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

Xiangyu Zhi
Xiao Yan*

Bo Tang
Ziyao Yin

Yanchao Zhu
Minqi Zhou

DANIEL SCHAFFER

MARCH 12, 2024

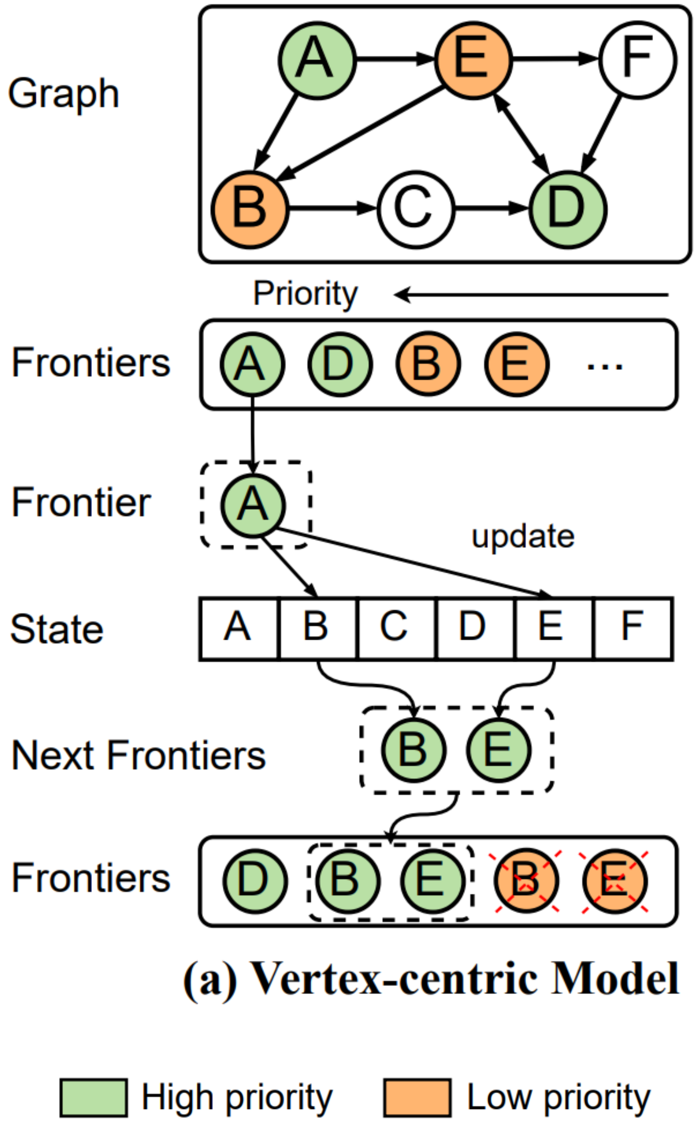
Introduction

Review: Graph frameworks

- Allow the execution of arbitrary graph algorithms
- Consider the current state of a vertex and
  - Update the neighbors of that vertex
  - Maintain *frontiers* consisting of updated neighbors
    - These will be processed soon
- Focus here on single-machine, multi-core, in memory execution

Vertex-centric frameworks

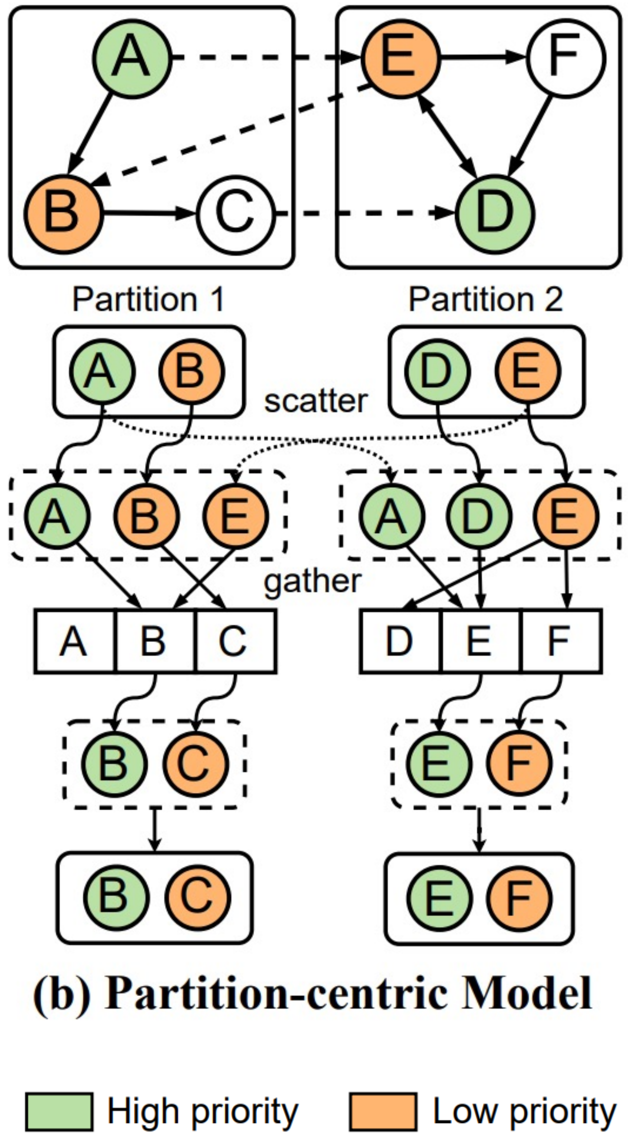
- Process one *frontier* vertex at a time
  - Pop from a priority queue of *frontiers*
  - Higher priority goes to vertices that will make more "progress"
- Then, update the neighbors
  - Re-add them to the queue with new priorities
- Work-efficient, measured by the # of updates
  - AKA the number of edges traversed
- Not cache efficient → memory stall



