Cache-Efficient Fork-Processing Patterns on Large Graphs

Shengliang Lu
National University of Singapore
Singapore

Shixuan Sun
National University of Singapore
Singapore

Johns Paul
National University of Singapore
Singapore

Yuchen Li
Singapore Management University
Singapore

Bingsheng He
National University of Singapore
Singapore

ABSTRACT

As large graph processing emerges, we observe a costly *fork-processing pattern* (FPP) that is common in many graph algorithms. The unique feature of the FPP is that it launches many *independent queries* from different source vertices on the same graph. For example, an algorithm in analyzing the network community profile can execute Personalized PageRanks that start from tens of thousands of source vertices at the same time. We study the efficiency of handling FPPs in state-of-the-art graph processing systems on multi-core architectures, including Ligra, Gemini, and GraphIt. We find that those systems suffer from severe cache miss penalty because of the irregular and uncoordinated memory accesses in processing FPPs.

In this paper, we propose *ForkGraph*, a cache-efficient FPP processing system on multi-core architectures. In order to improve the cache reuse, we divide the graph into partitions each sized of LLC (last-level cache) capacity, and the queries in an FPP are buffered and executed on the partition basis. We further develop efficient intra- and inter-partition execution strategies for efficiency. For intra-partition processing, since the graph partition fits into LLC, we propose to execute each graph query with efficient sequential algorithms (in contrast with parallel algorithms in existing parallel graph processing systems) and present an atomic-free query processing method by consolidating contending operations to cache-resident graph partition. For inter-partition processing, we propose two designs, yielding and priority-based scheduling, to reduce redundant work in processing. Besides, we theoretically prove that ForkGraph performs the same amount of work, to within a constant factor, as the fastest known sequential algorithms in FPP queries processing, which is work efficient. Our evaluations on real-world graphs show that ForkGraph significantly outperforms state-of-the-art graph processing systems (including Ligra, Gemini, and GraphIt) with two orders of magnitude speedups.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**; • **Computing methodologies → Parallel algorithms**; • **Information systems → Parallel and distributed DBMSs**.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457253

KEYWORDS

Graph Processing Systems; Fork-Processing Pattern; Concurrent Query Execution; Buffered Execution Model

ACM Reference Format:
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 (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457253

1 INTRODUCTION

Graphs are *de facto* data structures in various applications such as social network analysis, bioinformatics, online transaction analysis, and weblink analysis. We observe a costly *fork-processing pattern* (FPP) that is common in many graph processing algorithms, as defined in Algorithm 1. The unique feature of an FPP is that it launches many *independent queries* from different source vertices on the same graph (we call those queries *FPP queries*). Below are several representative examples of FPP-based graph algorithms.

Algorithm 1 Fork-processing pattern (FPP) on graph.

```
1: Generate vertex set S
2: parallel_for_each vertex v ∈ S do
3:     Launch a graph query from v
```

(1) *Betweenness centrality* (BC) is widely used to calculate the relative importance of vertices in a graph [26]. On an unweighted graph, BC is solved by first invoking many independent BFSs (breadth-first searches), each from a random vertex. Next, the algorithm gathers the results of each BFS to obtain the centrality of vertices [8]. Although various algorithm variants have been proposed, they have common FPPs of launching massive BFS queries [18, 52].

(2) *Network community profile* (NCP) is defined as the function of the (approximate) best conductance for clusters of a given size in the graph versus the cluster size [33]. An efficient method computing NCP is based on local clustering algorithms, which start a number of PPRs (personalized page ranks) from randomly selected vertices to calculate NCP approximately [17, 47, 51, 56]. The number of PPRs can be at the scale of tens of thousands in the previous study [47].

(3) *Landmark labeling* (LL) pre-computes the shortest paths between selected landmark vertices to accelerate the path queries. Researchers proposed to compute the labels by executing a batch of SSSPs (single-source shortest paths) or BFSs simultaneously [1]. The number of queries in a batch can range from 16 to 1,024 in the previous studies [1].

Table 1: Profiling performance analysis of processing 10,000 PPRs on LiveJournal graph using existing GPSs.

System	Ligra			Gemini			GraphIt		
#Threads in total	1	10	10	1	10	10	1	10	10
Execution Scheme	single-threaded	$t = 10$	$t = 1$	single-threaded	$t = 10$	$t = 1$	single-threaded	$t = 10$	$t = 1$
Instructions ($\times 10^{14}$)	4.57	4.59	4.56	2.07	2.20	2.46	1.30	1.55	1.31
LLC loads ($\times 10^{12}$)	9.10	9.00	9.21	1.30	1.46	1.37	1.59	1.63	1.60
LLC miss ratio	50.0%	48.1%	79.0%	40.1%	31.6%	76.4%	50.1%	38.9%	85.6%
Runtime (hour)	46.74	7.65	6.75	11.66	2.56	1.64	8.39	2.09	1.59

In practice, the processing time of the FPP is the major bottleneck of those graph algorithms, which takes an overwhelming majority of the execution time (> 90%) in our experiments. In this paper, we study whether and how we can improve the performance of handling FPPs on large graphs.

As large graph processing emerges recently, substantial efforts have been made in developing parallel graph processing systems (GPSs) [40, 45, 62, 66]. Those GPSs mainly focus on improving the performance of a single query by taking advantage of the intra-query parallelism. Since an FPP has many independent queries, we study how existing GPSs can take advantage of inter-query parallelism. To this end, we use t to denote the number of threads assigned to a query, and evaluate different t values for balancing the intra- and inter-query parallelisms.

We evaluate three state-of-the-art GPSs (Ligra [45], Gemini [66], and GraphIt [62]) on a 10-core machine (with hyperthreading disabled). Table 1 presents the performance analysis of handling 10,000 PPRs for NCP on a real graph. The detailed experimental setup can be found in Section 6. For varying different t values, we fix the total number of threads to be ten (one thread per core). Specifically, when $t = 1$, GPSs fully take advantage of inter-query parallelism. When $t = 10$, GPSs process queries one by one, and each query runs in parallel, taking advantage of intra-query parallelism only. We find that the configuration of $t = 1$ achieves the best performance among different t values. In the table, we also show the profiling of executing GPSs using a single thread.

We make an important observation across different GPSs. Despite that the executions with configurations of $t = 1$ achieve the best performance by taking advantage of inter-query parallelism, they suffer from a high LLC (last level cache) miss ratio. It can be up to 85.6%, a huge rise from both the single-threaded execution and the approach of intra-query parallelism ($t = 10$). When $t = 1$, threads are handling FPP queries independently and simultaneously, and they are filling up the CPU cache with different parts of the graph. Such uncoordinated memory accesses among FPP queries cause severe LLC cache misses. We present more details in Section 2.

To improve the efficiency of handling FPPs, we develop *ForkGraph*, a cache-efficient system for processing FPPs for in-memory graphs on multi-core machines. The core design of ForkGraph is based on a novel *buffered execution model* on graphs to leverage the locality and sharing opportunities among FPP queries by coordinating their operations to the graph. Specifically, we divide the graph data into LLC-sized partitions and associate each partition with a buffer to store the queries' operations to the partition. We dynamically schedule a partition to process, and the buffered operations are performed in a batch on the cache-resident graph partition.

We further develop efficient intra- and inter-partition processing strategies for efficiency. For intra-partition processing, since the graph partition fits into LLC, we propose to execute each buffered operation with efficient sequential algorithms (unlike parallel algorithms in existing graph systems that have more work due to parallelism) and develop atomic-free mechanisms by consolidating contending operations in the cache-resident graph partition. For inter-partition processing, we observe that a wrong execution order of graph partitions causes a significant amount of redundant work, as well as the benefits of each buffered operation in the same graph partition vary significantly in many graph applications. Thus, we propose two designs accordingly. The first is priority-based scheduling to select the partition that could lead to convergence quickly. The second is yielding optimization that early terminates a query's intra-partition processing to reduce redundant work.

Besides, with the designs in intra- and inter-partition processing, ForkGraph performs the same amount of work, to within a constant factor, as the fastest known sequential algorithms in FPP queries processing, which is work efficient.

We perform a comprehensive analysis of ForkGraph's performance in comparison with three state-of-the-art GPSs (Ligra, Gemini, and GraphIt). The workload includes three applications BC, NCP, and LL on eight real-world graphs. Our experiments show that ForkGraph reduces the total number of LLC misses by up to a factor of 100× compared to other GPSs, and consistently outperforms GPSs, showing 32×, 307×, and 38× speedups over Ligra, Gemini, and GraphIt on average, respectively.

Supplement results are presented in the complete version [36]. Our source code is publicly available at (*https://github.com/Xtra-Computing/ForkGraph*).

The remainder of this paper is organized as follows. In Section 2, we introduce the preliminary and motivation. Section 3 presents an overview of ForkGraph. We present the design of intra- and inter-partition processing in Section 4 and Section 5, respectively. We present the experimental results in Section 6. Finally, we review the related work in Section 7 and conclude in Section 8.

