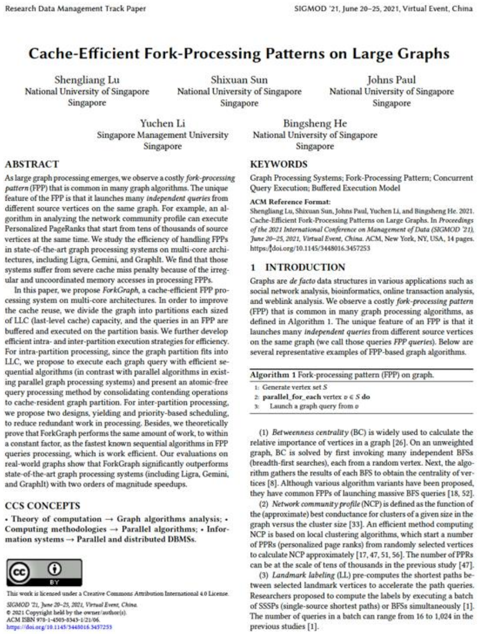
Cache-Efficient Fork-Processing Patterns on Large Graphs

Paper Review // 6.5060 Algorithm Engineering

Cache-Efficient Fork-Processing Patterns on Large Graphs

Proceedings of the 2021 International Conference on Management of Data (SIGMOD ’21)



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.

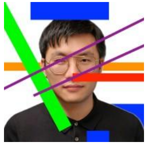
```
Generate vertex set S
parallel_for_each vertex v ∈ S do
    Launch a graph query from v
```

(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].

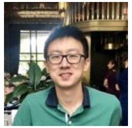
National University of Singapore



Shengliang Lu
National University of Singapore
Singapore

Shixuan Sun
National University of Singapore
Singapore

Johns Paul
National University of Singapore
Singapore

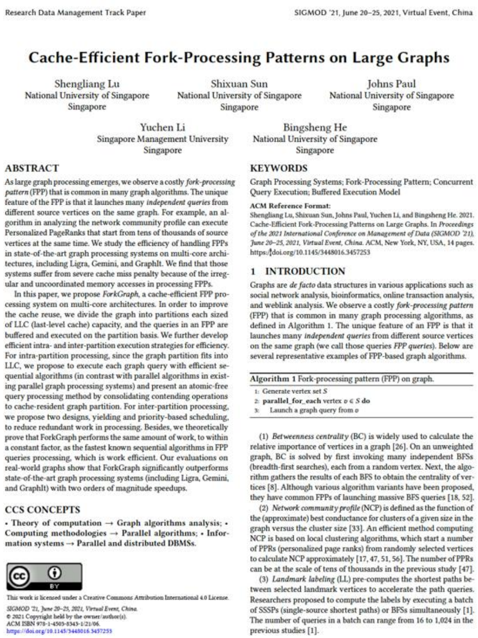


Yuchen Li
Singapore Management University
Singapore

Bingsheng He
National University of Singapore
Singapore

Cache-Efficient Fork-Processing Patterns on Large Graphs

Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)



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.

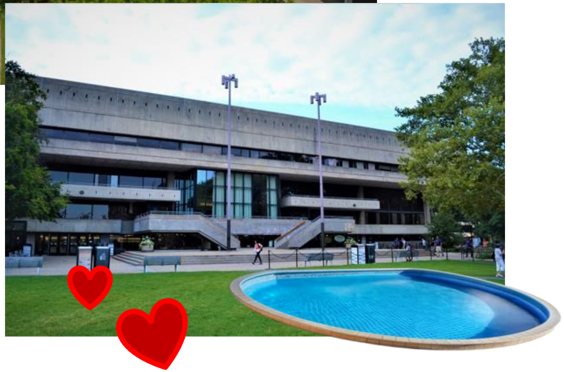
```
1: Generate vertex set S
2: parallel_for_each vertex v ∈ S do
3:    Launch a graph query from v
```

(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].