(a) Vertex-centric Model

Partition-centric frameworks

- Process one partition of vertices at a time
  - Sized to fit into cache
- Copy the frontiers of each partition to the partitions of their neighbors (*scatter*)
- Update all affected vertices in each partition using the copied fronter vertices (*gather*)
- Cache-efficient: working partition is always in the cache
- But, not work-efficient without a priority order



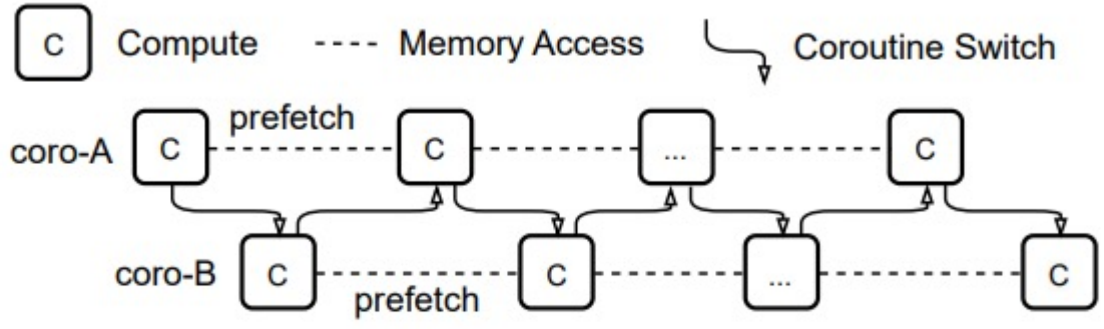
(b) Partition-centric Model

Work vs Cache tradeoff

	System	SSSP		
		Memory bound	# of edges (M)	Time (ms)
	Ligra	65.3%	569	1270
	Gemini	55.2%	573	954
	GraphIt	60.5%	145	782
Partition-centric	GPOP	32.9%	604	594
Vertex-centric	Galois	70.2%	89	513

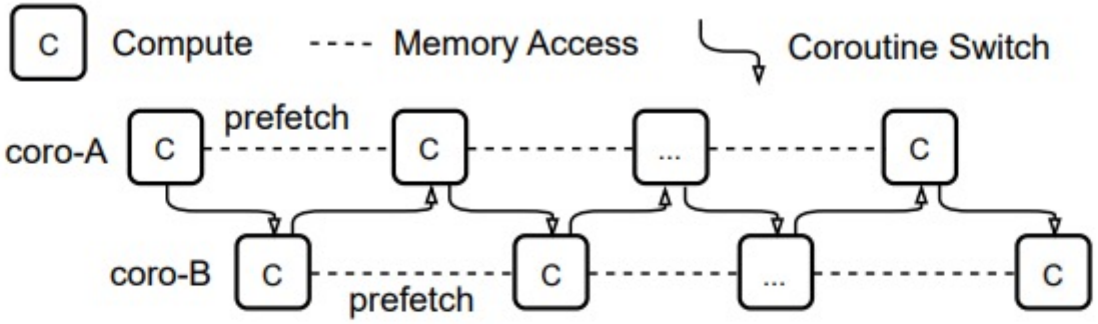
A first idea

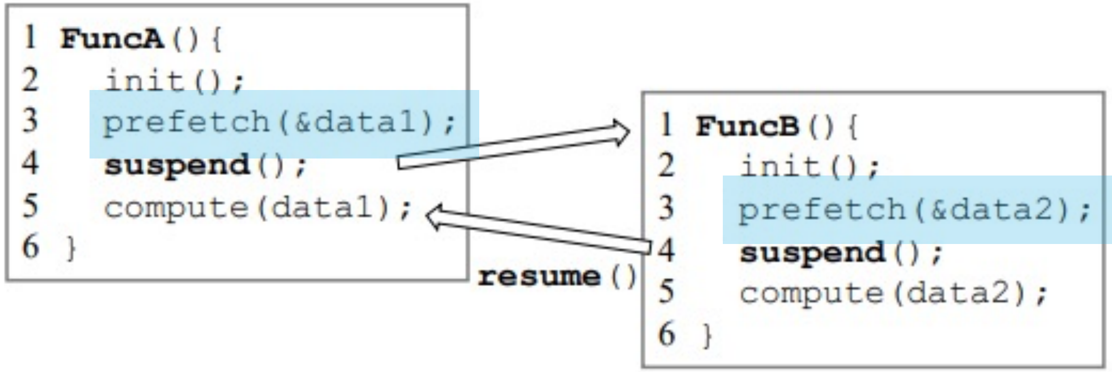
- Do other work while we wait for data to load
  - This can only provide a limited improvement
- But, this is still a useful approach in general
- Generally, the CPU will do this in thread-switching, but it takes time
  - Automatically chooses places to switch
  - Needs to move call stack and local variables each time



Coroutines

- If we know some information about the workflow, we can do much better at the software level
- *Coroutines* are much faster to switch
  - functions with specific suspend and resume points
  - Retain their local variables in allocated memory
  - Do not require a separate call stack