2 PRELIMINARIES AND MOTIVATIONS

2.1 Preliminaries

A graph $G = (V, E)$ is defined to be a directed graph, where V is a set of vertices and E is a set of edges. An undirected graph can be represented as a directed graph by replacing each undirected edge with two edges from both directions. Given $G = (V, E)$, graph partition is defined as follows.

Definition 2.1 (Graph Partition) A partition plan of graph G is a division of V into $|\boldsymbol{P}|$ disjoint vertex sets. A partition P_i of the graph G is a subgraph of G on the i-th vertex set with $1 \leq i \leq |\boldsymbol{P}|$. V_{P_i} and E_{P_i} denote the vertex set and the edge set of P_i, respectively. Specifically, $E_{P_i} = \{e(u, v) \in E | u \in V_{P_i}\}$.

FPP-based graph applications. We have observed the existence of FPPs in many graph applications beyond those presented in the Introduction, including ant colony optimization [16], single-source k-shortest path [57], and graph learning using random walks [21, 42].

Definition 2.2 (FPP Queries) Given a graph $G = (V, E)$, the FPP queries denoted by $Q = \{q_1, q_2, ..., q_{|Q|}\}$ are a set of graph queries that are homogeneous graph queries (e.g., PPRs) *simultaneously* launched from source vertices $src_1, src_2, ..., src_{|Q|} \in V$ on the same graph G, where $|Q|$ denotes the number of queries in the FPP.

$|Q|$ varies in different applications. In the previous studies, it ranges from tens to thousands. For example, Shun et al. [47] run 10^5 PPRs from random vertices to compute NCP. The LL calculation in Akiba et al. [1] performs batches of SSSPs/BFSs simultaneously, and the number of queries in a batch varies from 16 to 1,024. Some graph workloads in the previous studies do not belong to FPP, such as single-query applications like a single BFS [5], multiple dependent queries like multi-commodity flow [3], and heterogeneous graph queries [58]. As FPP is emerging in many graph processing, mining, and learning applications, we focus on FPP queries.

Definition 2.3 (An FPP Query's Operations to Graph) An FPP query can have many operations that are spreading randomly in the graph. We define an operation of an FPP query as a triple in the format of $\langle q_i, v, val \rangle$ to represent an operation belonging to q_i at vertex v with value(s) val.

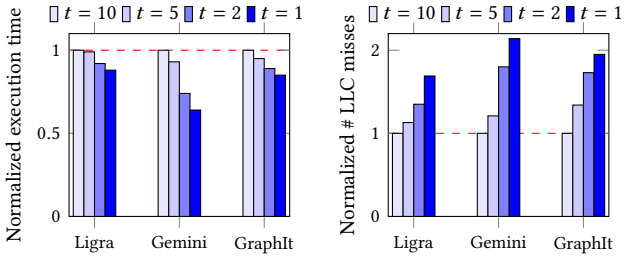
For example, in an SSSP query q_1, each vertex v is assigned with a property that represents the length of the shortest path found from source vertex src_1 to v; an operation to vertex v contains the length l of a path from src to v, denoted as $\langle q_1, v, l \rangle$. If l is shorter than the existing path, we update the vertex's property using the operation and generate new operations to the vertex's neighbors.

2.2 Motivation

We present more detailed results of Table 1 to explain our motivation. *First, the large number of LLC misses is the performance bottleneck.* When evaluating 10,000 PPRs using the three GPSs, we observed that the number of stalled memory cycles is 34 − 40% of the total time spent in memory units when leveraging intra-query parallelism ($t = 10$). The percentage increases up to 55% when leveraging inter-query parallelism ($t = 1$). The memory stalls are mainly caused by the LLC misses, which bottleneck the performance. Thus, we focus on LLC among the multi-layer caches in this work.

Second, leveraging the inter-query parallelism in existing GPSs brings uncoordinated memory accesses and thus more LLC misses, which limit the potential benefits of the inter-query parallelism. Figure 1 shows the evaluation of the GPSs' performance and the number of LLC misses by varying the t value on a 10-core machine. Figure 1a shows that leveraging inter-query parallelism by setting $t = 1$ is better than other settings, mainly because of the reduced synchronization and locking overhead among threads, as well as better load balancing among threads. However, as shown in Figure 1b, the total number of LLC misses significantly increases (up to 2.1×) as t changes from ten to one.

As threads under the inter-query parallelism are filling up the CPU cache with different parts of the graph, the uncoordinated memory accesses among FPP queries cause high LLC cache misses



(a) Normalized execution time. **(b) Normalized #LLC misses.**

Figure 1: GPSs' performance affected by cache contention with different numbers of threads assigned to each query, tested with 10,000 PPRs on LiveJournal graph.

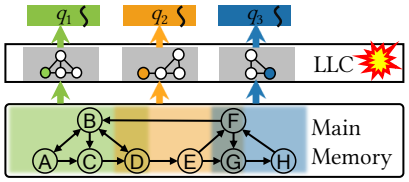


Figure 2: A GPS leveraging the inter-query parallelism.

and limit the potential benefits of the inter-query parallelism. We use Figure 2 to illustrate the scenario of uncoordinated operations of FPP queries to the graph. In this example, a GPS is processing three FPP queries simultaneously, each with a thread sharing the LLC. Since threads are handling FPP queries independently, they contend with each other to the limited cache for storing different parts of the graph. This causes severe cache thrashing and downgrades the performance of handling FPPs.

We also observe that, among these three GPS, GraphIt is the only GPS with cache-optimized techniques to break the graph into LLC-sized segments to limit random accesses within the cache [60, 62] and thus performs better than other GPSs when leveraging the intra-query parallelism. However, GraphIt is the most vulnerable GPS when leveraging the inter-query parallelism with the cache optimization. Particularly, GraphIt ($t = 1$) spends 1.59 hours on 10,000 PPRs. Thus, the total CPU time on 10 cores is 15.9 hours, 190% over the CPU time of the single-threaded counterpart (8.39 hours). Similarly, the CPU time of Ligra ($t = 1$) is only 144% over the CPU time of the single-threaded counterpart as it does not optimize the cache for intra-query parallelism.

Although inter-query parallelism exhibits poor cache efficiency, it still outperforms the default intra-query parallelism in GPSs for most of the cases. The reasons are as follows. First, the inter-query parallelism inherently eliminates cost synchronizations among threads. Second, the inter-query parallelism can benefit from efficient execution by employing state-of-the-art sequential algorithms. Last but not least, as Beamer et al. [6] show that many parallel implementations do not fully utilize the memory bandwidth, the inter-query parallelism mitigates data dependences and increases memory-level parallelism during processing. It thus shows the potential for significant performance improvement on GPSs with current memory systems.

In summary, the above-mentioned insights motivate us to propose a system to address the cache inefficiency in leveraging inter-query parallelism so that we can efficiently support FPP queries.

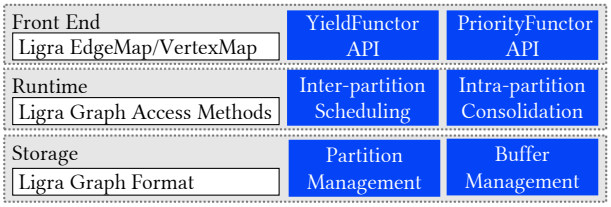


Figure 3: System overview of ForkGraph.

3 SYSTEM OVERVIEW

To improve the cache efficiency of processing FPPs, we propose a cache-efficient system, namely ForkGraph. The core design of ForkGraph is based on a novel buffered execution model. We divide graph G into LLC-sized partitions and associate each partition with a buffer that stores the FPP queries' operations to the partition. The buffered execution model leverages the temporal localities among FPP queries by batching operations from different queries and executes them in a batch for each partition. Since each graph partition can fit into LLC, random memory accesses of the operations in the batch are naturally limited in the LLC with a low cache miss rate.

We develop efficient intra- and inter-partition processing strategies for efficiency. For intra-partition processing, the work efficiency becomes essential since most operations are processed in cache-resident graph partitions. Therefore, we propose to apply sequential implementations to execute multiple operations simultaneously. We leverage inter-query parallelism by assigning a single thread to handle each buffered operation. We adopt sequential implementations because they are usually more work-efficient than parallel algorithms. Besides, ForkGraph consolidates the operations in the buffer that belong to the same query, and thus the operations belonging to the same query can be processed in an atomic-free manner. Moreover, this query-centric operation consolidation significantly reduces redundant operations.

For inter-partition processing, we observe that a wrong execution order of graph partitions causes a significant amount of redundant work, as well as the benefits of each buffered operation in the same graph partition vary significantly in many graph applications. We show these two observations in the experiments. Thus, we have two tasks: 1) to determine when to terminate intra-partition processing and switch to the next partition, and 2) to decide which partition to process next. We propose a yielding optimization to early terminate a query in intra-partition processing to reduce redundant work within a partition. Furthermore, ForkGraph applies a priority-based scheduling to select the partition that leads to convergence quickly. We will detail the design of inter-partition scheduling shortly in Section 5.