Proposed improvements:



Fork

ForkGraph

The Fork-Processing Pattern (FPP)

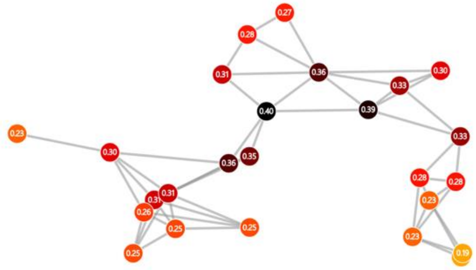
Algorithm 1 Fork-processing pattern (FPP) on graph.

1: Generate vertex set S
2: **parallel_for_each** vertex $v \in S$ **do**
3: Launch a graph query from v

Many graph processing algorithms look like this (also learning, mining)

Betweenness Centrality

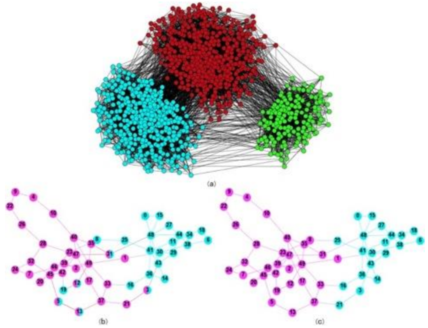
Many independent BFS



www.networkpages.nl

Network Community Profile

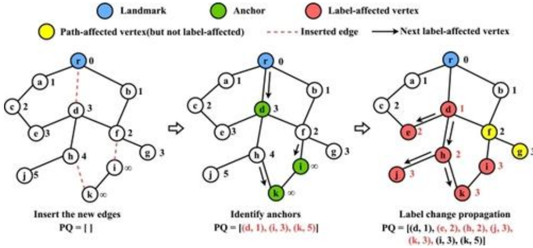
Multiple PPRs from randomly selected vertices



Finding Communities by their Centers, Chen et al.

Landmark Labeling

Multiple SSSP / BFS simultaneously in a batch



FulBM: Fast Fully Batch Maintenance for Landmark-based 3-hop Cover Labeling, Zhang et al.

This leads to a tremendous amount of redundancy and LLC misses.

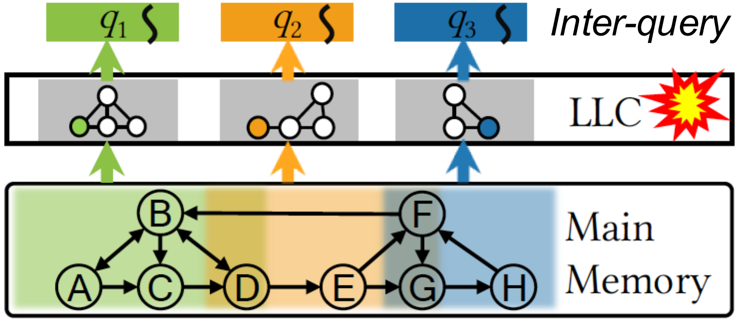
**t = number of threads assigned to a query*

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

- 90% of processing time is FPP
- LLC misses are the bottleneck
  - 34-40% of time spent in memory units is stalled memory cycles from LLC misses
  - 55% if t=1 (inter-query parallelism)

Where are the LLC misses coming from?



- Each query *q* is processed separately
- Uncoordinated cache use leads to thrashing
- Bonus redundancies

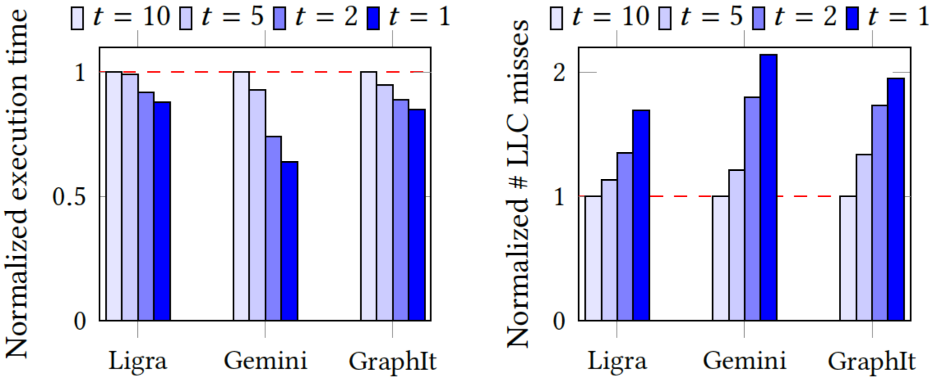


Figure 1: GPSs' performance affected by cache contention with different numbers of threads assigned to each query, tested with 10,000 PPRs on LiveJournal graph.