Prefetching data

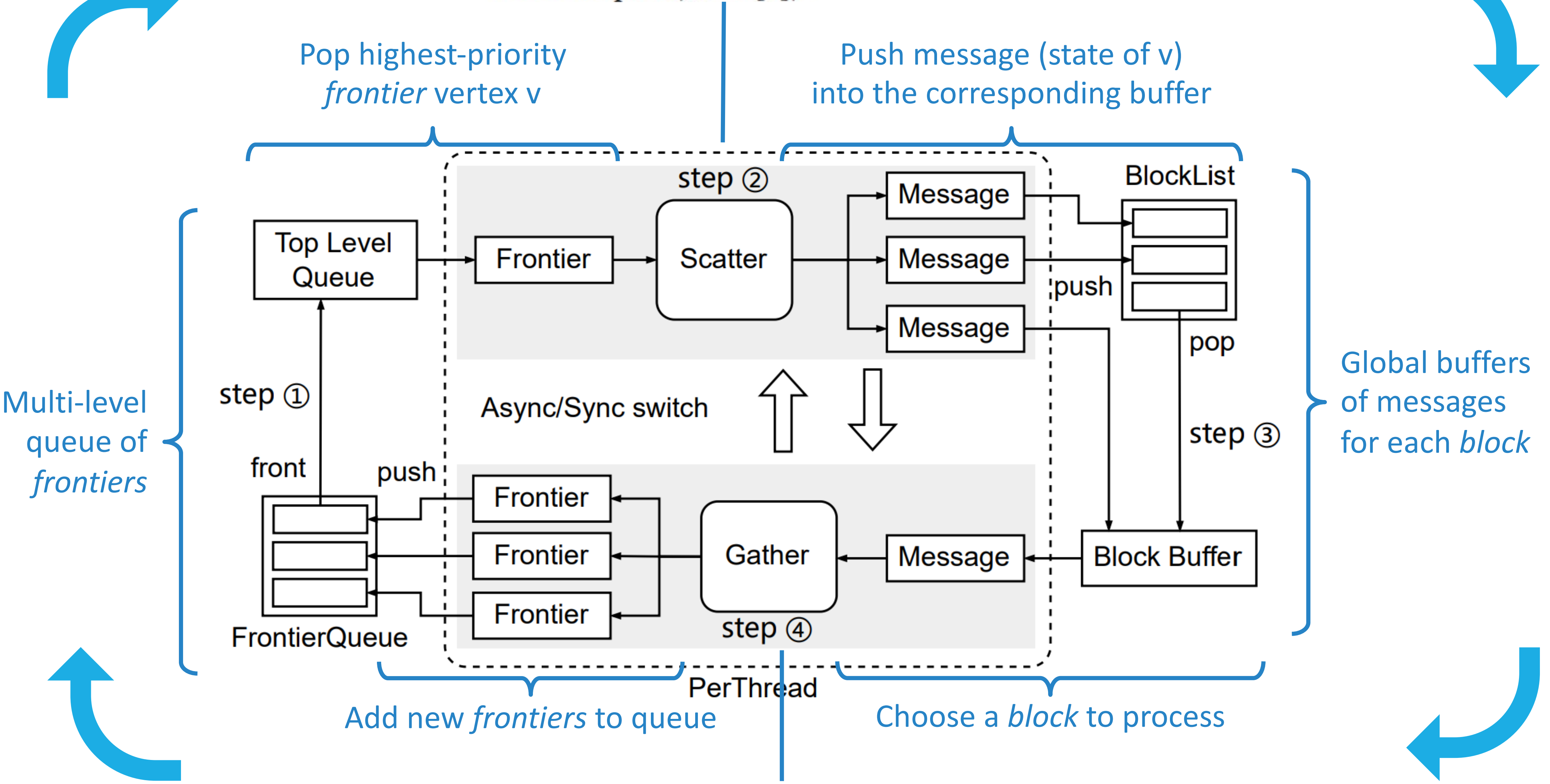
- Retrieving data into the cache before it is actually accessed
- System may try to guess automatically
- If we know what data will be needed, we can specify arbitrary data
  - This is key for coroutines
- Widely used to avoid/hide cache misses in many libraries
- Prefetched data is only valid if no writes are made