Currently, ForkGraph supports a list of queries commonly used in FPPs, including BFSs [46], DFSs (Depth-first searches) [49], SSSPs [15], PPRs [47], and RWs (random walks) [2]. On top of those queries, one can implement FPPs for applications like NCP, BC, and LL. Based on ForkGraph, users can also easily implement more fundamental query types to support other FPP-based graph applications.

3.1 System Architecture

Figure 3 shows the architecture of ForkGraph. ForkGraph extends the Ligra framework by including its APIs (application programming interfaces), graph access methods, and the graph storage.

Algorithm 2 ForkGraph: FPP Processing on graph partitions $\boldsymbol{P}$.

```
1:  InitBuffers(P, Q)
2:  while At least one buffer has operations do
3:    P_c ← ScheduleNextPart()                    ▷ see Sec. 5
4:    IntraPartProcess(P_c)                       ▷ see Sec. 4
5:  procedure ScheduleNextPart()
6:    nextP ← get the next partition according to PriorityFunctor()
7:    return nextP
8:  procedure IntraPartProcess(P_c)
9:    Consolidate operations in P_c.buffer according to each query
10:   parallel_for_each q ∈ Q do
11:     for op ∈ P_c.buffer.getOps(q) do
12:       newOps ← Compute(op, P_c)
13:       Use newOps to update P_c.buffer
14:       if CanYield(P_c, q, YieldFunctor()) then
15:         Yield() and goto Line 10              ▷ terminating q, see Sec. 5
16:   Send operations to neighbor partitions
```

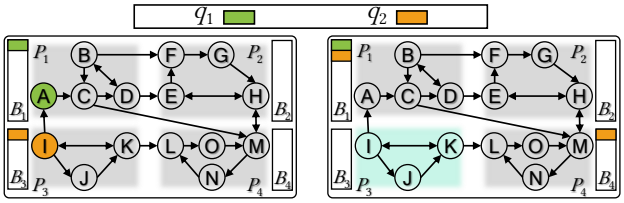
We choose Ligra because of its high performance and wide adoptions by lines of excellent works, including [13, 14, 44, 47]. Another reason is that users can leverage the friendly programming interfaces in Ligra. It provides two very simple APIs vertexMap and edgeMap, used for user-defined functions over vertices and edges, respectively, making programs in Ligra very simple and concise.

On top of Ligra, we expose two user-defined APIs including priority functor and yield functor to users for customizing the logic for inter-partition scheduling. We also add the inter-partition scheduling and intra-partition consolidation on top of Ligra's runtime. We reuse the efficient graph storage and the access methods to edges/vertices in Ligra, which adopts the popular CSR (Compressed Sparse Row) format to store a graph. We add graph partition and buffer management based on the efficient storage of Ligra.

3.2 Overall Execution Flow

Algorithm 2 shows the overall execution flow of ForkGraph on handling an FPP. We assume that the graph is already partitioned, and the set of graph partitions are represented as $\boldsymbol{P}$. At Line 1, ForkGraph initializes a buffer, which is a dynamic-sized contiguous memory space, for each partition. Also, we assign the FPP queries in Q to the corresponding buffers in the initialization.

As long as there exists a non-empty buffer (Line 2), ForkGraph invokes ScheduleNextPart to find the next partition P_c to process. The scheduling is priority-based, targeting at convergence quickly. Next, ForkGraph processes the buffered operations in the scheduled partition P_c by calling IntraPartProcess at Line 4. In IntraPartProcess, ForkGraph consolidates the operations in the buffer and assigns operations of the same query to a single thread so that the Compute procedure is in an atomic-free manner. Thus, we put a "parallel for" execution for different queries at Line 10. The processing of queries' operations can generate many new operations targeting at P_c and P_c's neighbor partitions. We have two cases for each of the new operations generated: 1) if it targets $\boldsymbol{P}_c$, we put it to the $\boldsymbol{P}_c$'s buffer; 2) if it targets other graph partitions, we do not send them to the buffers of their target partitions immediately. For each of the $(|\boldsymbol{P}| - 1)$ partitions, we maintain a local buffer and store the operation into the local buffer. We send them in batches after finishing processing $\boldsymbol{P}_c$ (at Line 16).



(a) Initialization. (b) Processed operations in P_3.

Figure 4: ForkGraph processes two SSSP queries q_1 and q_2 start from vertices A and I, respectively, as highlighted.

At Line 14, ForkGraph monitors the processing of query q and adopts the yielding optimization to terminate the processing of q earlier for work efficiency. The yielding happens within the intra-partition processing, but controls the amount of the work spent in the current partition to trigger the scheduling of the next partition to process. This is similar to the concept of *yield* in process scheduling in operating systems.

An example in buffered execution. We use an example in Figure 4 to illustrate the buffered execution. There are 15 vertices divided into four partitions, and we set the weight of all edges to be one for simplicity purposes. We consider an SSSP-based FPP such as LL, and use two SSSPs q_1 and q_2, from source vertices A and I in partitions P_1 and P_3, respectively. ForkGraph initializes these queries as operations in the corresponding buffers, as shown in Figure 4a. Suppose P_3 is the first partition scheduled to process. Figure 4b shows the state after processing P_3. ForkGraph sends operations to neighbor partitions, P_1 and P_4, as shown in their buffers B_1 and B_4. This process repeats until all buffers are empty.

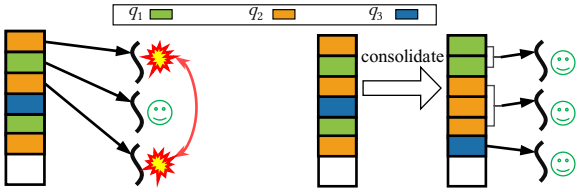
4 INTRA-PARTITION PROCESSING

By applying the buffered execution model, ForkGraph explores the opportunities of optimizing cache-efficient intra-partition processing of FPPs by batching the operations to LLC-sized graph partitions. In this in-cache processing, work efficiency becomes essential, with the following two main issues. First, we find that the parallel execution model adopted by the current GPSs can be extremely inefficient for in-cache processing many operations. We give more details shortly in Section 4.1. Second, processing many operations at the same time could cause severe synchronization overheads and conflicts if they are from the same query. For example, one operation may read the property of a vertex and the other operation from the same query may update the property of the same vertex, which causes read-write conflicts. We develop an atomic-free approach to eliminate the conflicts in Section 4.2.

4.1 Sequential vs. Parallel Execution

The current GPSs [40, 45, 62, 66] use parallel algorithms to execute one query in order to take advantage of intra-query parallelism, which, however, is not free. Firstly, it comes with the overhead in parallelization, such as thread synchronization, locking, and scheduling. Second, most of the parallel algorithms perform significantly more work than their sequential counterparts. Those overheads can become relatively more significant given the in-cache processing.

Since there are usually more operations than the number of available CPU threads in handling FPPs, we propose to leverage the inter-query parallelism. Particularly, we choose the sequential algorithm to execute each operation and execute multiple operations simultaneously, i.e., each thread fetches one operation at a time from the buffer and processes it using the corresponding sequential algorithm. Specifically, ForkGraph reuses sequential algorithms from existing works. The SSSP and BFS algorithms are obtained from the problem based benchmark suite (PBBS) [46], and we reuse the sequential PPR implementation from [47].



(a) Without consolidation. (b) With consolidation.

Figure 5: Comparison of the execution on buffered operations with and without consolidation.

For a partition, if the buffer only has one operation and there are no other on-going operations, we can switch to parallel algorithms to process operation. However, we hardly observe this scenario happening and thus use sequential algorithms in most cases.

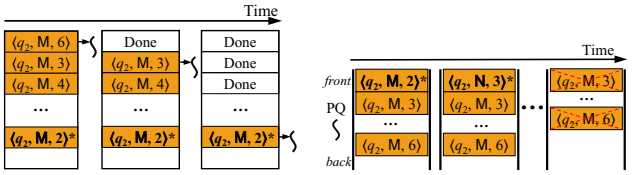
4.2 Query-centric Operation Consolidation

As multiple operations are processed simultaneously, different threads can process operations of the same query at the same time, which may cause access conflicts. Therefore, it requires locking and atomic operations when threads simultaneously read and write the query-specific data, e.g., the intermediate results and vertices' properties. This is also commonly seen in GPSs [40, 45, 62, 66] that parallelize the processing of a single query.

In this work, we propose an atomic-free approach that efficiently eliminates the conflicts among processing different operations from the same query. Particularly, ForkGraph *consolidates* the operations based on the queries they belong to and uses a single thread to handle a set of consolidated operations from the same query. Multiple threads execute different FPP queries simultaneously, as shown in Algorithm 2 Line 10. The consolidation has the following benefits. First, operations of the same query can be processed in an atomic-free manner since there is only one thread updating the query-specific data. Second, as the thread only processes data of the same query and the query-specific data is stored in a contiguous memory space, it avoids stride memory accesses to data of other queries.

Figure 5 illustrates an example of the differences between processing with and without consolidation. We assume there are three threads. Without consolidation, threads fetch operations in the buffer and process them simultaneously. As shown in Figure 5a, when two threads process the operations from the same query q_2, we need to apply atomic operations to resolve the potential conflicts (read-write conflicts). Figure 5b shows that we perform the operation processing in an atomic-free manner by consolidating buffered operations and assigning those operations belonging to the same query to one thread.