Existing Graph Processing Systems (2021)

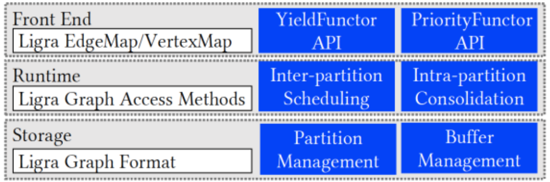
Fastest implementation, maintained
The authors <333 Ligra

Distributed graph processing system
Tested with message-passing functions disabled

State of the art DSL (domain specific language)
Cache-optimized to break graph into LLC-size segments and prevent random access within the cache.
Not great for inter-query parallelism.

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



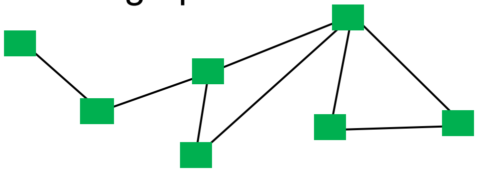
ForkGraph (à la Ligra)

1. Full graph *G* is partitioned into LLC-size chunks

2. Each partition is associated with a buffer for storing its operations

- Operations from different queries are buffered by partition and executed in a batch
- Naturally reduces LLC misses because each partition fits into cache

3. Queries are performed on the graph



Consolidation

- Operations from different queries are adjacent in the buffer and can be run without atomics
- Queries can be sorted to prevent redundancies

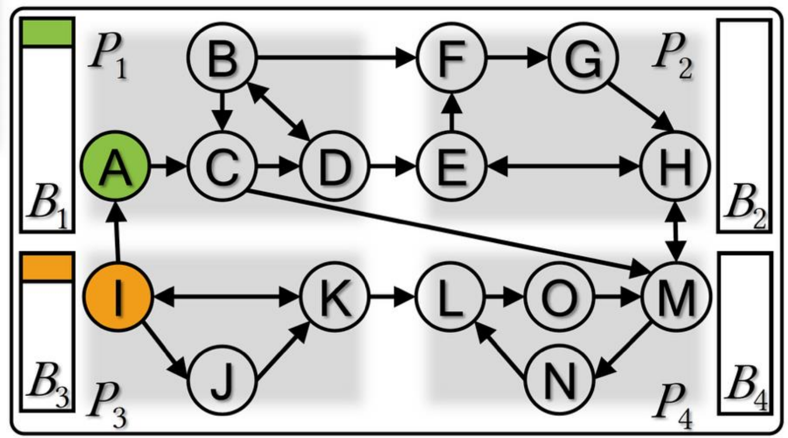
Intra-partition

- Sequential implementations for multiple simultaneous operations
- One-thread per buffered operation (effectively inter-query)

Inter-partition

- Need to decide which partitions to query in which order
- Minimize redundant work across partitions

Challenges with intra- and inter-partition parallelism



Example problem on a partitioned graph

(p.s. graph partitioning done using METIS)

Inter-

Multiple partitions processed independently

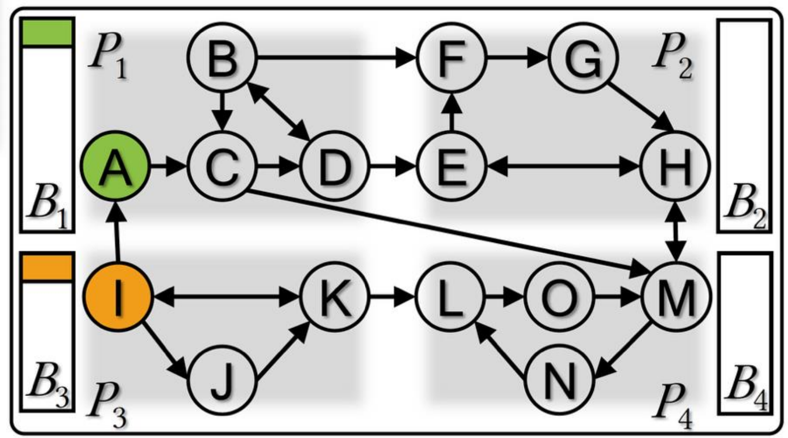
- Poor cache-efficiency in existing GPSs
- Memory-level parallelism
- How to determine execution order?