The CoroGraph Framework

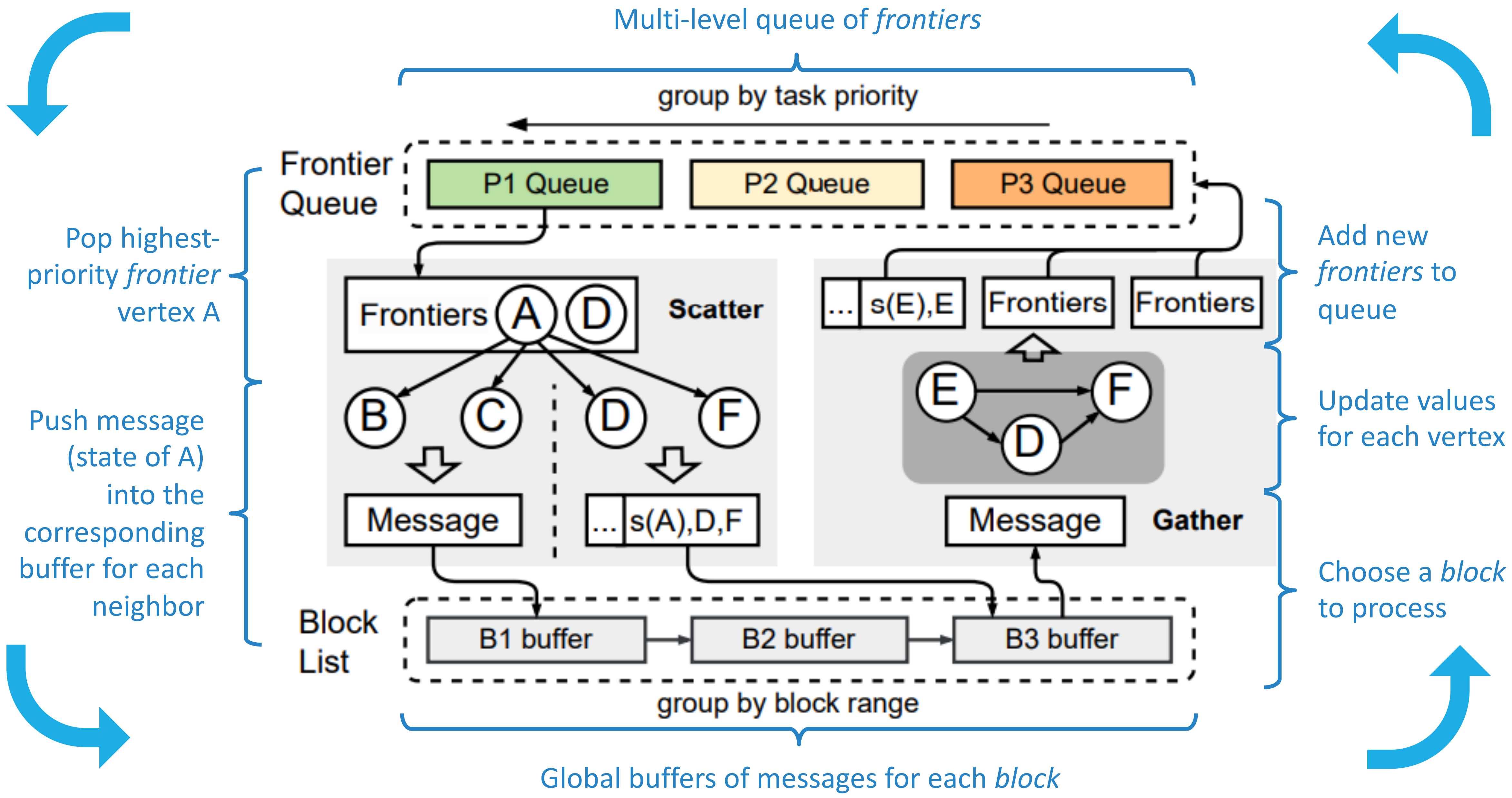
Key idea: CoroGraph

- Use both the vertex-centric and partition-centric approaches for different steps of the algorithm
- Process frontiers in priority order using a queue
  - Generate updates to all neighboring nodes
  - Work-efficiency of vertex-centric frameworks
- Perform updates to all nodes in a partition (block) at once
  - *Gather* phase, one thread per partition
  - Cache-efficiency of partition-centric frameworks
  - No possibility of parallel writes to the same partition

for u ∈ v.ngh **do** // *traverse neighbors of v*
BlockList.push(u, Res[v])



for (u, Res[v]) ∈ msg **do**
if UPDATE(Res[u], Res[v]) **do** // *if u is new frontier*
FrontierQueue.push(u, Res[u])



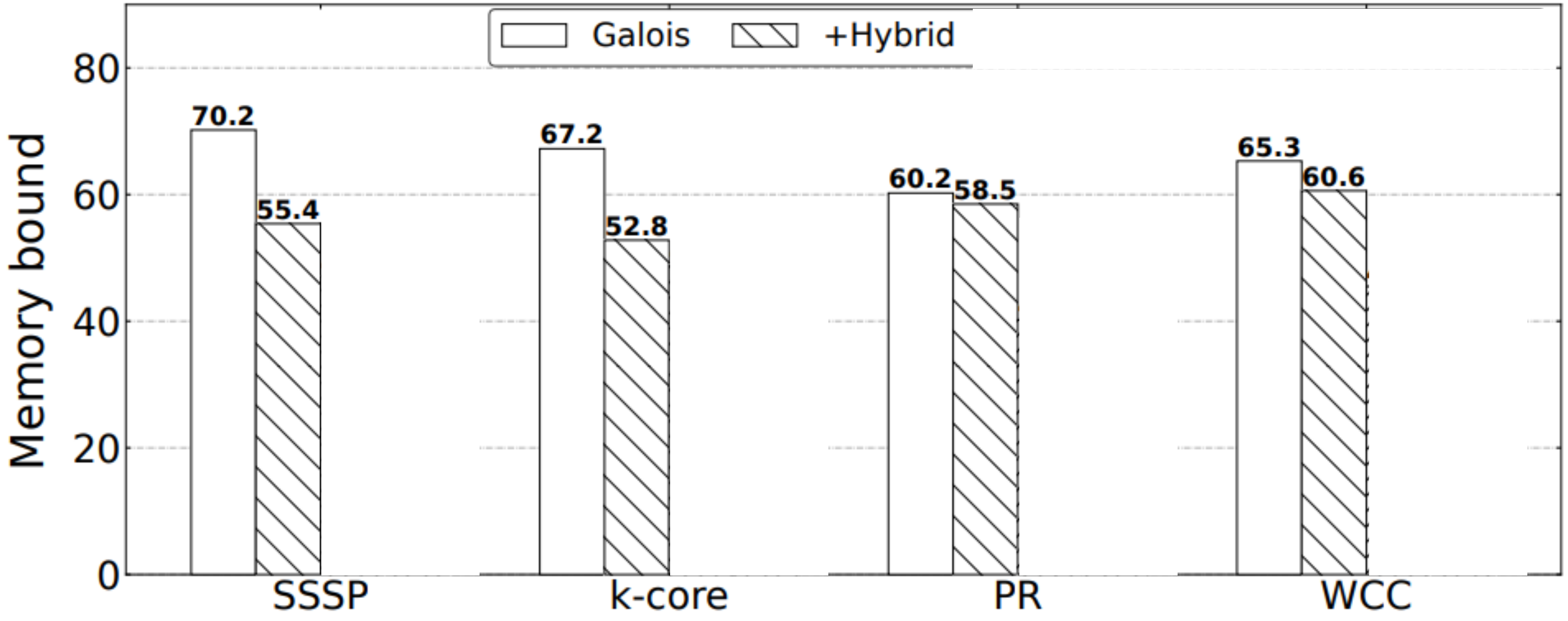
Multi-level queue of *frontiers*
group by task priority
Frontier Queue
P1 Queue
P2 Queue
P3 Queue
Pop highest-priority *frontier* vertex A
Add new *frontiers* to queue
Frontiers
A
D
Scatter
... s(E),E
Frontiers
Frontiers
B
C
D
F
E
F
D
Update values for each vertex
Push message (state of A) into the corresponding buffer for each neighbor
Message
... s(A),D,F
Message
Gather
Choose a *block* to process
Block List
B1 buffer
B2 buffer
B3 buffer
group by block range
Global buffers of messages for each *block*