Given the user-defined priority functor, we can further reduce the redundant work based on query-centric consolidation. The priority



(a) Without priority functor. (b) With the priority functor.

Figure 6: Comparison of the redundancy in processing operations with and without using the priority functor in SSSP. We highlight the operations with the optimal value using ∗.

functor relies on the logics of the sequential algorithms provided. During the consolidation, we maintain an order of operations from the same query according to the priority functor. In the execution, we always choose the one with the highest priority, which has more pruning power on redundant work. Particularly, commonly adopted SSSP implementations, e.g., Dijkstra's algorithm, rely on a priority queue (PQ) that assigns higher priorities to shorter paths. The algorithm uses shorter paths conveyed in the operations to prune non-optimal ones. The sequential PPR implementation from [47] relies on a multiset structure that allows inserting all operations of the same query to the same queue for sequential processing in the decreasing order of the residual values. When solving a BFS, the priority value is the lowest level from the source in BFSs.

Figure 6 gives an example of processing consolidated operations of an SSSP query, where we can leverage the user-defined priority functor to reduce the number of redundant operations. Without prioritizing, a thread is fetching the operations to process one by one. The operation with the most significant value can be in any place of the buffer. For example, $\langle q_2, \mathsf{M}, 2\rangle$ contains the optimal value and is queued after many other operations in Figure 6a. The operations before $\langle q_2, \mathsf{M}, 2\rangle$ are all redundant. In contrast, as shown in Figure 6b, by leveraging the priority functor of Dijkstra's algorithm for solving SSSP [15], ForkGraph processes the most beneficial operation of the same query. Here, the processing of $\langle q_2, \mathsf{M}, 2\rangle$ in the first place can effectively prune the redundant ones.

5 INTER-PARTITION SCHEDULING

With efficient intra-partition processing, we still need to decide the order of scheduling partitions to execute. We observe that a wrong execution order of partitions takes more steps to converge and causes momentous redundant work. In this section, we focus on the following two issues of inter-partition scheduling that significantly affect the work efficiency of handling the entire FPP.

First, when should ForkGraph terminate intra-partition processing and switch to the next partition? One basic approach is to finish all the operations within the current partition before switching to the next partition. However, this would cause a significant amount of redundant work. According to the usage of the priority functor in query-centric consolidation, the later operations of the same query tend to have diminishing contributions to the convergence of many graph problems [40, 45]. They can even be pruned by the future operations generated by other partitions. Correspondingly, we present a yielding optimization to decide the early termination of a query in intra-partition processing (Section 5.1).

Second, which partition should ForkGraph process the next? We propose a priority-based scheduling to select the partition that could lead to quick convergence as the next to process (Section 5.2).

5.1 Heuristic-Based Yielding

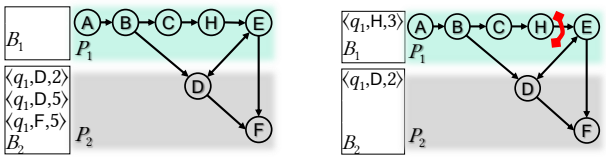
The yielding optimization partially processes a partition to avoid redundant work, i.e., early termination for intra-partition processing. Determining the optimal strategy for yielding on early termination is impractical due to the complexity of graph structures and the convergence of graph applications. For example, one question here is: how to determine the utility (or benefits) of executing an operation to the convergence of the entire application. Existing studies [23, 47, 59] rely on heuristics, such as priority, to approximate the utility. Our problem is more challenging and more complex than this question. Therefore, like the previous studies, we empirically use two heuristics in ForkGraph to decide if a thread should yield the processing of a query. The threshold values of these heuristics can be tuned and adjusted by users. We experimentally evaluate their impacts in the evaluation.

Yielding heuristic 1: on the number of edges processed. The first heuristic is to examine the number of edges processed since the start of processing in the partition and yield if the number is beyond a threshold. For example, the processing of a PPR gradually converges into local stable states with more edges processed but could be easily fluctuated by operations sent from other partitions in later steps. Similarly, an SSSP process goes deeply in a subgraph with more paths enumerated via processing edges, where the chance of generating non-optimal paths is higher since there are more routes from neighbor partitions that could lead to shorter ones. Such operations tend to be redundant. ForkGraph reduces such operations by limiting the number of edges processed.

Yielding heuristic 2: on the operations' values updated. The second heuristic is to check if the values updated so far in the partition exceeds a range, inspired by the Δ-stepping and similar approaches [7, 39]. The heuristic works in this way: for each query in the partition, we record the value α conveyed from the first operation we execute. Based on the intra-partition processing, this operation has the most benefits to the convergence. If the currently processed operation's value is much worse (greater or smaller) than α value by a factor or a threshold, we should yield the processing.

For example, when solving SSSPs, if the shortest path to process in the current partition is greater than $dist_{min} + \Delta$, where $dist_{min}$ is the distance from the source vertex to the closest unprocessed vertices, there is a high chance that there would be better paths [39, 59] that are not yet exploited. Δ is a tunable parameter of the Δ-stepping algorithm, and the algorithm restricts the processing to vertices whose distances from the source are within $[dist_{min}, dist_{min} + \Delta)$. Similarly, in solving PPRs, if the highest residual of vertices in the partition is not significant enough, instead of continuing processing for local stable states, a better solution is to yield and wait for more influencing operations propagated to this partition.

The yielding heuristics only pause the processing of the current query in the partition. After a query yields, the unprocessed operations (including newly generated ones) for that query are kept in the partition's buffer and processed later. The processing will



(a) Finish q_1 in P_1 w/o yielding. (b) Yield q_1 at edge H to E.

Figure 7: Comparison of the execution and number of operations with and without yielding in P_1. Shorest path query q_1 starts at vertex A in P_1. All edges are with unit lengths.

eventually converge as the standard algorithms and the correctness of processing results are guaranteed.

Figure 7 shows the comparison of the execution and the number of operations during processing with and without yielding. The figure reflects the states of finishing processing P_1 and switching to P_2. In Figure 7a, ForkGraph switches to P_2 when it finishes all operations in P_1 and sends three operations to P_2. It then schedules to process P_2. We can expect that ForkGraph has to revisit and update vertex E in P_1 again because the shortest path from A to E will be found via vertex D.

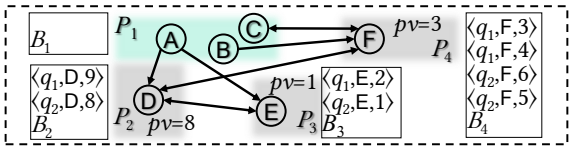
In Figure 7b, ForkGraph yields the process after vertex H, stores the operation at H in P_1, and sends one operation to P_2. When ForkGraph finishes the execution in P_2, it sends the shortest path update in an operation $\langle q_1, \mathsf{E}, 3\rangle$ to E. Compared with the execution without yielding, the redundant operations $\langle q_1, \mathsf{D}, 5\rangle$ and $\langle q_1, \mathsf{F}, 5\rangle$ are pruned. ForkGraph will resume to process the remaining operation $\langle q_1, \mathsf{H}, 3\rangle$ in P_1's buffer together with operations sent from P_2. In this way, we reduce the redundant updates and guarantee the exact results.

To summarize, yielding can significantly reduce the redundant operations within the current partition and the redundant operations propagated to neighbor partitions. Furthermore, we give a theoretical proof to show that the yielding helps the work efficiency of FPP queries processing in the complete version [36].

5.2 Priority-Based Scheduling

When ForkGraph finishes the processing of a partition, inter-partition scheduling selects another partition with a non-empty buffer to process. A wrong execution order of graph partitions leads to the repeated revisiting of partitions. To avoid such inefficiency, we propose a priority-based scheduling that aims to pick a partition that can lead to quick convergence of the FPP processing to process. We assign each partition with a priority value based on the priority functor. Intuitively, some partitions are buffering operations that would quickly lead to the convergence of queries, e.g., the shortest path or the most effective value changes in PPR updates. Therefore, we prefer to process these partitions' buffered operations than others for quick convergence.

The key question is how to determine the priority value of each partition. The priority of a partition is defined to be the highest priority value among all the operations in the partition and the priority values are generated in the priority functor. Like many existing studies, the priority functor in ForkGraph is defined on per operation (i.e., per vertex), rather than on a set of vertices. For example, Dijkstra's algorithm for SSSP takes shorter distances as higher priorities [15], and it always uses the shortest path to



Scheduling	Execution order	#Operations processed
Random	$P_1,P_2,P_4,P_2,P_3,P_2,P_4$	11
Max #operations	P_1,P_4,P_2,P_3,P_2,P_4	9
FIFO	P_1,P_3,P_4,P_2,P_4	7
Priority-based	P_1,P_3,P_2,P_4	6

Figure 8: The execution orders under different scheduling methods. Shortest path queries q_1 and q_2 start in P_1. Only vertices with edges crossing partitions are shown for brevity. All edges are with unit lengths.

update other vertices. As there can be many buffered operations in a partition, selecting the value with the highest priority is a simple and effective approach to determine each partition's priority.

