shorter algorithm execution time than all baselines in 18 out of 20 cases (i.e., an algorithm plus a graph), and its speedup is usually over 2x. On average, CoroGraph speeds up the best-performing baseline by 2.16x, 2.13x, and 1.77x for SSSP, k-core, and WCC, respectively. Although GPOP comes with specialized optimizations for the PR algorithm, CoroGraph matches its performance for PR. Micro benchmarks show that CoroGraph achieves both cache efficiency and work efficiency, which is indicated by a low memory stall and small number of processed edges in Table 1. We also conduct ablation studies for the designs of CoroGraph, and the results suggest that they are effective in reducing algorithm execution time.

To summarize, we make the following contributions:

- We inspect existing graph frameworks and observe the fundamental conflict between cache efficiency and work efficiency that stems from their execution models.
- We propose a hybrid execution model that combines the benefits of the partition-centric models and vertex-centric execution model in existing graph frameworks.
- We design a suite of efficiency optimizations, e.g., coroutine-based prefetch pipeline, cache-friendly graph format, flexible synchronization, and NUMA-aware task scheduling.
- We architect and open source the CoroGraph framework, which achieves both cache efficiency and work efficiency.

2 BACKGROUND

In this part, we provide background knowledge for our work, which includes the execution models of existing graph algorithm frameworks, software prefetch, and coroutine.

Software Prefetch. Modern processors such as Intel Xeon support hardware prefetch mechanism such as next-line and stride prefetchers, which load data adjacent to the currently accessed memory location [3]. Software prefetch instructions such as *_mm_prefetch* and *__builtin_prefetch (addr)* are more flexible as they allow programs to specify the data to load and has been used to reduce memory stall [41]. However, using software prefetch requires sophisticated implementations such as operation group [9], asynchronous execution [23] and handcrafted state machines [25].

Coroutine. Coroutines are generalizations of functions with two special characteristics. Firstly, coroutines can suspend and resume at specific points during their invocation and return processes. Secondly, each coroutine retains local variables until it is destroyed. As shown in Figure 2, a coroutine (FuncA) can save local variables during execution, suspend, then switch to another coroutine (FuncB), and upon resuming from FuncB, FuncA executes from the suspended point. The C++20 standard introduces stackless coroutine [21], which has low context switch overhead and makes it easy to conduct software prefetch. The new stackless coroutines do not own stacks and run on the call stack of the underlying thread. Coroutine states include local variables that live across suspension points, and are stored in dynamically allocated memory known as coroutine frames. These features make coroutines lightweight. Coroutine switch latency is below 50 ns, which is shorter than last-level cache miss ($\geq$100 ns) and thread switch (several hundred nanoseconds to microseconds).

Many systems use coroutine-based prefetch to hide cache miss [19, 23, 41], and we will discuss them in Section 3. From these systems, we summarize two major design considerations when using coroutines. The first is decomposing the task into coroutines. A task can be decomposed into coroutines in many ways, e.g., some for IO and some for computation. To achieve high efficiency, the decomposition should ensure that the coroutines overlap well with each other and have simple dependencies. The second is scheduling the coroutines. Coroutine switch still has overhead and scheduling is conducted frequently to manage coroutine switches. As such, the scheduling mechanism should conduct switch judiciously and have a low overhead. For our problem, capitalizing coroutine-based prefetch in a general graph algorithm framework poses unique technical challenges (e.g., data conflicts, synchronization overhead, and high memory stall), and a straightforward solution does not work as discussed in Section 1.2.

3 RELATED WORK

Graph algorithm frameworks. Due to the importance of graph algorithms, many systems are designed to execute them efficiently. As a seminal work, Ligra proposes general APIs with *VertexMap* and *EdgeMap* functions to express graph algorithms [46]. It adopts the vertex-centric execution model and can switch between push mode (i.e., frontiers push updates to their neighbors) and pull mode (i.e., vertices pull updates from neighboring frontiers) according to the number of active frontiers. Julienne extends Ligra by using a multi-level priority bucket to manage the frontiers [12] and handles the frontiers in priority order for work efficiency. Both Ligra and Julienne conduct synchronous processing, i.e., handling all current frontiers before generating new frontiers. Galois designs a concurrent multi-level priority queue to manage the frontiers and introduces graph topology-aware work stealing to balance the workloads of the threads [40]. GraphIt [49] automatically generates implementations for graph algorithms that integrates various optimizations, e.g., direction optimization, cache-aware partitioning and data layout. PCPM [32] introduces a specialized optimization for the PageRank algorithm, which avoids writing the edges in the scatter messages using the fact that PageRank conducts updates along all edges. GPOP [30] builds upon PCPM and extends the write optimization to other graph algorithms. In particular, GPOP employs a density-aware mechanism, which scatters the entire partition when there are enough updates in the partition. Gemini [50] supports both single machine and distributed execution. It partitions the graph over different sockets for locality, conducts

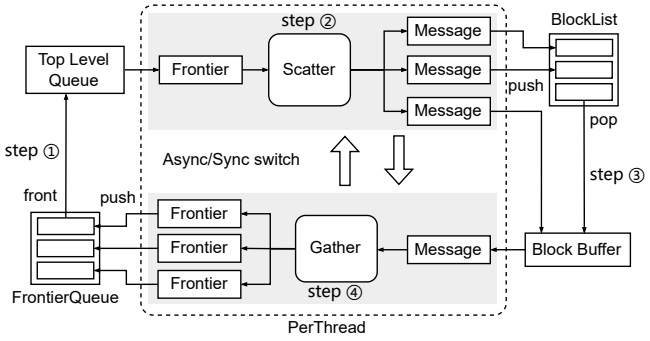

Figure 3: Schematic diagram of our hybrid execution model.

updates with atomic operations, and uses work-stealing over partitions to improve load balance. As discussed in Section 1, these systems optimize either work efficiency or cache efficiency while our CoroGraph achieves both.

ForkGraph divides the graph topology into cache-sized partitions and shares cache-resident graph topology among multiple graph algorithm instances (e.g., SSSP with different target vertices) to reduce cache miss [33]. ThunderRW executes multiple random walk instances and hides memory stall by switching among different random walks [47]. It divides a random walk into stages and uses a state machine to manage prefetch and computation.

There are also graph algorithm systems that consider either distributed execution with multiple machines, e.g., Pregel [34], TAO [6], Trinity [44], PowerGraph [16], or out-of-core execution for disk-resident data, e.g., GridGraph [51], Blaze [24], GraphChi [29], and X-stream [42]. As communication through PCIE or network has high overhead, their primary optimization goals are usually cross-machine or disk-memory communications [16, 17], which are different from the cache miss and work efficiency optimizations in CoroGraph. For example, GraphX [17] uses a vertex-cut to partition the data graph over machines to achieve computation locality and employs a Gather-Apply-Scatter execution pattern to reduce cross-machine data transmission when sending the update messages.