Thread usage

- A thread pops the top of the frontier queue and conducts scatter for these frontiers
  - Uses a chunk size and local buffers to reduce writes to the global buffers
- Once the top-level queue is empty, the thread switches to gather
  - Receives the global message buffer for a particular block
  - Processes updates for the vertices in that block
    - Single thread on cache-sized data
  - In Async mode, some threads are in each mode at any time
  - In Sync mode, all threads must finish scatter before gather
    - Can use local buffers instead of global buffers

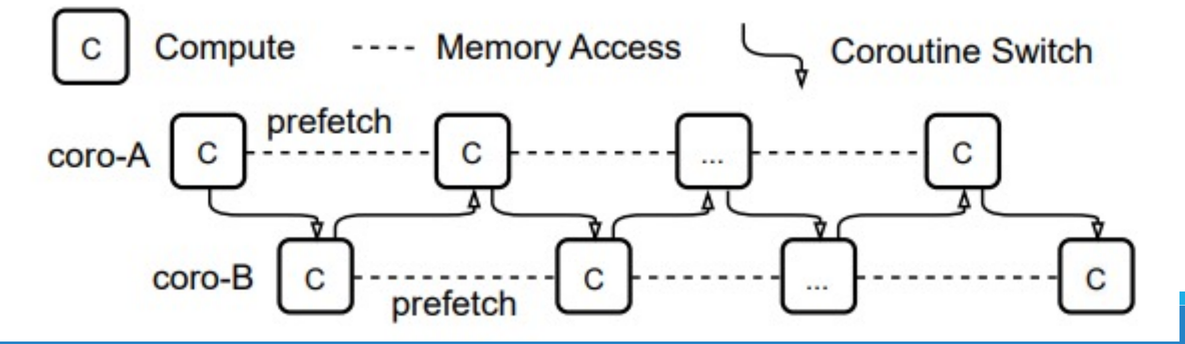
Does this help?

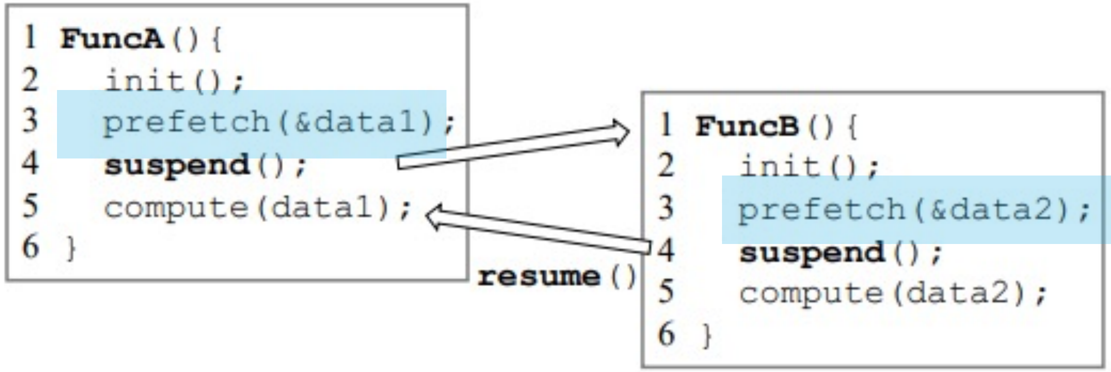
- Specifically, is memory bound improved compared to a vertex-centric framework?
- A: Not that much



Recall: Coroutines

- If we know some information about the workflow, we can do much better at the software level
- *Coroutines are much faster to switch*
  - functions with specific suspend and resume points
  - Retains its local variables in allocated memory
  - Do not require a separate call stack





Improvement: using prefetch and coroutines

- Run two coroutines per thread, where each has a group of vertices
  - Alternate prefetching and computation for the two groups
  - 2 coroutines per thread gives the optimal tradeoff for latency vs switching overhead
- During gather, the cache is “warmed” by loading vertices
  - At some point, we want to stop using prefetch/coroutines
  - Switch after (# of processed vertices) = 1-2x (size of block)
  - At that point, w.h.p. most vertices are in the cache

Improvement: graph data structure

- Store edges of low-degree vertices directly in the offset array
  - Save some lookups/cache misses for the edge array
- Store edges of high-degree nodes separately and refer to a piece of that array
  - Do not need to record all edges in the message buffer
  - These values are distinguished from edge IDs by a bit
- Conversion is linear per graph from CSR format