With a given priority functor, ForkGraph always schedules the partition with the highest priority in the graph to process next. However, the scheduled partition might be the most desired partition for only a subset of FPP queries, but not for all. Redundant operations may be generated when executing operations of queries that desire other partitions. To deal with the redundancy, we control the amount of work spent by each query using the proposed yielding optimization. We theoretically prove that ForkGraph is work-efficient on handling FPP queries and one of the reasons is that the yielding optimization effectively reduces such redundancy. Due to the space limit, we put the proof in the appendix of the complete version of this paper [36].

Figure 8 shows the comparison of the execution orders using different scheduling methods. *pv* denotes the priority value of each partition at this stage. The random scheduling picks an arbitrary partition with operations to process at each step, which could take more steps than other methods, resulting in slow convergence. The heuristic of picking up the partition with the most number of operations (denoted as "Max #operations" in the figure) could maximize the reuse of cache content. However, our experiments in Section 6.4 show that it is slower than other methods because of involving more redundant work. The FIFO-based scheduling visits partitions based on the orders of operation generated. Compared with these methods, leveraging the priority functors from Dijkstra's algorithm, ForkGraph can schedule the execution orders to maximize the exploitation of shortest paths and reduce redundant work. In this example, the priority-based scheduling method shows the smallest number of visits to partitions and the least number of operations processed. We provide the detailed work-through of this example in the appendix of the complete version [36].

The priority functors are programmed by users. Sometimes, it can be non-trivial to develop a functor for a certain graph operation. Fortunately, in the decades of research on graph processing, priority functors have been developed for many graph algorithms [15, 46, 47], especially in a wide range of existing sequential algorithms. Thus, users can reuse the implementation of those priority functors as what we did in the experiments. By default, ForkGraph uses FIFO queues to schedule partitions.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate ForkGraph on handling FPPs on real-world graphs compared with state-of-the-art GPSs.

6.1 Experimental Setup

Hardware configuration. We conduct experiments on a Linux server with a 10-core Intel® XEON® W-2155 CPU (hyperthreading disabled) and 256GB memory. The frequency of the CPU is 3.3GHz, and the LLC is 13.75MB. We compile all the implementations using g++ 7.5.0 with -O3 flag and OpenMP enabled.

Implementation Details. We develop ForkGraph in C++ with OpenMP, where most of the components added to Ligra are developed by reusing existing GPSs' primitives or using the C++ Standard Template Library (STL). For inter-partition scheduling, we adopt the priority queue container in STL because the scheduling workload is not heavy (there are at most $|\boldsymbol{P}|$ elements maintained, each of which is the priority of a partition, where $|\boldsymbol{P}| << |V|$); as discussed early, we directly adopt the priority functors and yielding functors from the state-of-the-art sequential implementations [15, 39, 47]. For intra-partition processing, ForkGraph fixes the total number of available threads as the number of hardware threads to ensure high CPU utilization, as well as avoid high context switches and process migration overhead.

In the buffer management, we develop a simple and efficient multi-bucket structure to buffer operations, where we allocate K ($K \leq |Q|$) independent buckets for each partition's buffer. We equally divide the $|Q|$ queries into K corresponding groups and assign each group to explicitly use one of the buckets in each buffer, reducing the overhead of consolidating different queries' operation. In our evaluation, we set K to be much larger than the number of cores in the system to allow fine-grained workload allocation. We implement the bucket using a parallel vector structure from GraphIt's code base [62], which dynamically adjusts capacities with a tunable growth factor, to make each bucket dynamic-sized and in contiguous memory, without wasting much memory.

For graph partitioning, we mostly use METIS [29], one of the state-of-the-art edge-cut tools, to pre-process the graph with the objective of minimizing the total number of edges across different partitions. Since METIS shows poor partitioning quality on large-scale social network graphs [53], we randomly partition these graphs into parts with equal number of vertices.

In the evaluation, we use the priority functor in Dijkstra's algorithm [15], and set yielding heuristic 2 as guided in the Δ-stepping algorithm [39] for BC and LL; we use the priority functor in [47], and set yielding heuristic 1 as guided in the appendix of [36] for NCP. Particularly, the priority-based scheduling is implemented as priority queue with binary comparison functors (comparators) for all the three applications evaluated in this paper. In the evaluation of BC and LL, the functors compare two operations and return "true" if the path length carried by the first operation is shorter than that carried by the second. For NCP, the functor returns "true" if the residual carried by the first operation is higher than the one carried by the second. This setup is used for all the tests in the rest of this paper, unless specified otherwise.

System settings. The system settings contain three major perspectives. First, given a machine, for any given graph, the partition size is fixed to be the LLC size. In other words, $|\boldsymbol{P}|$ is calculated as $G.size/LLC.size$. Second, users can provide the priority functors. Typically, users obtain the priority functors from existing sequential algorithms. If there is no functor provided, ForkGraph uses a FIFO queue for processing and scheduling by default. Third, we set the yielding heuristics based on the work efficiency analysis. In this way, ForkGraph shows comparable or the best performance among different settings, as shown later in Table 4. Although these settings may not achieve the best-case performance, they are sufficiently good in practice.

Comparisons. We compare ForkGraph to the other three representative GPSs, Ligra [45], Gemini [66], and GraphIt [62]. Ligra has the fastest implementation of many algorithms, as it is still actively maintained [13, 47]. Gemini is a distributed graph processing system with notable shared-machine performance. Gemini is compiled and tested with message passing functions disabled. We also compare with GraphIt [62], the state-of-the-art DSL (domain-specific language) for high-performance graph analytics. All three systems allow users to explore various optimization and tradeoffs (such as push vs. pull, and dense vs. sparse frontier). In our evaluation, all the systems are carefully tuned and tested with different configurations. Particularly, the tuned configurations include the direction switch threshold in Ligra and Gemini, the scheduling in GraphIt, the yielding and priority-based scheduling in ForkGraph. We present the best result for all systems among those tests.

As mentioned in the Introduction, we use t to denote the number of threads assigned to a query in Ligra, Gemini, and GraphIt. Particularly, when $t = 10$ and FPP queries are executed one by one (with intra-query parallelism on ten cores), we denote these three GPSs as Ligra ($t = 10$), Gemini ($t = 10$), and GraphIt ($t = 10$), respectively. When $t = 1$, we run each query independently using one hardware thread and use OpenMP's dynamic scheduling to allow each GPS to fetch and process FPP queries without synchronizations. We denote those implemenations as Ligra ($t = 1$), Gemini ($t = 1$), and GraphIt ($t = 1$), respectively.

We measure the time for each system to complete the FPP queries processing. This measurement excludes the time spent reading data from the disk because we focus on optimizing the in-memory computation. For each of the FPP-based graph applications evaluated, we generate three testing sets per graph, representing three testing instances of an application on a graph. We run all tests five times and report the average execution time of the $3 \times 5 = 15$ tests. Note that there are a few FPP queries in each test, representing a single instance of an FPP application. Details are given below.

Applications. We evaluate competing systems' performance on three applications, BC, LL, and NCP. To keep consistent with previous work [1, 18, 47], we configure the three applications as follows.

- The original *BC* performs one SSSP from each vertex for a weighted graph (for unweighted graphs, one BFS from each vertex). This is too time-consuming to be practical. Instead, we adopt an approximate approach by Eppstein et al. [52]. The algorithm samples the starting nodes from the graph. Therefore, in our evaluation, we randomly sample a batch of 100 source vertices for each graph [18].
- The testing of *NCP* follows [47]. Each NCP requires running PPRs using a seeding of 0.01% of the vertices that are randomly sampled in the target graph.

Table 2: Input graphs ($\overline{d}$ is the average degree).

Graph	Source	#V	#E	$\overline{d}$	Memory	$\|P\|$
Ca	California [11]	1.9M	4.6M	2.4	0.07GB	5
Us	USA [11]	23.9M	57.7M	2.4	0.82GB	62
Eu	Europe [4]	50.9M	0.1B	2.1	1.65GB	120
Or	Orkut [32]	3.1M	0.1B	38.1	1.37GB	100
Wk	Wikipedia [10]	3.6M	45.0M	12.6	0.54GB	40
Lj	LiveJournal [32]	4.8M	87.5M	18.0	1.04GB	76
Pt	Patents [32]	16.5M	33.0M	2.0	0.50GB	37
Tw	Twitter [30]	61.6M	1.5B	23.8	17.27GB	1256

• The testing of *LL* follows [1]; each LL requires executing 1,024 independent SSSPs from source vertices that are randomly sampled in the target graph.

Data sets. All the data sets are publicly available and widely used in the previous literature [45, 62, 66] to benchmark algorithms and frameworks. The data sets are listed in Table 2. Following the experimental studies in [45] and [12], we create weighted graphs by selecting edge weights between $[1, log|V|)$ uniformly at random. Ca, Us, and Eu are road networks, Or, Lj, and Tw are social networks, Wk is a hyperlink network, and Pt is a citation network.