Intra-

Multiple operations running simultaneously within a partition

- Requires costly synchronization
- How to make work-efficient?

Outstanding questions in intra- and inter-partition parallelism



Example problem on a partitioned graph

(p.s. graph partitioning done using METIS)

Inter-

Multiple partitions processed independently

1. When to terminate processing in a given partition (yielding)
2. Which partition to go to next

Intra-

Multiple operations running simultaneously within a partition

1. Work-efficient sequential implementations for multiple operations simultaneously
2. Remove atomics through consolidation (query-centric) and sorting (remove redundancies by prioritizing)

ForkGraph's approach to efficient parallelization

Parallel algorithms in existing GPS implementations require significant overhead (synchronization, locking, scheduling) and are not work efficient

ForkGraph proposes a cache-efficient intra-partition processing method

Parallelism on the **level of individual queries**, not partitions

Each thread retrieves from the buffer and processes the operations of a given query sequentially using known state-of-the-art sequential algorithms

Execution flow

Algorithm 2 ForkGraph: FPP Processing on graph partitions *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
```

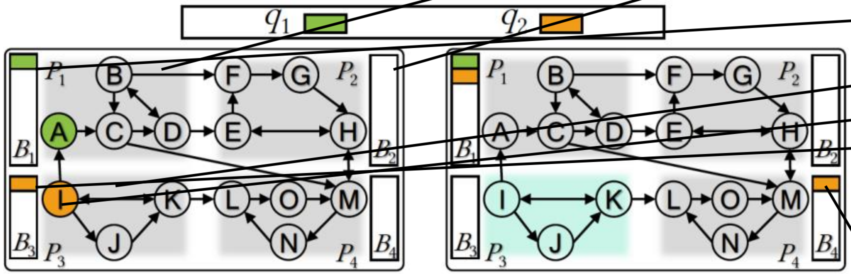
Graph has been pre-partitioned into **P**

Initialize buffers (dynamic sized, contiguous memory)
+ assign initial queries

While non-empty buffers exist:

- Find the next partition to process and process it
- Consolidate the operations stored in that partition's buffer by query and assign each query to a thread
- Track newly generated operations and assign to own buffer to be processed (if internal to partition) or held in a local buffer (like an outbox)
- Yield each query if it exceeds threshold (e.g. number of edges processed)