Systems using prefetch and coroutine. As the gap between CPU computation speed and memory access latency continues to widen, software prefetch is widely used to avoid cache miss and improve performance [19, 23, 35, 38, 41]. For instance, prefetch is exploited to accelerate hash joins [39] and index structures [23]. Theses works usually use many prefetch groups to form a pipeline to overlap memory access and computation. AMAC [25] dynamically balances the workloads of the prefetch groups with an asynchronous execution pattern that allows to pause queries and save their states. CORO [41] uses coroutines to implement prefetch pipelines for index joins and adopts succinct implementation due to the properties of coroutine. CoroBase [19] implements an automic coroutine-based prefetch pipeline without changing the API codes of transactions. We use coroutine-based prefetch to accelerate graph algorithm execution, which poses unique challenges that differ from the aforementioned systems, e.g., high memory stall for vertex-centric execution, data conflicts among the threads, and synchronization overhead for shared data structures.

Algorithm 1 Workflow of the hybrid execution model

Input: The graph G = (V, E), the frontier set F
Output: The result Res(V)

```
1:  procedure HYBRID-SCATTER-GATHER
2:    Res(V), FrontierQueue ← INITIALIZE(G, F)
3:    parallel_for_each thread do
4:      while TopQueue ← FrontierQueue.front() do ▷ step ①
5:        SCATTER(TopQueue)                       ▷ step ②
6:        while Block ← BlockList.pop() do        ▷ step ③
7:          GATHER(Block)                         ▷ step ④
8:  function SCATTER(TopQueue)
9:    while frontiers ← TopQueue.pop() do
10:     for v ∈ frontiers do
11:       for u ∈ v.ngh do // traverse neighbors of v
12:         BlockList.push(u, Res[v])
13: function GATHER(Block)
14:   while msg ← Block.buffer.pop() do
15:     for (u, Res[v]) ∈ msg do
16:       if UPDATE(Res[u], Res[v]) do // if u is new frontier
17:         FrontierQueue.push(u, Res[u])
```

4 THE COROGRAPH FRAMEWORK

In this part, we introduce the workflow of CoroGraph, which follows the *hybrid execution model*. Like many existing graph algorithm frameworks, CoroGraph assumes that graph data resides in the main memory of a machine with multi-core CPUs. CoroGraph uses a graph format similar to compressed sparse row (CSR), which allows to identify the out-neighbors of a vertex efficiently.

4.1 Hybrid Execution Model

The hybrid execution model lies in between the vertex-centric model and partition-centric execution model of existing graph frameworks. At a high level, like the partition-centric model, the hybrid model involves a scatter phase that generates messages from the frontiers to their neighbors, and a gather phase that merges the messages and update vertex states at partition granularity. Like the vertex-centric model, the hybrid model manages the frontiers using a priority queue and conducts scatter for the frontiers following their priority order.

Figure 3 illustrates the architecture of the hybrid execution model, Algorithm 1 describes its procedure, and Figure 4 shows the scatter phase and gather phase with more details. CoroGraph executes 1-4 steps iteratively. In particular, the neighbor vertices are organized into blocks by a range partition over their IDs, and CoroGraph ensures that a block fits in L2 cache such that it can be loaded into cache when conducting updates. CoroGraph uses a *frontier queue* and a *block list* to schedule scatter tasks and gather tasks, respectively. The frontier queue contains the frontiers and sorts them by priority. We use the multiple-level queue [40], which organizes frontiers with the same priority as a level and maintains a pointer points to the highest level queue (step 1 of Algorithm 1). The block list is a linked list that consists of pointers to the block buffers, and each block buffer contains the messages sent from the frontiers to a block. The threads retrieve tasks from the frontier

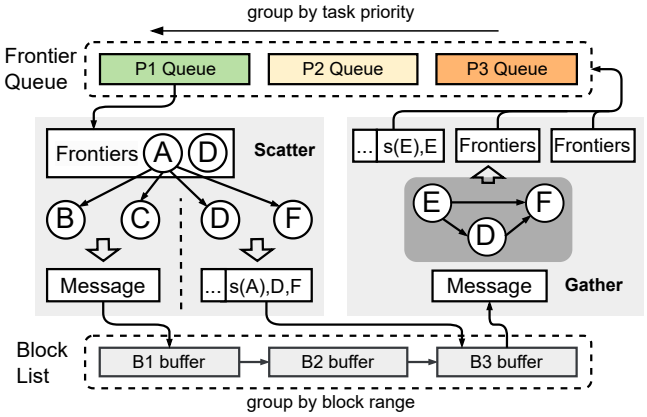

Figure 4: The scatter phase and gather phase of our hybrid execution model.

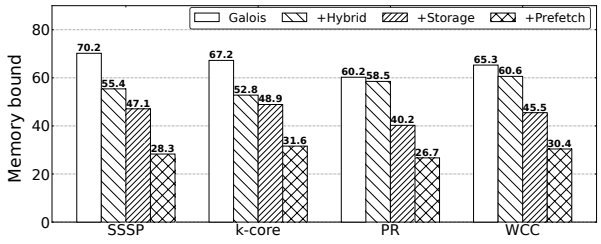

Figure 5: Memory bound profiling for different graph algorithms on the Livejournal graph with our optimizations.

queue and block buffers at chunk granularity for execution, and each chunk contains 2,048 vertices by default.

As shown in Algorithm 1, CoroGraph iterates over scatter phase (step 1 and 2) and gather phase (step 3 and 4) until the frontier queue becomes empty, which signals algorithm termination. In the scatter phase, a thread pops the highest level of the frontier queue (i.e., TopQueue in Line 4) and conducts scatter for these frontiers a chunk each time (i.e., Lines 9-12). As shown in the bottom of Figure 4, each block has a global message buffer to receive the update messages scattered from all the threads, which is called block buffer. For each frontier v, scatter messages are written to the block buffers containing v's neighbors correspondingly. In order to avoid the synchronization costs caused by frequent block buffer writes, each thread has a local message buffer for each block, which is of chunk size. A thread spills its scatter messages to a local buffer and copies the local buffer to the block buffer when the local buffer is full. After a thread copies its local messages to an empty block buffer, the pointer to the buffer is appended to the tail of the block list such that the buffer can be retrieved in the gather phase.

A thread continues to scatter until exhausting the frontiers it obtains from the frontier queue (i.e., TopQueue), and then proceeds to the gather phase. In the gather phase, the thread retrieves a pointer from the block list, which corresponds to the global message buffer of a block. A chunk of messages are consumed each time to conduct compute and update the vertex states in the block. A block is processed by a single thread until its message buffer becomes empty, and the thread also determines the new frontiers in the block and pushes these frontiers to the frontier queue. We use two pointers to indicate the head and tail of a message buffer and allows other threads to append messages at the tail while a thread consumes messages at the head. We do not observe straggler issues, where a block has much more gather work than the other blocks and stalls the gather stage. Note that in our hybrid execution model, the threads may be in different phases, i.e., some threads are conducting scatter while the other threads are conducting gather.

Discussions. The hybrid execution model attains the work efficiency of vertex-centric execution by processing the frontier in the order of their priority. Hybrid execution also enjoys the cache efficiency of partition-centric execution by conducting gather at partition granularity. This is because each block fits in L2 cache and is updated by a single thread, and thus vertex states fetched to the cache can be reused when the thread processes the same vertices again. Block-based update also mitigates read amplification because vertex states piggybacked in a cache line load may also be used by subsequent update operations. Moreover, using a single thread to update each block eliminates data conflicts, enabling efficient lock-free update and avoiding invalidating prefetched data for the coroutine-based prefetch pipeline in Section 5.1.