Experimental outline. We first present the overall comparison between ForkGraph and other GPSs in Section 6.2. In Section 6.3, we present the profiling results on cache performance and work efficiency. In Section 6.4, we evaluate the impacts of individual techniques and system tuning. Due to the space limit, we summarize some other results in Section 6.5 and present the details in the appendix of the complete version [36].

6.2 Overall Performance Comparison

Finding (1): ForkGraph significantly outperforms Ligra, Gemini, and GraphIt in different execution schemes by 32×, 307×, and 38× speedups on average, respectively.

Figure 9 shows the performance of the applications with different implementations. For Ligra, Gemini, and GraphIt, we evaluate different threading configurations (t), and only show the best results for brevity. As the execution time of different test cases varies significantly, we present the normalized execution time to the performance of Ligra ($t = 1$). The normalized performance of ForkGraph is annotated on the plots.

In Figures 9a and 9c, since Ligra ($t = 1$), Gemini ($t = 1$), and GraphIt ($t = 10$) outperform Ligra ($t = 10$), Gemini ($t = 10$), and GraphIt ($t = 1$), respectively, for almost all the cases, we omit the results for brevity. Similarly, we omit the results of Ligra ($t = 10$), Gemini ($t = 10$), and GraphIt ($t = 10$) in Figure 9b.

Overall, ForkGraph significantly outperforms the other three GPSs in different schemes on all the tested applications with two orders of magnitude speedups on average. Besides, we show later that ForkGraph reduces the number of LLC misses by more than a factor of 10×. We make the following observations in comparison with each of state-of-the-art GPSs.

First, ForkGraph shows 51× speedups over Ligra ($t = 10$) and 32× over Ligra ($t = 1$) on average. ForkGraph accelerates the convergence within partitions with low cache thrashing and adopts the sequential implementation to reduce the total workload.

Second, Gemini's implementations suffer from high synchronization overhead in each iteration because it is designed with the

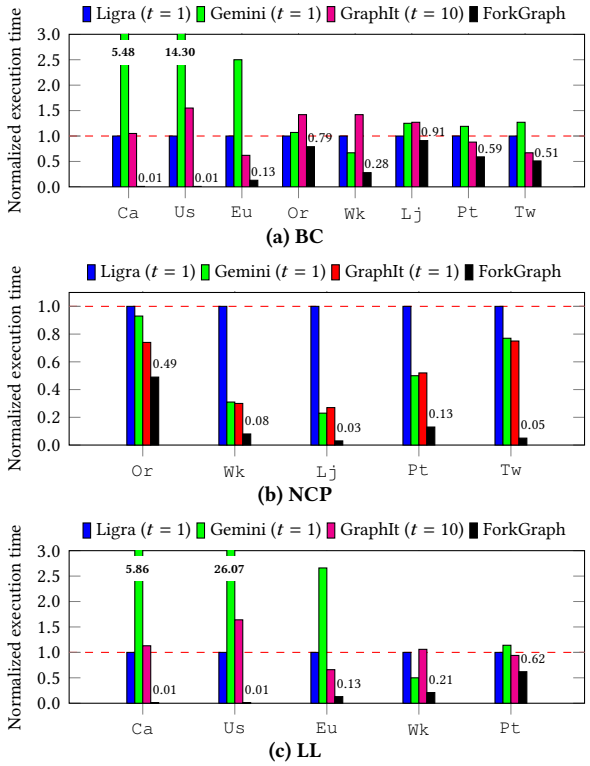


Figure 9: Overall execution time for the three applications with different implementations. We carefully tune all the systems and only report the best performance.

message passing mechanism for a distributed setting. Although all the message-passing functions are disabled in our evaluation, the materialization overhead between consecutive iterations is significant. ForkGraph handles the operations of FPP queries using fast implementations of sequential algorithms, which incurs minimal overhead in synchronization and atomic operations. As a result, ForkGraph delivers three orders of magnitude speedups over Gemini, especially on road graphs with high diameters.

Third, although GraphIt optimizes a single query's cache usage, it shows higher contention with more threads enabled. Except for solving PPR, GraphIt shows a slowdown when $t = 1$. The reason is that the graph-traversal based queries in BC and LL benefit the cache optimization provided in GraphIt, while the high LLC misses due to uncoordinated memory accesses is too high to be covered by the performance gain when leveraging the inter-query parallelism in GraphIt ($t = 1$). Compared with GraphIt, ForkGraph aims to optimize the performance in the inter-query parallelism setting and achieves up to 197× speedups over the best of GraphIt under different schemes. ForkGraph slightly outperforms GraphIt ($t = 10$) on social networks by 1.3× on solving BC because GraphIt ($t = 10$) generates more efficient direction-optimized traversals by searching through a much larger space of optimizations. On data sets that do not rely on direction optimization, ForkGraph achieves more than 100× speedups over both GraphIt ($t = 10$) and GraphIt ($t = 1$).

As an example that details the actual execution time, Table 3A shows the execution times of the four systems on solving NCP. We can observe that it takes Ligra ($t = 10$) 11.5 hours to process the

Table 3: Execution time and memory consumption of NCP on different data sets using the four systems.

A. Execution time (minutes)

	Or	Wk	Lj	Pt	Tw
Ligra ($t = 10$)	0.7	10.1	23.5	28.9	692.0
Ligra ($t = 1$)	0.7	8.1	19.8	17.8	243.3
Gemini ($t = 10$)	0.7	4.2	7.2	13.2	230.8
Gemini ($t = 1$)	0.7	2.5	4.5	8.9	187.4
GraphIt ($t = 10$)	0.6	3.5	7.1	11.2	198.8
GraphIt ($t = 1$)	0.5	2.4	5.4	9.3	181.9
ForkGraph	0.4	0.6	0.7	2.3	11.7

B. Memory consumption (GB)

	Or	Wk	Lj	Pt	Tw
Ligra	7.9	17.2	39.7	70.3	148.0
Gemini	16.1	19.9	35.0	103.2	130.8
GraphIt	17.1	20.0	37.3	103.9	149.8
ForkGraph	12.7	23.8	39.1	98.6	152.1

PPRs on the Tw graph, while ForkGraph only needs 12 minutes. Thus, ForkGraph is more practical for many graph applications. Table 3B shows the memory usages of the four systems. Basically, most of the memory is spent on storing the execution results of FPP queries, and ForkGraph consumes 5 – 19% more memory than other GPSs, mainly caused by buffers.

6.3 Cache Efficiency and Work Efficiency

Finding (2): ForkGraph shows up to a factor of 100× reduction of the number of LLC misses. First, the buffered execution is cache-efficient and it reduces the LLC misses of ForkGraph even with the same amount of work as other GPSs. Second, the work efficient design of FPP queries processing further reduces the amount of total LLC accesses.

Figure 10 shows the profiling of cache performance and the number of edges processed of different GPSs. We present the amount of work as the number of edges processed during processing. For brevity, we show the profiling of LL and NCP applications on two representative graphs for each. We observe similar findings on other graphs. With the buffered execution model and work-efficient optimizations, ForkGraph significantly reduces the total LLC misses and work compared to others. Figure 10a shows that ForkGraph reduces the number of LLC misses by up to a factor of 100×, compared to other GPSs' execution with $t = 1$, and by up to a factor of 60× over others' execution with $t = 10$.

We also count the number of edges processed in executing sequential algorithms on the same data sets. We find that ForkGraph only processes 10.4 – 16.7× more edges than the sequential algorithm (Dijkstra's algorithm) on BC and LL and 5.2 – 9.4× more than the sequential algorithm on NCP.

The significant reduction of LLC misses mainly comes from two optimizations: 1) the LLC-sized partition helps to guarantee that operations are limited within LLC during intra-partition processing and 2) the total reduction of workloads as ForkGraph can be theoretically proved to be work-efficient. As shown in Figure 10b, ForkGraph significantly reduces the number of edges processed on road networks due to work-efficient yielding and scheduling, which is also the reason for significant speedups over the other GPSs. Although ForkGraph shows a similar amount of work as Gemini and GraphIt on Lj and Tw, it only incurs fewer than 1/10

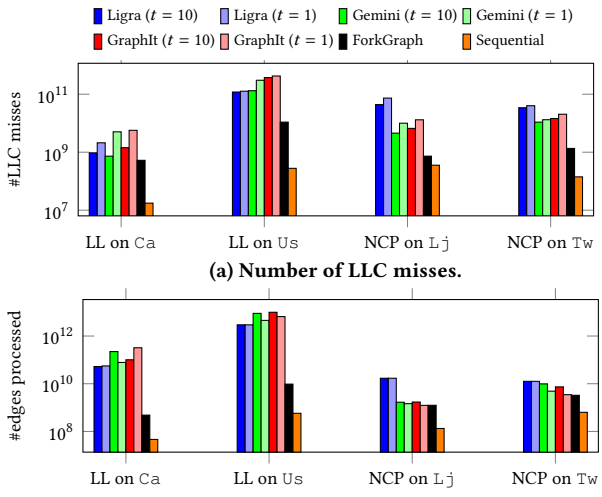


Figure 10: Profiling results of the number of LLC misses and the number of edges processed per FPP query on four GPSs, solving two applications on four graphs.