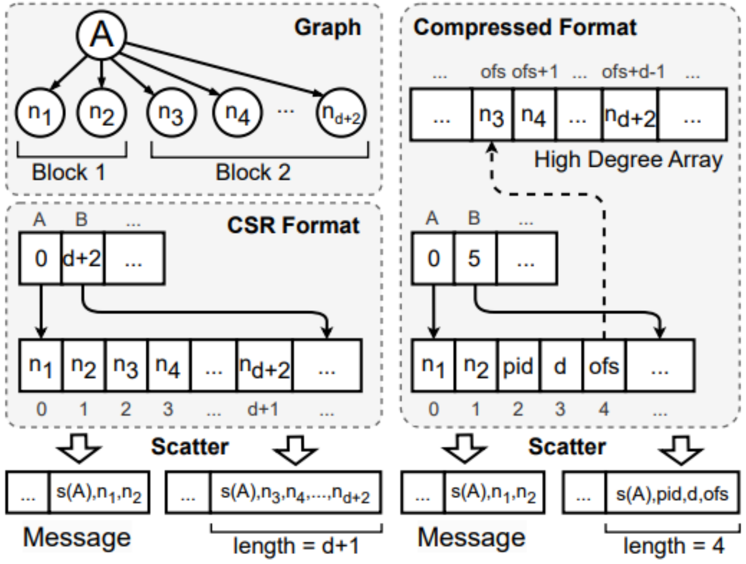
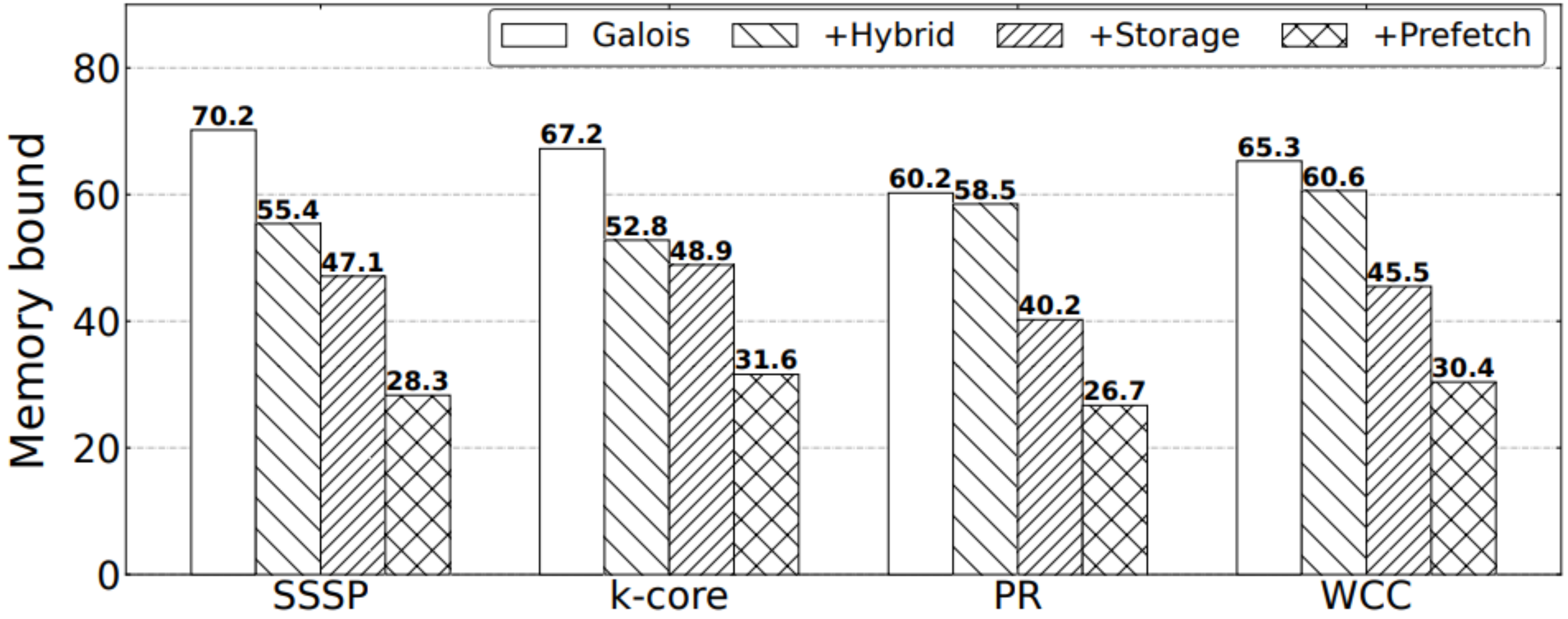


Figure 7: Storing the degree and offset for high degree vertex *A* in block 2 in the cache-friendly graph format.