4.2 APIs and Modes

CoroGraph extends the API of Ligra, which is popular due to its conciseness and expressiveness. In particular, Ligra allows users to define two core functions, i.e., *VertexMap* for initializing and filtering the frontiers, and *EdgeMap* for updating the neighbors of a frontier. CoroGraph requires users to define the scatter function and gather function in *EdgeMap*. CoroGraph also requires a priority function to calculate the priority of a frontier. For most graph algorithms, logic of the functions required by CoroGraph is straightforward and can be found in the literature. Programming CoroGraph is as easy as Ligra, for instance, SSSP takes 79 lines of code (LoC) in Ligra and 41 LoC in CoroGraph. We also provide built-in implementations of popular graph algorithms such SSSP, PageRank, and k-core in our open-source code.

CoroGraph supports two modes to cater for the characteristics of different graph algorithms, i.e., *synchronous* and *asynchronous*. In particular, synchronous graph algorithms (e.g., Bellman Ford, PageRank, and WCC) require to process all frontiers before updating the vertex states while asynchronous algorithms (e.g., Δ-step, k-core) do not have this requirement. In asynchronous mode, CoroGraph executes the scatter phase and gather phase asynchronously and allows the threads to be in different phases as discussed in Section 4.1. In synchronous mode, CoroGraph first assigns the threads to conduct the scatter phase and only enters the gather phase after all threads finish scatter. Instead of copying the local message buffers of each thread to the global buffers of each block, the gather phase of synchronous mode directly accesses the local buffers to conduct update in order to reduce one message copy.

5 EFFICIENCY OPTIMIZATIONS

Figure 5 profiles the memory bound of our hybrid execution model and compares with Galois, which adopts the vertex-centric execution model. The results show that memory stall is reduced by the

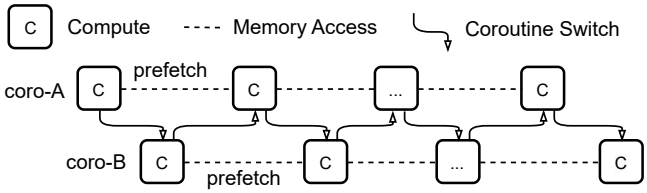

Figure 6: Coroutine-based prefetch pipeline.

hybrid model but still takes up over 50% of the execution time. Moreover, we also observe that the thread synchronization cost increases with the number of threads and limits scalability. In this part, we elaborate the optimizations to resolve these efficiency problems, including coroutine-based prefetch pipeline and cache-friendly graph format to reduce memory stall, and flexible synchronization to reduce thread synchronization cost.

5.1 Coroutine-based Prefetch Pipeline

There are two main sources of cache misses that cause memory stall in our hybrid execution model. In the scatter phase, cache miss occurs when reading the edges of a frontier v to determine the blocks to scatter v's messages. This is because a pointer chasing is required to read the edges of a vertex when the graph is stored using the CSR format. In the gather phase, cache miss occurs when reading the vertex states to conduct update because the neighbors of a vertex may not be in the cache. To hide these cache misses, we design a coroutine-based prefetch pipeline to issue software prefetch instructions for required data.

Figure 6 shows the prefetch pipeline for the gather phase, and the design is similar for the scatter phase. In particular, each thread runs two coroutines, and a coroutine handles a group of vertex states and separates the memory access and computation tasks. The group size is set as 64 by default. In the pipeline, coroutine A first issues software prefetch instructions for the required vertices and yields without waiting; then the thread switches to coroutine B, which also issues prefetch instructions and yields; after that, the thread executes the computation task of coroutine A, for which data has been loaded to cache. By running the two coroutines continuously to process different vertices, the pipeline overlaps the prefetch instructions of one coroutine with the computation tasks of another coroutine. More coroutines are used for prefetch in other systems [39, 47]. However, we observe that using 2 coroutines already provides good overlap for our case, and using more coroutines degrades performance due to coroutine switch overhead, which is shown by experiments in Section 7.3.

Cache-aware memory access. Prefetch hides cache miss but incurs the overhead of coroutine switch. When a thread conducts gather for a block, it gradually warms the cache by loading the vertices. As such, we observe that after conducting some gather tasks, the coroutine-based prefetch pipeline runs longer than directly reading data (although it stalls upon cache misses). This is because the portion of cache miss in the memory accesses reduces as the cache warms, and the overhead of coroutine switch outweighs the memory stall caused by cache miss. Therefore, we need to switch from coroutine-based prefetch to direct data access based on cache

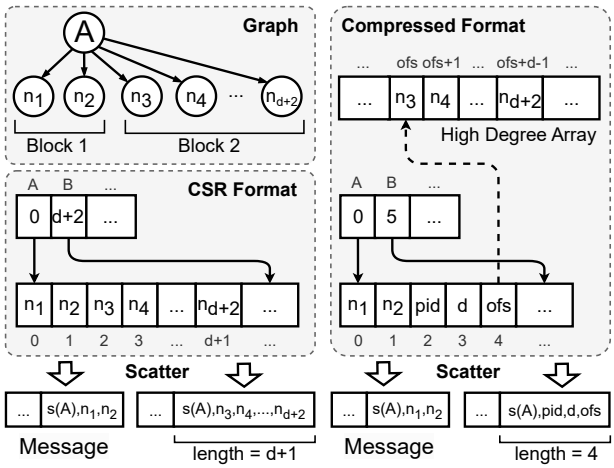

Figure 7: Storing the degree and offset for high degree vertex A in block 2 in the cache-friendly graph format.

status. However, recording the vertices that are in the cache for each block is expensive.

To tackle this problem, we record the number of processed vertices for each block (denoted as n_p) as a surrogate. This is lightweight because it only requires to count the vertices involved in the gather tasks. When n_p exceeds a threshold n_t, we switch to direct data access. We observe that setting n_t as 1-2 times of the number of vertices in a block (denoted as n_b) yields good performance and use $n_t = n_b$ by default. This strategy can be analyzed as a classical balls into bins problem [37] by modeling the vertices as bins and accesses as balls. Assume the accessed vertices are distributed uniformly and independently among the vertices in a block, then the expected portion of vertices that are not in cache is e^{-n_p/n_b} after processing n_p vertices. When $n_p > n_b$, a small e^{-n_p/n_b} suggests that most vertices are already in the cache.

5.2 Cache-friendly Graph Format

We also tailor the graph storage format to reduce cache miss. In particular, we modify the CSR format to store some edges for each vertex in the offset array. Each element of the offset array is sized to a cache line instead of a single integer and thus allows to pack some edges (plus possible weights) with the offset. With the original CSR format, fetching the neighbors of a vertex always requires to first read the offset array and then the edge array, which incurs one cache miss. With our inline edge storage, the edges of low-degree vertices are fetched via a single cache line read to the offset array.