- Local buffer gets distributed to other partitions' buffers in batches at end of processing

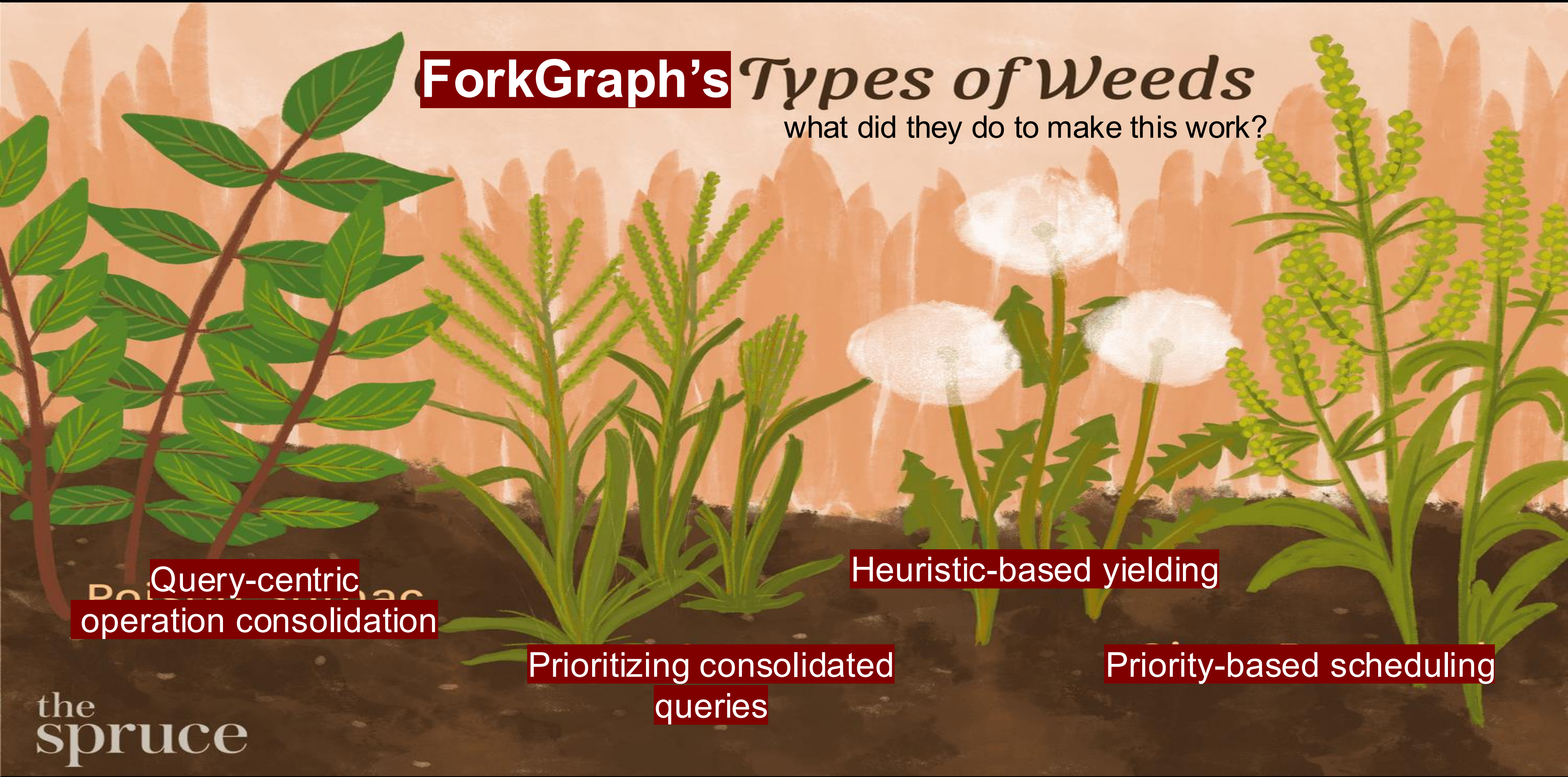
Execution flow



(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.

- Graph has been pre-partitioned into **P**
- Initialize buffers (dynamic sized, contiguous memory) + assign initial queries
- While non-empty buffers exist:
  - Find the next partition to process and process it
  - Consolidate the operations stored in that partition's buffer by query and assign each query to a thread
  - Track newly generated operations and assign to own buffer to be processed (if internal to partition) or held in a local buffer (like an outbox)
  - Yield each query if it exceeds threshold (e.g. number of edges processed)
  - Local buffer gets distributed to other partitions' buffers in batches at end of processing



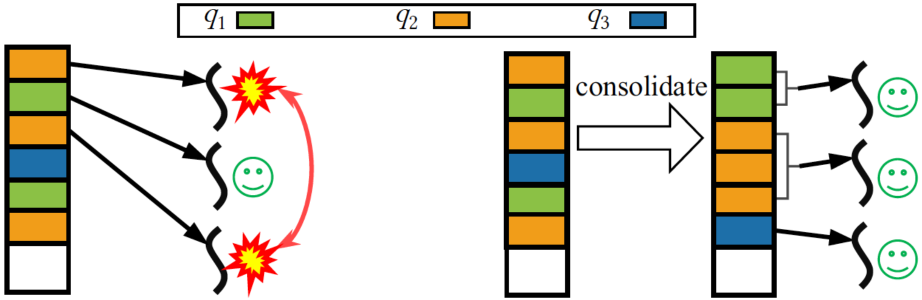
ForkGraph's

what did they do to make this work?

- Query-centric operation consolidation
- Prioritizing consolidated queries
- Heuristic-based yielding
- Priority-based scheduling

INTRA: Query-Centric Operation Consolidation

procedure IntraPartProcess(P_c)
  Consolidate operations in P_c.buffer according to each query
  parallel_for_each $q \in Q$ **do**



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

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

What is it?

- Assign all operations from same FPP query to individual threads
- Process all FPP queries in parallel

Why is it needed?

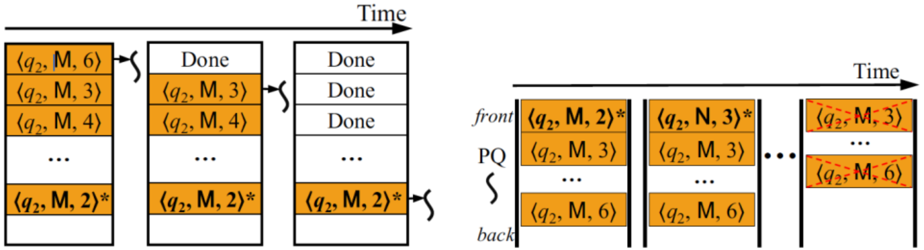
- Access conflicts from different threads processing operations of same query simultaneously
- Requires locking and synchronization operations (expensive)

What are the benefits?

- Operations of one query can be sequential and atomic-free
- Avoids stride memory access (query data is shared in contiguous memory space)

INTRA: Prioritizing Consolidated Queries