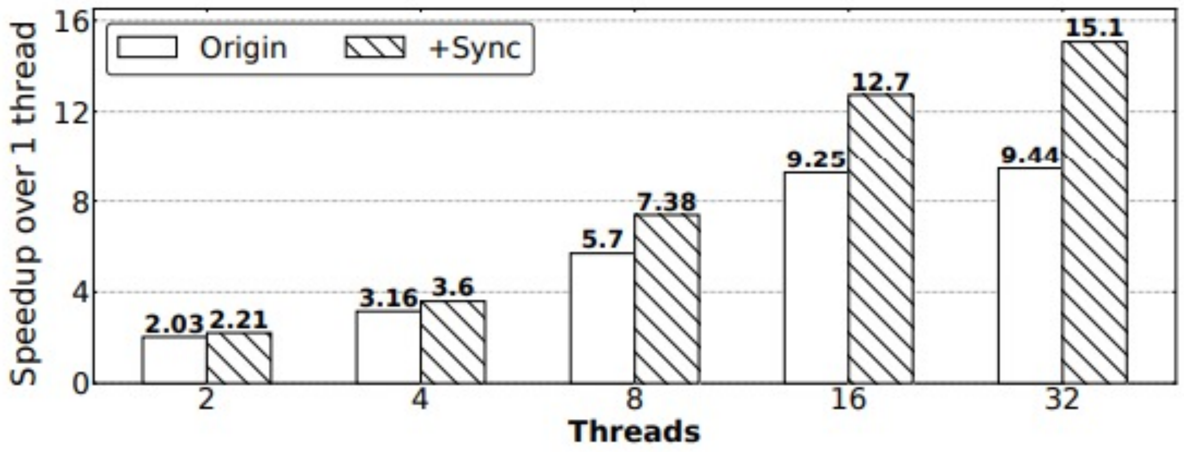
Now does this help?

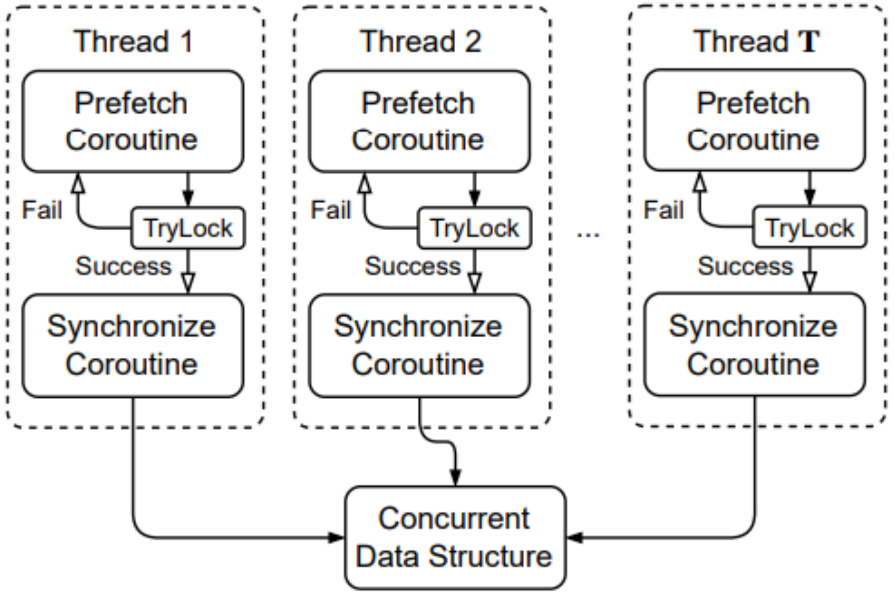
- Specifically, is memory bound improved compared to a vertex-centric framework?
- A: Yes!



Improvement: using prefetch and coroutines

- Synchronization for writes also causes threads to hang or switch
- Coroutines can help with this too…
  - Switch between prefetching data and synchronization





Benchmarking

Parameter tuning

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

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

# of vertices / block. 2^{18} fits in a 1MB L2 cache

(b) Degree threshold for write optimization.

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

Degree threshold to refer to list of edges (2)

(c) Chunk size for task execution.