In the scatter phase, each frontier writes its out-neighbors and updates to the block message buffers, and the gather phase reads these messages to conduct update. Such write and read have high memory traffic when a frontier has many neighbors in a block. To reduce the memory traffic, we adopt the compressed graph storage format in Figure 7 (note that in-line edge storage is not shown there). In particular, if a vertex has more than n_c (set as 2 by default) out-neighbors in a block, we record an *offset* that indicates the start position of these edges in an high degree array and a *count* for these edges. In the scatter phase, instead of writing these edges, we

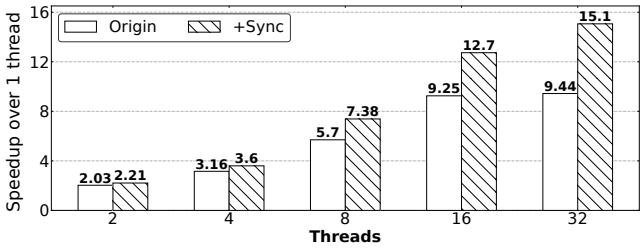

Figure 8: Thread scalability for SSSP algorithm on Friendster graph with and without flexible synchronization.

write the offset and edge count to the message buffer; the gather phase reads the actual edges using the offset and edge count. To differentiate the offset and count values from the normal edges in the gather phase, we use the most significant bit (MSB) of edge ID, which is set as 0 for normal edges. Accessing the compressed edges incurs one cache miss, and we hide it with coroutine-based prefetch. Note that converting the graph from the CSR format to our storage format requires only a linear scan. Moreover, the conversion overhead can be amortized by many runs of graph algorithms.

5.3 Flexible Synchronization

Our hybrid execution model has several shared data structures that can only be accessed by a single thread at a time. For instance, the threads need to acquire lock to pop frontiers from and push frontiers to the frontier queue. When a thread cannot acquire its required lock, it can either wait or yield the CPU core to another thread. The yield approach incurs thread context switch and evicts the data loaded by a thread, which harms cache efficiency. Using the wait approach, the thread block time increases with the number of threads and results in inferior thread scalability as shown in Figure 8. To tackle this problem, we adopt a flexible synchronization strategy, which allows the threads to proceed with other tasks when they cannot acquire locks for shared data structures.

We create a coroutine to handle the synchronization tasks (called sync coroutine) as shown in Figure 9. In particular, the sync coroutine is responsible for pushing the scatter messages to block buffers in the scatter phase and adding the new frontiers to the frontier queue in the gather phase. If the sync coroutine cannot acquire locks, it suspends and switches to the prefetch coroutines in Section 5.1. If the sync coroutine is pending, each time the prefetch coroutines suspend, it retries to acquire the locks. In Algorithm 1, we conduct synchronization after the thread completes a chunk of tasks (e.g., with 2048 vertices). To account for the synchronization delay caused by retry, we run the sync coroutine with finer granularity, i.e., every time the prefetch coroutines finish a group of tasks (e.g., with 64 or 128 vertices). If there are unsynchronized data after the thread completes a chunk of tasks, the sync coroutine loops until it successfully synchronizes these data. This ensures that there is no outdated synchronization and preserves work efficiency. As shown in Figure 8, flexible synchronization can improve scalability by more than 50% when using 32 threads. In Section 7.2, we show CoroGraph matches the scalability of existing graph frameworks.

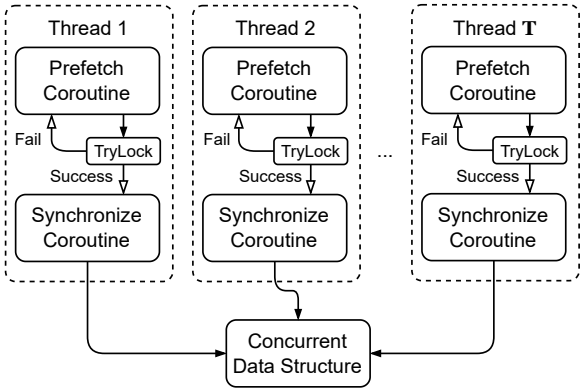

Figure 9: Flexible synchronization with coroutine

5.4 NUMA-aware Scheduling and Prefetch

Modern processors usually have multiple sockets, and the latency and bandwidth of intra-socket memory access are usually much better than inter-socket memory access. For instance, on our experiment platform, inter-socket memory access latency is about 2x of intra-socket latency. We adapt for the NUMA architecture with two designs, i.e., locality-aware scheduling and delay-aware prefetch.

We pin each block and thread to a NUMA node. The global message buffer of a block is allocated on the same NUMA node as the block, and we use a separate block list to manage the gather tasks for the blocks on each NUMA node. When allocating the gather tasks, we try to co-locate the thread and its assigned block on the same NUMA node. In our prefetch pipeline, the group size (i.e., number of vertices handled by a coroutine) controls the number of in-flight prefetch instructions and the size of the computation task. If the group size is too small, data may not have arrived at the cache yet when computation finishes; if the group size is too large, prefetched data may be evicted due to possible cache contention. Thus, we use different group sizes (i.e., 64 and 128 by default) for intra-socket and inter-socket prefetch to account for the difference in memory access latency.

6 IMPLEMENTATION

CoroGraph is implemented in C++ using about 4000 lines of code. We use the Galois parallel library [40] for thread parallelization and C++20 standard coroutine library [21] for coroutine-based prefetch and synchronization. To avoid the costs of frequent coroutine creations, we assign each thread to initialize the coroutines when an algorithm starts and invoke the coroutines as function calls to shared variables.

The data structures of CoroGraph are adapted from existing ones. In particular, the frontier queue is based on a concurrent priority queue called *obim* in Galois. With *obim*, each thread has a local map of the global priority queue's local queues for fast access. We modify the interface of *obim* such that the pop and push operations are conducted in chunks instead of individual frontiers in order to reduce the synchronization overheads. The block list is also implemented as a concurrent queue. The message buffer of each block is implemented as a concurrent linked list with chunks as the

Table 2: Statistics of the graphs used in the experiments.

Graph	Abbrv.	Vertices	Edge	Avg. Deg
Livejournal [5]	LJ	5 M	69 M	13.8
Orkut [27]	OR	3 M	234 M	76.2
RMAT-24 [8]	RM	16 M	520 M	32.5
Twitter [28]	TW	41 M	1469 M	35.8
Friendster [27]	FT	125 M	3612 M	28.9

elements, and each element stores a pointer to the next element. The block list only records pointers to the head and tail of each block message buffer. To access the block buffers locally, each thread also keeps a local copy of all pointers to the block buffers.

The block list has a subtle concurrency issue that may cause data conflicts among the threads. In particular, when a thread pops the last element from its responsible block buffer for gather, another thread may push a chunk of scatter messages into the same buffer. As the buffer is empty at the moment, the thread pushing the chunk will add this buffer to the block list, and another thread may obtain this buffer for gather task. This validates our requirement that the gather tasks of a block can only be conducted by one thread at a time. To tackle this problem, we use one bit to indicate the state of each block buffer. The bit is set to 1 if its block buffer is in the block list or there is a thread conducting gather task for the block. Only after a thread consumes all messages in a block buffer, it sets the bit to 0 such that the block can be inserted into the block list again for the other threads to conduct gather tasks.

For each vertex, we store its neighbors and edge weights together. To reduce graph size, we utilize the fact that the number of neighbors for a vertex in a block cannot exceed the block size (e.g., 2^{18}). As such, we pack the most significant bit (MSB), vertex degree, and block ID in a 4-byte integer with the format MSB|deg|bid. The MSB is used to indicate whether the vertex records actual edges or offest in our cache-friendly graph format in Section 5.

7 EXPERIMENTAL EVALUATION

We experiment to answer the following questions.