```
parallel_for_each q ∈ Q do
    for op ∈ P_c.buffer.getOps(q) do
        newOps ← COMPUTE(op, P_c)
```



(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 $*$.

What is it?

- Within each group of consolidated operations, order by priority
- Priority functor is determined by user (established literature)

Why is it needed?

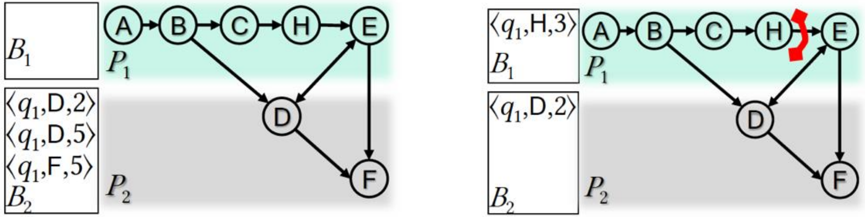
- Redundant processing (e.g. exploring shorter paths before longer ones, then pruning)

What are the benefits?

- Reduces the number of operations by prioritizing those most likely to converge

INTER: Heuristic-Based Yielding

```
if CanYield(P_c, q, YieldFunctor()) then
    Yield() and goto Line 10          ▷ terminating q, see Sec. 5
```



(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. Shortest path query q_1 starts at vertex A in P_1. All edges are with unit lengths.

What is it?

- Pause running a query if it satisfies user-determined heuristics (will be processed later in the partition buffer)

Heuristics

1. Number of edges processed (too many?
2. Operations' values updated (do they exceed a Delta range?)

Why is it needed?

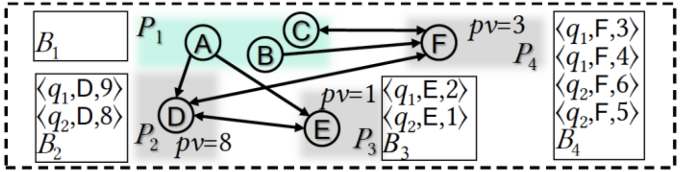
- Diminishing returns (e.g. PPR can converge to local stable states that are disrupted by external partitions; SSSP is more likely to spend time exploring unpromising paths if it stays within one partition)

What are the benefits?

- Reduces redundant operations
- Improves work efficiency of FPP operations

INTER: Priority-Based Scheduling

procedure SCHEDULENEXTPART()
  nextP ← get the next partition according to PRIORITYFUNCTOR()
  return nextP



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.

What is it?

- A decision-making process for selecting the next partition by priority

Methods tested

1. Random (worst)
2. Max # of operations (maximize reuse of cache content? But more redundant…)