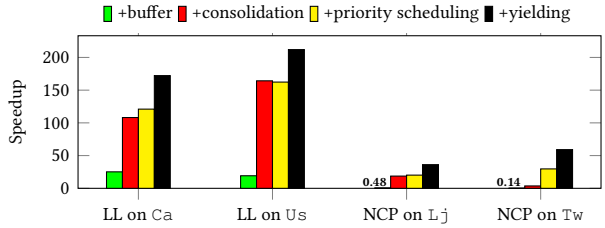


Figure 11: Speedups achieved by applying different optimizations cumulatively to the Ligra baseline.

of the LLC misses as others, delivering near an order of magnitude speedup over them.

6.4 Effects of Individual Techniques

Finding (3): The proposed techniques accumulatively improve the performance of ForkGraph.

We evaluate the performance impacts of major design rationales in ForkGraph, including buffered execution, query-centric operation consolidation, priority-based scheduling, and yielding. We cumulatively enable each of those optimizations one by one on the baseline Ligra ($t = 10$) to study the advantages of individual design decisions. For brevity, we show LL and NCP on four graphs again.

Figure 11 shows the performance improvement. We have the following observations. First, we enable the buffered execution model and sequential execution of FPP queries, denoted as +buffer. +buffer achieves 25× speedups on solving LL on road networks because it benefits from the good locality. Also, we observe that applying the buffered execution model brings negative performance improvements in solving NCP. The reason is that we only finish the processing of a PPR query in the partition when it converges, which brings extra workload when dealing with an excessive number of revisits. Second, we enable the atomic-free processing by query-centric operation consolidation, denoted as +consolidation.

It begins to show significant speedups by both reducing the overhead of atomic operations and the memory overhead caused by expanding superfluous operations to neighbor partitions. Next, we enable the optimizations of priority-based scheduling and yielding, respectively. These two optimizations holistically improve the performance over +consolidation by further 1.3 − 16.2×. The yielding optimization generally provides more significant speedups than others as it cuts off the work directly during the processing, while the priority-based scheduling reduces workloads indirectly.

Impacts of parameter tuning in inter-partition scheduling. We further evaluate the performance impacts of the priority-based scheduling and yielding. We only show the results for BC with 100 SSSPs on `Us` graph for brevity.

Table 4A shows the impact of different priority functors. We make the following observations. First, it is inefficient to schedule the partition with most operations to process ("Max #operations"), even though it is more cache efficient intuitively. As a result, it is even slower than the baseline by picking an arbitrary non-empty buffer to process in each step ("Random"). Second, the performance of the default FIFO scheduling is slightly better than a random scheduling. Third, compared to other priority functors, adopting the "Shortest" priority functor from the corresponding sequential algorithms delivers several times speedups over others.

Table 4B shows the performance of ForkGraph using different yielding thresholds based on the number of edges processed. We define μ to be the number of edges in the partition divided by the total number of queries, which is the theoretical threshold (see the proof in the appendix of the complete version [36]). We change the threshold value of the heuristics at the basis of μ. We can observe that the execution of applying threshold μ results in an execution time near the fastest but not necessarily the fastest. It is because that the threshold is obtained based on the theoretical upper bound of the number of revisits; however, the number of revisits is far below the bound in practice, which makes a larger threshold perform well. As there can be thousands of queries in our experiments for NCP applications on large graphs, we use a larger threshold, 100μ, for these cases.

Table 4C shows the impact of yielding heuristics based on the value updated. We adopt the $\Delta = 50,000$ used in [59] for the same data set `Us` and also test the execution instances with the threshold varied. We present the execution times of ForkGraph with the threshold setting varied from 0.25Δ to 4Δ for brevity. We have the observations as follows. First, when the threshold is large, ForkGraph spends more time as there are more redundant operations abandoned in each partition. Second, when the threshold is small, ForkGraph aggressively yields the processing in a partition and results in a high number of revisits. We choose the thresholds directly adopted from the Δ-stepping algorithm [39, 59] in our experiments.

6.5 Other Results

Due to the space limit, we summarize some experimental results as follows. The reader is referred to the details in the appendix of the complete version [36].

Memory stall distribution. As more than 34% of the execution time of other GPSs is spent on memory stalls, the time spent in ForkGraph is only less than 20% of the total execution time. ForkGraph's design limits the operations to graph partitions in the LLC and thus reduces the percentage of the costly DRAM accesses, which is shown as the low memory stall distribution.

Table 4: Performance of ForkGraph under different priority-based scheduling methods and yielding parameters, solving BC on `Us`.

A. Impacts of priority-based scheduling (yielding enabled).

Priority functor	Random	Max #operations	FIFO	**Shortest**
Execution time (s)	504.3	749.9	491.3	**168.8**

B. Impacts of yielding heuristic 1: on the number of edges processed (priority-based scheduling enabled).

Threshold	0.25μ	0.5μ	μ	$\mathbf{2\mu}$	4μ	No Yielding
Execution time (s)	450.9	412.4	325.6	**238.6**	248.3	1945.8

C. Impacts of yielding heuristic 2: on the operations' values updated (priority-based scheduling enabled).

Threshold	0.25Δ	0.5Δ	$\mathbf{\Delta}$	2Δ	4Δ	No Yielding
Execution time (s)	420.6	297.0	**168.8**	172.1	239.8	1945.8

Scalability. We evaluate the scalability of ForkGraph in the numbers of threads and queries. First, ForkGraph can achieve 7 − 8× speedups when scaling up from one to ten cores (with hyper-threading disabled) for most of the graphs. Second, ForkGraph shows the capability to remain at a high throughput with processing growing numbers of FPP queries.

Effects of Partition and Cache Size. We empirically study the effects of graph partitioning methods, partition sizes, and cache sizes. First, ForkGraph on METIS partition shows up to 14.1× and 4.2× speedups over a random partition and Gemini's lightweight partition [66], respectively, when executing LL and BC on different graphs. ForkGraph on METIS partition shows 1.1 − 3.6× speedups over other partitioning methods when executing NCP on different web and social networks. Second, our results show that using LLC-size partitions achieves the best performance for most cases. This is because the intra-partition processing can cause heavy cache thrashing if the partition size is larger than the LLC size. Further, if we divide the graph into small partitions, there can be a large number of partitions to schedule, which incurs a high overhead.

7 RELATED WORK

Graph processing on multi-core architectures. There has been substantial works on efficient parallel graph systems and frameworks over the past years, including [19, 35, 37, 62, 66] among many others. Ligra [45], Galois [40], as the representative shared-memory graph processing frameworks. These frameworks are designed with abstractions for users to conduct graph computations while leveraging hardware properties such as memory locality and multi-cores efficiently. GraphIt [62] is a DSL for graph processing, which generates parallel implementations of graph applications. GraphIt integrates a scheduling language to ease the exploration of the complicated tradeoff space. We refer the reader to [38, 54] for excellent surveys of this growing literature.

Zhang et al. [62] and Lakhotia et al. [31] propose to improve the cache utilization by breaking the graph into segments that fit in the LLC. In this way, random accesses at each partition are limited in the cache, avoiding costly memory accesses. Similarly, ForkGraph also partitions graphs into LLC-size partitions. Unlike their approaches, ForkGraph designs a buffered execution model to

specifically optimize processing queries simultaneously to enable the work-efficient and cache-efficient FPP processing.

Other related works in graph processing systems. Yan et al. propose Blogel [55], a block-centric, distributed graph processing framework. Both ForkGraph and Bogel consider a partition of a graph as a computing block (instead of a vertex or an edge). The block-centric computing model proposed by Bogel helps decrease the number of iterations compared to a vertex-centric algorithm and also reduces the number of messages transmitted through the network in the distributed setting. ForkGraph is different from Blogel since Blogel is distributed, and ForkGraph is in-memory. Besides, Blogel executes one query at a time, while ForkGraph focuses on the efficiency of inter-query parallelism among multiple FPP queries.

In addition, Zhao et al. propose GraphM [63], a storage system that efficiently handles consolidated, out-of-core concurrent graph queries. GraphM divides the graph into partitions and sets the highest priority to the partition with the most jobs. However, as shown in Table 4, this approach is inefficient for FPP queries because GraphM's scheduling is designed for the out-of-core, BSP model, but not work-efficient for in-memory execution.

Processing multiple graph queries simultaneously. Zhang et al. [61] propose CGraph, one of the the state-of-the-art disk-based graph processing systems that handle multiple queries simultaneously. CGraph efficiently amortizes the high disk access cost when processing multiple queries simultaneously by a subgraph-based scheduling algorithm. Differently, ForkGraph targets cache-efficient FPP queries processing in memory. While CGraph processes all queries in every iterations, ForkGraph only processes a subset of FPP queries in the partition-level granularity, leveraging work-efficient sequential executions. Hauck et al. [24] motivate the experimental studies of processing concurrent queries based on the multi-user setups in classic relational enterprise database environments or web-scale environments. They study the inter- and intra-parallelism of handling different types of graph queries by assigning different threads to different instances of Galios [40]. However, executing multiple instances contains inevitable contentions of the system resources, including memory and threads. The authors provide an in-depth discussion of the limitation of GPSs in handling multiple queries but do not come out with a suitable solution.