- *How does the algorithm execution time of CoroGraph compare with state-of-the-art graph algorithm frameworks?*
- *Does CoroGraph achieve its design goals and attain both cache efficiency and work efficiency?*
- *How effective are the designs of CoroGraph in reducing algorithm execution time?*

In this part, we introduce the experiment settings in Section 7.1, compare CoroGraph with state-of-the-art baselines in Section 7.2, and evaluate our designs in Section 7.3.

7.1 Experiment Settings

Datasets. We utilize 5 graphs in the experiments, and their statistics are reported in Table 2. In particular, *Livejournal*, *Orkut*, *Twitter*, and *Friendster* model social networks and are widely used to evaluate graph algorithm frameworks [31, 40, 46]. RMAT-24 is a synthetic graph created using the famous RMAT generator [8]. Following [12, 49], we randomly generate edge weights in the range of [0, 100) for the graphs. We intentionally choose these graphs for diversity, i.e., their edges range from tens of millions to billions and their average degrees vary. For conciseness, we refer to the graphs using abbreviations.

Table 3: Configurations of the two experiment machines.

Item	Parameter	Server A	Server B
Core	L1 cache	32 KB	32 KB
	L2 cache	1 MB	1 MB
	Frequency	3.6 GHz	2.1 GHz
Socket	L3 cache	16 MB	36 MB
	Core count	4	26
Memory	Size	512 GB	640 GB
CPU	Model	Gold 5122	Gold 6230R

Baselines and algorithms. We compare our CoroGraph with 5 graph algorithm frameworks, i.e., Ligra [46], Gemini [50], GPOP [30], Galois [40], and GraphIt [49], which represent the state-of-the-art. In particular, Ligra is a classical graph framework and adopts the vertex-centric execution model. Galois optimizes vertex-centric execution with efficient management of the frontiers and good load balance of the threads. Both GPOP and Gemini adopt the partition-centric execution model to improve cache efficiency but GPOP also incorporates specialized optimizations for the PR algorithm to reduce memory traffic. GraphIt automatically generates C++ code for graph algorithms with optimizations such as push/pull selection and cache-aware partitioning. We do not compare with other graph systems such as GraphMat [48] and Grezelle [18] because they are outperformed by our baselines [30, 49].

We experiment with 4 popular graph algorithms, i.e., *SSSP*, *k-core*, *PageRank (PR)*, and *weakly connected component (WCC)*. When an algorithm has multiple variants, and we choose the variant that yields the shortest execution time for each system. In particular, for SSSP, we use the Bellman–Ford algorithm [11] for Ligra, Gemini, and GPOP, and the work-efficient Δ-step algorithm [36] for GraphIt, Galois, and CoroGraph. We ensure that all systems process the same target vertices for SSSP such that they conduct the same tasks. For PR, Ligra, GPOP, and Galois use the vanilla PR algorithm [11] while Gemini, GraphIt, and CoroGraph use the work-efficient PR-delta algorithm [46]. For both k-core and WCC, all systems adopt the same algorithm variant. Our evaluation methodology follows [30] and is fairer than using the same algorithm variant for all systems because the systems suit different algorithm variants due to their execution model and specific optimizations. Note that Δ-step and k-core are asynchronous algorithms while the other algorithms are synchronous, and synchronous algorithms require all scatter tasks to finish before gather for each round.

Platform and metric We use 2 different machines for the experiments, and their configurations are listed in Table 3. The memory type is DDR4. Both servers run on Linux 4.15.0, and we disable hyper-threading in all experiments. All code is compiled using g++ 7.5.0 with the -O3 flag. We conduct the experiments on server A unless stated otherwise. By default, we use 8 threads for the experiments on server A and server B. We use algorithm execution

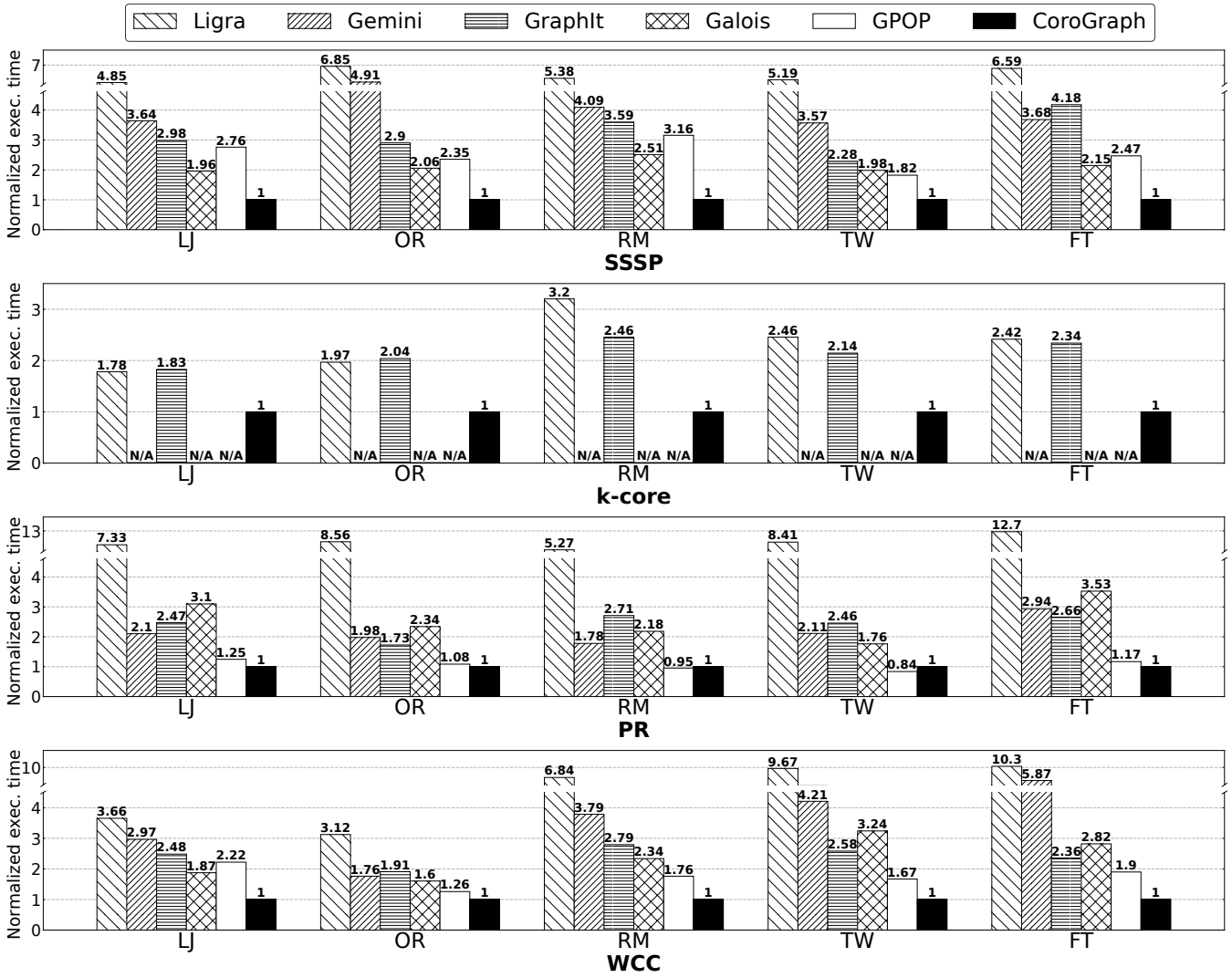

Figure 10: Algorithm execution time of CoroGraph and the baseline systems, normalized by the execution time of CoroGraph.

time as the main performance metric and report the median in 5 runs for an algorithm on one graph. As micro metrics, we measure the memory stall using VTune to indicate cache efficiency and the number of processed edges to indicate work efficiency. To avoid the influence of the micro measurements on execution time, we measure execution time and the micro metrics in separate runs. We report three effective numbers for the experiment results.