3. FIFO is default
4. Best is user-specified, algorithm-specific priority functors

Why is it needed?

- Incorrect ordering of partitions makes it likely that you will repeatedly revisit the same partitions
- Pick the partitions that are buffering the most promising operations for convergence

What are the benefits?

- Reduces redundancy and accelerates convergence

Effect of individual techniques*

Cuts the total number of operations

Reduces memory overhead and atomic operations

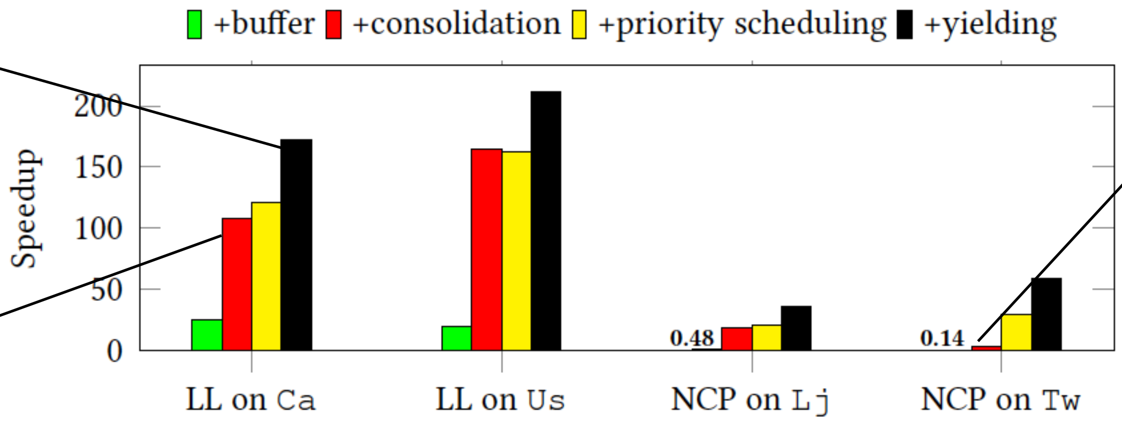


Figure 11: Speedups achieved by applying different optimizations cumulatively to the Ligra baseline.

Negative performance improvements!

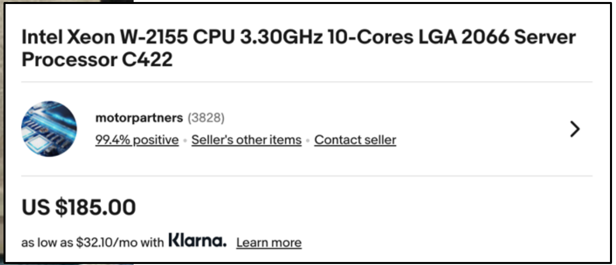
Only finishing PPR query when complete in partition, which is highly redundant.

*can be further improved by parameter tuning of yielding conditions

Experimental Evaluation

Hardware setup and implementation

www.ebay.com



10-core Intel Xeon W-2155 CPU ~~(hyperthreading disabled)~~
256GB memory
3.3GHz frequency (there's a turbo mode up to 4.5GHz)
13.75MB LLC

Compiled in
- g++ 7.5.0
- with -O3 flag (highly optimized)
- OpenMP enabled (multi-platform shared memory)

Some fun facts
- STL priority queue suffices due to low scheduling workload (#P << #V)
- Buckets implemented using GraphIt's parallel vector structure (dynamic-sized and contiguous-memory)
- METIS used for partitioning

Graphs tested

Graph	Source	#V	#E	$\bar{d}$	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

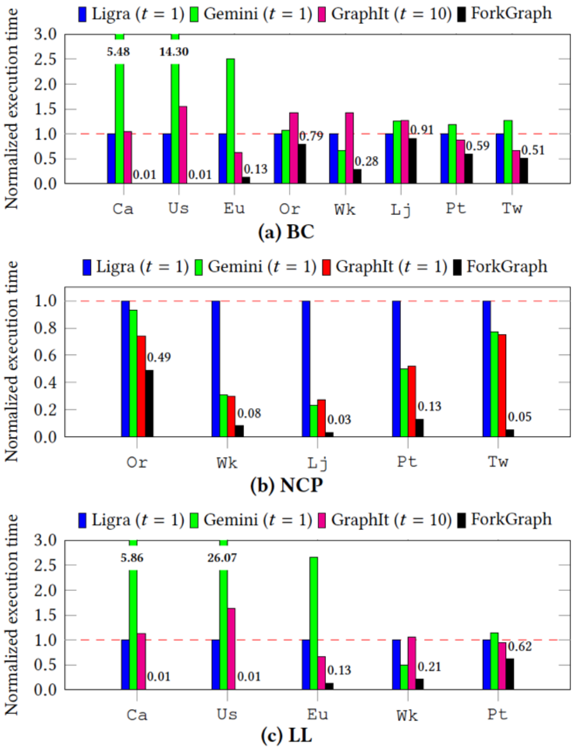
Road networks

Social Networks

Hyperlink network

Citation network

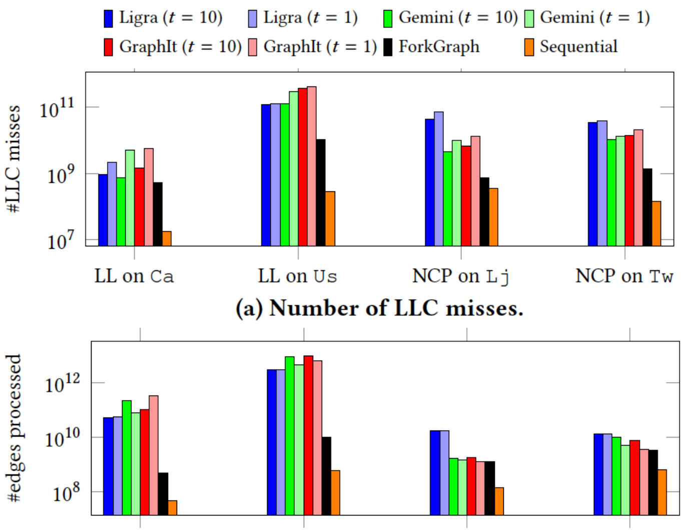
Overall performance



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

1. Accelerates convergence within partitions with low cache thrashing and uses sequential algorithms for work-efficiency