MS-BFS [50] and iBFS [34] are proposed to accelerate multiple BFS queries using multi-core CPUs and GPUs (Graphics Processing Units), respectively. Instead of visiting vertices individually for each BFS, both work leverage joint frontier queue and bitwise operations for multiple BFS queries. However, the techniques specifically serve only BFS queries, losing the generalities.

Buffering accesses to index structures. The access buffering model proposed in this work is inspired by Zhou and Ross's work [64, 65] on buffering accesses to tree-structured indexes, e.g., B^+-tree [9], and many other related buffering techniques [20, 25, 43, 48]. The buffering techniques are mainly used for avoiding cache thrashing between query accesses by processing buffered lookups at index nodes. Besides, He et al. [25] develop a cache-oblivious design on buffering accesses. Those previous works inspire our buffer execution model. However, the previous studies work on a relatively simpler problem on tree accesses, which always go from the root to leaf nodes. In FPPs, the access pattern is more irregular. A query could start randomly from any vertex and expand to different neighbors, and the access can be repeated, unlike the accesses on trees. Therefore, this work develops novel intra- and inter-partition mechanisms to improve the work and cache efficiency.

Other related topics from relational databases. The cache-aware techniques in ForkGraph have been greatly inspired by the substantial studies from relational databases. Particularly, Harizopoulos et al. propose Qpipe [22], an relational query engine that exploits overlap across concurrent queries at runtime. Qpipe buffers data pages brought by queries and reuses them for other submitted queries. Moreover, systems like Dora [41], H-Store [28], and many other works partition the data logically or physically to enable concurrent transactions execution in parallel on partitioned data, which share the similar spirit of ForkGraph on LLC-sized graph partitions. The logging solution proposed by Johnson et al. [27] aggregates requests from threads to reduce the contention among them by making the requests independent from the number of threads. This is similar to the consolidation process of ForkGraph.

However, the techniques proposed in previous studies cannot be directly applied in this work since these studies are in relational databases and this work focuses on graph processing. Particularly, this work has the following differences. First, the order of operations in Qpipe and other works in relational databases does not affect the work efficiency, while the order of graph query operations affect the work efficiency. Second, compared with the consolidation techniques proposed by Johnson et al. [27], ForkGraph not only reduces the contention but also leverages the algorithmic properties such as priority and yielding in graph algorithms to reduce redundant computation.

8 CONCLUSIONS

As graph applications emerge, we observe a common and costly fork-processing pattern (FPP) that launches many independent queries from different source vertices on the same graph. Our profiling studies demonstrate that existing parallel graph systems suffer from severe cache thrashing due to irregular graph structures and many parallel queries in FPP. Thus, we propose ForkGraph, a cache-efficient graph processing system for processing FPPs on in-memory graph data. Specifically, ForkGraph embraces a cache-efficient buffer execution model to handle operations of many FPP queries. Moreover, we develop effective intra- and inter-partition mechanisms to improve work efficiency. Our evaluations on real-world graphs show that ForkGraph significantly outperforms state-of-the-art graph processing systems (including Ligra, Gemini, and GraphIt) by two orders of magnitude speedups.

ACKNOWLEDGMENTS

This project is supported by the grant "Asian Institute of Digital Finance" awarded by National Research Foundation, Singapore and administered by the Infocomm Media Development Authority under its Smart Systems Strategic Research Programme in 2020. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD (2013)*. 349–360.

[2] Morteza Alamgir and Ulrike Von Luxburg. 2010. Multi-agent random walks for local clustering on graphs. In *ICDM (2010)*. IEEE, 18–27.

[3] Baruch Awerbuch and Tom Leighton. 1994. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In *STOC (1994)*. 487–496.

[4] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2011. 10th DIMACS Implementation Challenge-Graph Partitioning and Graph Clustering.

[5] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC (2012)*. 1–10.

[6] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IISWC (2015)*. 56–65.

[7] Guy E Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel shortest paths using radius stepping. In *SPAA (2016)*. 443–454.

[8] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[9] Douglas Comer. 1979. Ubiquitous B-tree. 11, 2 (1979).

[10] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011), 1–25.

[11] Camil Demetrescu, Andrew V Goldberg, and David Johnson. 2008. 9th DIMACS implementation challenge–Shortest Paths (2006). (2008).

[12] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA (2017)*. 293–304.

[13] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2018. Theoretically efficient parallel graph algorithms can be fast and scalable. In *SPAA (2018)*. 393–404.

[14] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *GRADES & NDA (2020)*. 1–8.

[15] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[16] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2006. Ant colony optimization. *IEEE computational intelligence magazine* 1, 4 (2006), 28–39.

[17] Santo Fortunato and Darko Hric. 2016. Community detection in networks: A user guide. *Physics reports* 659 (2016), 1–44.

[18] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing large graphs on GPUs with unified memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1119–1133.

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI (2012)*. 17–30.

[20] Goetz Graefe and Harumi Kuno. 2011. Modern B-tree techniques. In *ICDE (2011)*. IEEE, 1370–1373.

[21] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *SIGKDD (2016)*. 855–864.

[22] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD (2005)*. 383–394.

[23] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. *Acm Sigplan Notices* 46, 8 (2011), 3–12.

[24] Matthias Hauck, Marcus Paradies, and Holger Fröning. 2017. Can Modern Graph Processing Engines Run Concurrent Queries Efficiently?. In *GRADES (2017)*. 1–6.

[25] Bingsheng He and Qiong Luo. 2008. Cache-Oblivious Databases: Limitations and Opportunities. *ACM Trans. Database Syst.* 33, 2, Article 8 (June 2008), 42 pages. https://doi.org/10.1145/1366102.1366105

[26] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2017. Parallel algorithm for incremental betweenness centrality on large graphs. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2017), 659–672.

[27] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 681–692.

[28] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[29] George Karypis and Vipin Kumar. 1998. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing* 48, 1 (1998), 96–129.

[30] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW (2010)*. 591–600.

[31] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2019. GPOP: A cache and memory-efficient framework for graph processing over partitions. In *PPoPP (2019)*. 393–394.

[32] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[33] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[34] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *SIGMOD (2016)*. ACM, 403–416.

[35] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. In *UAI (2010)*. 340–349.

[36] Shengliang Lu, Shixuan Sun, Johns Paul, Yuchen Li, and Bingsheng He. 2021. Cache-Efficient Fork-Processing Patterns on Large Graphs. arXiv:2103.14915 [cs.DB]

[37] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD (2010)*. 135–146.

[38] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *Comput. Surveys* 48, 2 (2015), 1–39.

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

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP (2013)*. 456–471.

[41] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented transaction execution. *Proceedings of the VLDB Endowment* 3, ARTICLE (2010).

[42] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *SIGKDD (2014)*. 701–710.

[43] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. In *SIGMOD (2016)*. 1523–1538.

[44] Julian Shun. 2020. Practical parallel hypergraph algorithms. In *PPoPP (2020)*. 232–249.

[45] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[46] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *SPAA (2012)*. 68–70.

[47] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W Mahoney. 2016. Parallel Local Graph Clustering. *Proceedings of the VLDB Endowment* 9, 12 (2016).

[48] Tomáš Skopal, David Hoksza, and Jaroslav Pokorný. 2007. Construction of tree-based indexes for level-contiguous buffering support. In *DASFAA (2007)*. 361–373.

[49] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[50] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. 2014. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment* 8, 4 (2014), 449–460.

[51] Di Wang, Kimon Fountoulakis, Monika Henzinger, Michael W. Mahoney, and Satish Rao. 2017. Capacity Releasing Diffusion for Speed and Locality. In *ICML (2017)*. JMLR.org, 3598–3607.

[52] Joseph Wang and D Eppstein. 2001. Fast approximation of centrality. In *SODA (2001)*. 228–229.

[53] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *SIGMOD (2016)*. ACM, 1813–1828.

[54] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Found. Trends Databases* 7, 1–2 (2017), 1–195.

[55] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[56] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[57] Jin Y Yen. 1970. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quart. Appl. Math.* 27, 4 (1970), 526–530.

[58] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *management Science* 17, 11 (1971), 712–716.

[59] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *CGO (2020)*. 158–170.

[60] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *(Big Data (2017)*. IEEE, 293–302.

[61] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *ATC (2018)*. 441–452.
[62] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *OOPSLA (2018)* 2 (2018), 121.
[63] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *SC (2019)*. ACM, 3.
[64] Jingren Zhou and Kenneth A Ross. 2003. Buffering accesses to memory-resident index structures. *Proceedings of the VLDB Endowment* (2003).
[65] Jingren Zhou and Kenneth A Ross. 2004. Buffering databse operations for enhanced instruction cache performance. In *SIGMOD (2004)*. 191–202.
[66] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI (2016)*. 301–316.