7.2 Main Results

Figure 10 reports the algorithm execution time of the systems. For each case, i.e., algorithm plus graph, we normalize the execution time of the baseline systems by dividing the execution time of CoroGraph. This is because for an algorithm, the execution time can vary by scale for the graphs due to the difference in their sizes, which will make Figure 10 difficult to read if we use the actual execution time. Gemini, GPOP, and Galois do not support the k-core algorithm, and thus their results are missing. We make the following observations.

CoroGraph provides strong performance and outperforms all baselines in 18 out of the 20 cases. In particular, compared with the best-performing baseline, the speedup of CoroGraph is up to 2.51x (i.e., for SSSP on the RM graph) and over 1.67x in 15 out of the 20 cases. Considering different algorithms, the average speedup of CoroGraph over the best-performing baseline are 2.16x, 2.13x, and 1.77x for SSSP, k-core, and WCC, respectively. For PR, GPOP is the strongest baseline because it incorporates specialized optimizations for PR to reduce memory traffic. However, CoroGraph matches its performance for PR with 3 speedups and 2 slowdowns in the 5 graphs, and the slowdown is moderate (i.e., peaks at 16% for the TW graph). Overall, CoroGraph achieves larger speedup over the baselines for SSSP and k-core than PR and WCC. This

Table 4: Execution statistics of the systems on the Orkut Graph. The number of processed edges (i.e., #Edges) are in millions, and the execution time (i.e., Time) is in millisecond. The best value is marked in bold for each column.

System	SSSP			k-core			PR			WCC		
	MemB	#Edges	Time	MemB	#Edges	Time	MemB	#Edge	Time	MemB	#Edge	Time
Ligra	65.21%	1676	4539	60.32%	234	3160	69.52%	1346	5822	68.45%	474	1050
Gemini	48.67%	1697	3254	N/A	N/A	N/A	55.25%	2343	1208	59.24%	471	592
GraphIt	55.21%	378	1792	52.89%	234	2760	57.29%	1346	1173	62.93%	473	642
Galois	67.52%	353	1331	N/A	N/A	N/A	62.39%	2343	1565	66.35%	468	539
GPOP	35.44%	969	2138	N/A	N/A	N/A	**22.39**%	2343	738	33.22%	1363	423
CoroGraph	**28.25**%	**336**	**663**	**29.02**%	**234**	**1606**	27.38%	**1346**	**680**	**30.35**%	**465**	**336**

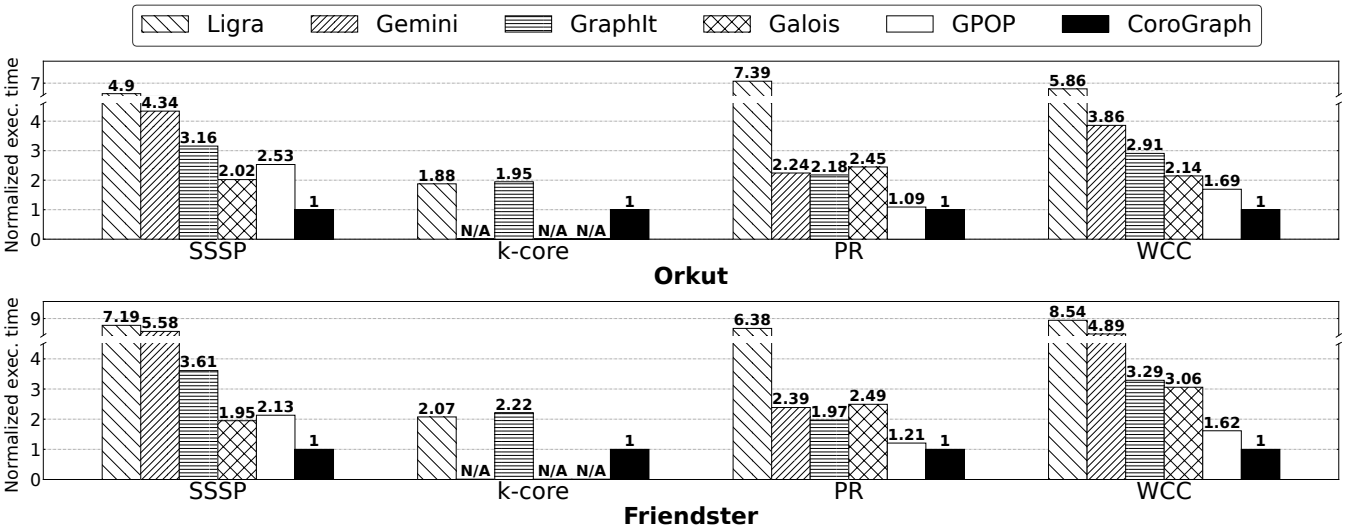

Figure 11: Algorithm execution time on another server, normalized by the execution time of CoroGraph.

is because SSSP and k-core are asynchronous algorithms, which conduct scatter and gather simultaneously and have more conflicts when accessing the shared data structures. The flexible synchronization of CoroGraph avoids thread block in these conflicts. We observe no obvious pattern for the speedup of CoroGraph across different graphs, which suggests that the performance advantages of CoroGraph do not depend on data characteristics.

There is no single winner among the baseline systems. For instance, considering SSSP, Galois generally performs well but is 1.1x slower than GPOP for the TW graph. Ligra performs the worst in most cases but is the best-performing baseline for k-core on the LJ graph. Thus, the consistently good performance of CoroGraph may eliminate the burden of choosing the most efficient system for different graphs and algorithms. Moreover, beside the execution model, the particular optimizations of a system are also important for performance. For instance, both Galois and Ligra adopt the vertex-centric execution model but Galois usually performs much better than Ligra. This is because Galois incorporates optimizations for frontier management and thread workload balancing. As we will show in subsequent experiments, the strong performance of CoroGraph is also attributed to both the right execution model and the effective optimizations.

Table 4 reports the execution statistics of the systems for the Orkut graph, and the observations are similar on the other graphs. These statistics help understand the query time results in Figure 10. In particular, CoroGraph achieves both cache efficiency and work efficiency with a low memory stall and a small number of processed edges, which explains its short algorithm execution time. For the PR algorithm, GPOP has even lower memory stall than CoroGraph due to its algorithm specific optimizations, which explains its strong performance for PR in Figure 10. Table 4 also echos our observation in Section 1, i.e., existing systems only achieve either cache efficiency or work efficiency. For instance, GPOP has low memory stall but processes more edges than the other systems while Galois processes a small number of edges when using work-efficient algorithm variants (i.e., for SSSP and WCC) but suffers from high memory stall.

Results on another machine. In Figure 11, we report the algorithm execution time of the systems on server B to validate the generality of CoroGraph across platforms. We use Orkut and Friendster as representatives of the small and large graphs respectively and note that the observations are similar on the other graphs. Figure 11 shows that the results on server B resemble the results on

Table 5: Algorithm execution time (in seconds) when using different number of threads on server B, the graph is Friendster.