2. Gemini is hamstrung by message-passing materialization overhead (even if message-passing is disabled)
3. GraphIt does better with more threads (despite being optimized for single-query cache usage) because LLC misses overwhelm the advantages of inter-query parallelism.

Cache performance



(a) Number of LLC misses.

(b) Number of edges processed.

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.

1. Reduces LLC misses by factor of up to 100x for *t=1* and up to 60x for *t=10*
2. Only processes 10.4-16.7x more edges on BC, LL than Dijkstra's algorithm (sequential) and 5.2-9.4x more edges on NCP than sequential
3. Similar work on Lj and Tw but fewer cache misses makes it significantly faster

Bonus findings

1. **20% of time spent on memory stalls** vs >34% by other GPS
2. **Scalable** (7-8x speedup from 1 to 10 cores, and **high throughput** even with additional FPP queries)
3. **METIS partition** up to 14.1x faster than random partition, 4.2x faster than GraphIt lightweight partition
4. **LLC-sized partitions** tend to perform best (too big and you exceed cache, too small and you incur a large scheduling overhead)

Questions

- For Julian: Did Ligra end up implementing ForkGraph?
- ForkGraph is designed for a single multicore node, how does it scale to distributed memory environments? What happens when each partition has its own node and own cache? Does coordinating the partitions and buffers over the network negate the cache-efficiency gains?
- What happens when two FPP queries are interdependent?
- It only seems applicable to certain kinds of FPP algorithms that can be expressed as sequences of small operations (e.g. BFS). What if you try to do multiple algorithms together (e.g. BFS + PPR)? Or need to perform triangle counting?
- Dependence on heuristics is concerning and might not translate to other hardware systems.