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

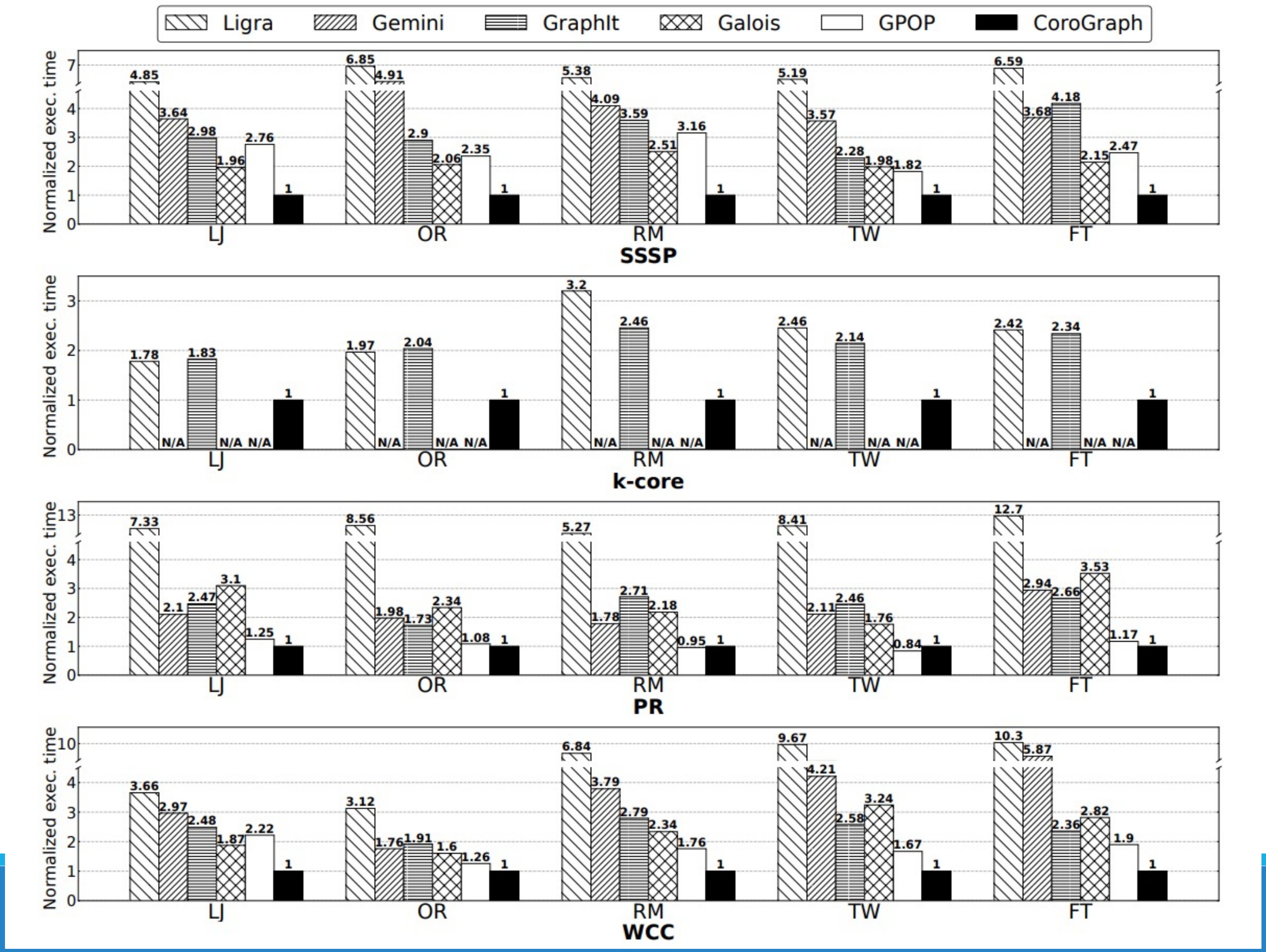
Size of local message buffer

(d) Number of coroutines in the prefetch pipeline.

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

# of coroutines per thread (2)

Is CoroGraph faster in practice?



Other comparisons…

- Is CoroGraph *both* work-efficient and cache-efficient?
- Yes!

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

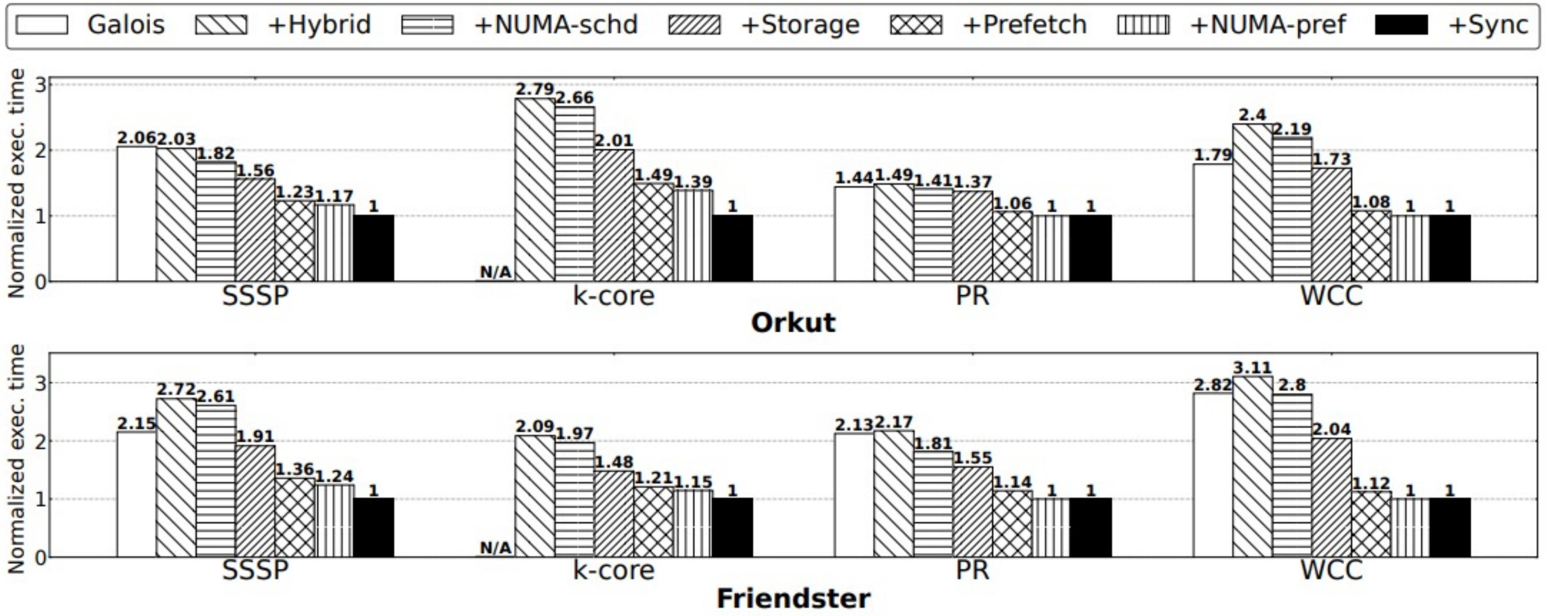
Other comparisons…

- Is CoroGraph parallelized in practice?
- Yes, but with diminishing returns

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

Other comparisons…

- Are all of the optimizations needed?
- Mostly. In fact, most of the improvement comes from them rather than the core algorithm.



Conclusions

- CoroGraph combines aspects of vertex-centric and partition-centric graph frameworks to achieve both work and cache efficiency
- In practice, most of the improvements come from additional optimizations
  - The core algorithm alone does not clearly outperform others
  - Optimizations include prefetching data in the background, improved data structures, and optimizing for multi-socket processors
- Could these optimizations help other frameworks?