Threads	SSSP			k-core			PR			WCC		
	Galois	GPOP	Coro	Ligra	GraphIt	Coro	Galois	GPOP	Coro	Galois	GPOP	Coro
1	75.13	85.09	**71.56**	278.2	265.7	**214.3**	95.85	83.12	**75.92**	42.38	46.72	**41.02**
2	37.00	60.56	**32.37**	177.1	159.6	**96.70**	61.92	41.20	**35.93**	28.92	35.75	**22.32**
4	23.80	25.33	**19.89**	91.53	87.19	**51.67**	30.28	20.78	**18.15**	20.87	15.08	**11.35**
8	13.18	14.84	**9.69**	46.70	43.28	**25.29**	21.86	10.45	**8.93**	12.01	9.48	**5.83**
16	8.12	11.86	**5.62**	25.19	23.12	**13.56**	17.56	8.61	**7.52**	6.99	6.58	**3.52**
32	7.96	9.49	**4.75**	17.24	15.91	**8.31**	13.29	7.66	**6.43**	3.86	6.23	**2.72**

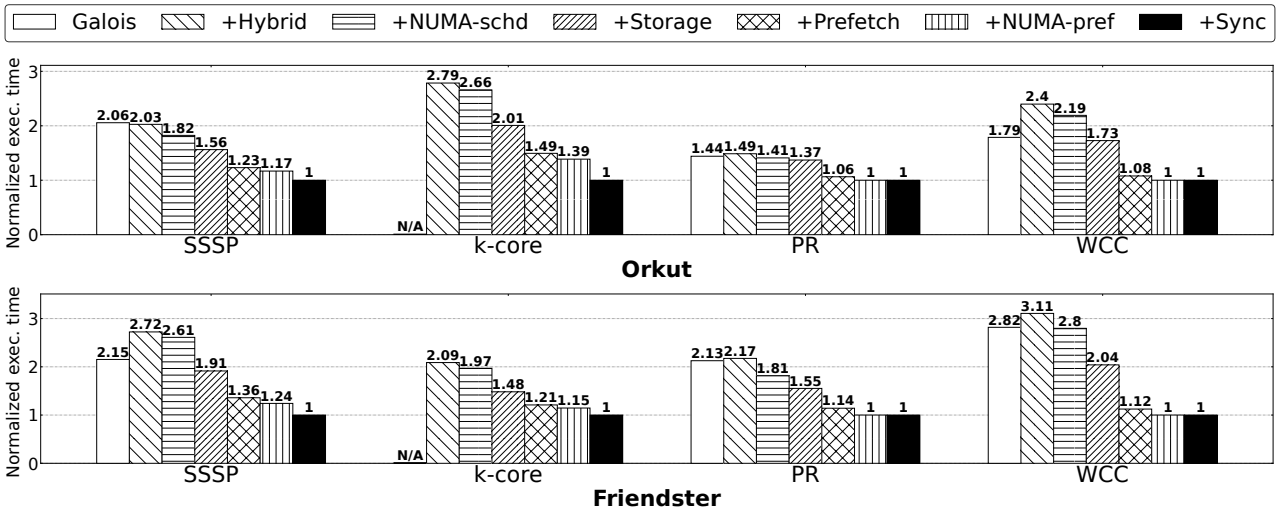

Figure 12: Ablation study of our designs, where we incrementally adds our key designs to CoroGraph.

server A (i.e., Figure 10), i.e., CoroGraph provides strong performance and its speedups over the baselines are large (usually over 2x) except for the PR algorithm.

Multi-thread scalability. Table 5 reports the algorithm execution time of the systems when using different number of threads on server B. For each algorithm, we only include the two best-performing baselines because the other baselines perform much worse. The results show that CoroGraph consistently outperforms the baselines when using different number of threads. Moreover, all systems achieve a speedup around 10x when increasing from 1 thread to 32 threads but CoroGraph sometimes exhibits better thread scalability. For instance, when increasing from 1 thread to 32 threads for SSSP, the speedup of Ligra, GraphIt, and CoroGraph are 9.43x, 8.97x, and 15.06x, respectively. All systems scale almost linearly before using 4 or 8 threads but much worse afterwards. This is because the computation in graph algorithms is lightweight, and thus the memory bandwidth is saturated with a small number of threads [31].

7.3 Ablation Study and Parameter Settings

Ablation study. Figure 12 conducts ablation study for our key designs to understand their contributions to performance. In particular, *Hybrid* means our hybrid execution model, *Storage* refers to the cache-friendly graph format, *Prefetch* stands for the prefetch pipeline, and *Sync* is the flexible synchronization. We also include Galois as a baseline, which adopts a vertex-centric execution model and serves as the starting point of our optimizations. We make the following observations.

First, simply using the hybrid execution model offers limited gain and may even degrade performance compared with Galois. This is because although the hybrid execution model reduces memory stall as shown in Section 5, it incurs the overhead of generating the scatter messages. In contrast, the vertex-centric model directly applies updates without generating the scatter messages. However, we note that the hybrid execution model is crucial for CoroGraph because it eliminates thread data conflicts and serves as the foundation of other optimizations (e.g., prefetch and synchronization).

Second, our three key optimizations, i.e., cache-friendly graph format, prefetch pipeline, and flexible synchronization, are all effective in reducing algorithm execution time. The gain of the prefetch pipeline is the most significant and can be more than 2x (e.g., for SSSP and WCC on the Friendster graph). Cache-friendly graph format is effective in all cases and usually reduces execution time by one third. Flexible synchronization improves the asynchronous algorithms (i.e., SSSP and k-core) but has no effect on the synchronous algorithms (i.e., PR and WCC) because the synchronous algorithms

Table 6: Influence of parameters on the Friendster graph.

(a) Vertex state block size $|B|$.

$\|B\|$	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
SSSP	8.76	6.39	5.14	**4.75**	6.58
WCC	6.34	4.59	3.17	**2.71**	5.39

(b) Degree threshold for write optimization.

n_c	0	2	4	6	8	10
SSSP	5.30	**4.75**	4.97	5.42	5.89	6.42
WCC	2.88	**2.71**	2.89	3.02	3.33	3.86

(c) Chunk size for task execution.

Size	128	256	512	1024	2048	4096
SSSP	5.36	4.91	**4.75**	5.15	5.64	6.04
WCC	3.25	2.94	2.85	2.79	**2.71**	2.71

(d) Number of coroutines in the prefetch pipeline.

Number	1	2	3	4	5
SSSP	5.02	**4.75**	5.23	5.97	6.45
WCC	2.80	**2.71**	2.79	2.94	3.56

only conduct synchronization when all threads finish scatter and have fewer data structures conflicts.

Third, our NUMA-aware optimizations also improve performance. In particular, 'NUMA-schd' and 'NUMA-pref' refer to NUMA-aware scheduling and NUMA-aware prefetch, respectively. The results show that NUMA-aware scheduling and prefetch consistently reduce algorithm execution time across the datasets and algorithms. The speedup of NUMA-aware scheduling is larger for the Friendster graph (i.e., 1.2x-1.4x) than the Orkut graph (i.e., 1.1x-1.2x) because Friendster is larger and requires more cross-NUMA data access to move the vertex states during algorithm execution. The speedup of NUMA-aware prefetch ranges from 1.05x to 1.1x.

Parameters. Table 6 reports the execution time of CoroGraph for the Friendster graph when using different parameters. We use SSSP and WCC as representatives for asynchronous and synchronous algorithms, respectively, and note that the observations are similar on the other graphs.

Table 6(a) shows that both algorithms run the fastest when the block size of vertex state (i.e., $|B|$) is 2^{18}. This is because each vertex state takes 4 bytes and 2^{18} allows a vertex block to fit in the 1MB per-core L2 cache of server B. Execution time increases when block size deviates from the optimal value. This is because larger block does not fit in L2 cache and thus the loaded data may be swapped out while smaller block reduces the opportunity for cache sharing. Table 6 also shows that cache eviction has higher penalty than reduced sharing opportunity.

Recall that we use n_c as the threshold to determine the high-degree vertices for our write optimization in the scatter phase, and the high-degree vertices write an offset and edge count instead of the actual edges. Table 6(b) shows that both algorithms achieve the shortest execution time with $n_c = 2$. This is because when n_c is too small, writing the offset and edge count is more expensive than directly writing the edges; while when n_c is too large, writing many edges becomes more expensive than the offset and edge count. The execution time does not degrade much unless n_c is far from optimal because the costs of the two options (i.e., writing edges and writing offset) transit smoothly.

We use a chunk as the granularity for executing the scatter tasks and gather tasks, and chunk size is the number of vertices in a chunk. Table 6(c) shows that both SSSP and WCC run the fastest with an intermediate chunk size and become slower when the chunk size deviates from the optimal. This is because chunk size balances synchronization overhead and work efficiency. For instance, a small chunk size means fine-grained coordination, which benefits work efficiency but increases synchronization overhead. SSSP requires a smaller optimal chunk size than WCC because work efficiency is more important for it.

Table 6(d) reports the algorithm execution time when using different number of coroutines. As we have discussed in Section 5, CoroGraph does not benefit from using a larger number of coroutines as in other works, and 2 coroutines suffice because memory access and computation are already overlapped. Execution time jumps when increasing from 4 coroutines to 5 because with a large number of prefetch operations, the data loaded latter may evict the data loaded earlier. Using one coroutine also allows to hide some memory access costs because we conduct a group of tasks (with each task consists of prefetch and compute) in a coroutine, and the prefetch operation of a task may finish when we issue all prefetch operations and come back to conduct computation for the task.

Overall, Table 6 shows that the parameters in CoroGraph are easy to configure because (i) the parameters adopt a single optimal value and (ii) the optimal value generalizes for different algorithms.

8 CONCLUSION

We observe the trade-off between cache efficiency and work efficiency in existing graph algorithm systems, and find that existing systems cannot achieve both due to the fundamental limits in their execution models. We architect the CoroGraph system to bridge this trade-off and attain both cache efficiency and work efficiency. CoroGraph adopts a novel hybrid execution model, which combines the benefits of the execution models in existing systems. CoroGraph also extensively uses coroutine-based prefetch to overlap the random memory access of graph algorithms with computation. Experiment results show that CoroGraph yields shorter algorithm execution time than existing systems and provides strong performance across different algorithms and datasets.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. This work has been partially supported by Guangdong Basic and Applied Basic Research Foundation (Grant No.2021A 1515110067), Shenzhen Fundamental Research Program (Grant No. 20220815112848002), the Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and a research gift from Huawei Gauss department. Dr. Bo Tang is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China.

REFERENCES

[1] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery* 29, 3 (2015), 626–688.

[2] J Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large scale networks fingerprinting and visualization using the k-core decomposition. *Advances in neural information processing systems* 18 (2005).

[3] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.

[4] Alex Beutel, Leman Akoglu, and Christos Faloutsos. 2015. Fraud detection through graph-based user behavior modeling. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1696–1697.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.

[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO:Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[7] Yukuo Cen, Jing Zhang, Gaofei Wang, Yujie Qian, Chuizheng Meng, Zonghong Dai, Hongxia Yang, and Jie Tang. 2019. Trust relationship prediction in alibaba E-commerce platform. *IEEE Transactions on Knowledge and Data Engineering* 32, 5 (2019), 1024–1035.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[9] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 17–es.

[10] Intel Coorporation. 2016. *Intel 64 and IA-32 architectures optimization reference manual*.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[12] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 293–304.

[13] David Easley and Jon Kleinberg. 2010. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge university press.

[14] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A system for recommending 3+ billion items to 200+ million users in real-time. In *Proceedings of the 2018 world wide web conference*. 1775–1784.

[15] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. 2004. Modelling disease outbreaks in realistic urban social networks. *Nature* 429, 6988 (2004), 180–184.

[16] 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.

[17] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 599–613.

[18] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making pull-based graph processing performant. *ACM SIGPLAN Notices* 53, 1 (2018), 246–260.

[19] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2021. Coroutine-Oriented Main-Memory Database Engine. *Proceedings of the VLDB Endowment* 14, 3 (2021), 431–444.

[20] Trey Ideker, Owen Ozier, Benno Schwikowski, and Andrew F Siegel. 2002. Discovering regulatory and signalling circuits in molecular interaction networks. *Bioinformatics* 18, suppl_1 (2002), S233–S240.

[21] ISO/IEC. 2017. *Technical Specifcation — C++ Extensions for Coroutines*. https://www.iso.org/standard/73008.html

[22] Hawoong Jeong, Bálint Tombor, Réka Albert, Zoltan N Oltvai, and A-L Barabási. 2000. The large-scale organization of metabolic networks. *Nature* 407, 6804 (2000), 651–654.

[23] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[24] Juno Kim and Steven Swanson. 2022. Blaze: fast graph processing on fast SSDs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[25] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment* 9, 4 (2015), 252–263.

[26] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009), 30–37.

[27] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[29] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi:Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

[30] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2019. GPOP: A cache and memory-efficient framework for graph processing over partitions. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 393–394.

[31] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2020. Gpop: A scalable cache-and memory-efficient framework for graph processing over parts. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–24.

[32] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2018. Accelerating PageRank using Partition-Centric Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 427–440.

[33] Shengliang Lu, Shixuan Sun, Johns Paul, Yuchen Li, and Bingsheng He. 2021. Cache-efficient fork-processing patterns on large graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 1208–1221.

[34] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[35] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[36] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[37] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.

[38] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *Proceedings of the 2021 International Conference on Management of Data*. 1331–1344.

[39] Vikram Narayanan, David Detweiler, Tianjiao Huang, and Anton Burtsev. 2023. DRAMHiT: A Hash Table Architected for the Speed of DRAM. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 817–834.

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 456–471.

[41] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment* 11, CONF (2017), 230–242.

[42] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[43] Stephen B Seidman. 1983. Network structure and minimum degree. *Social networks* 5, 3 (1983), 269–287.

[44] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 505–516.

[45] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. Corescope: Graph mining using k-core analysis—patterns, anomalies and algorithms. In *2016 IEEE 16th international conference on data mining (ICDM)*. IEEE, 469–478.

[46] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[47] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: an in-memory graph random walk engine. (2021).

[48] Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.

[49] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[50] 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 16)*. 301–316.

[51] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph:Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.