GraphChi: Large-Scale Computation on Just a PC

6.886

Joana M. F. da Trindade

Apr 5th 2019

Motivation

Large graphs are everywhere: social networks, web graphs, protein interaction, ...

Cannot be naturally decomposed into smaller parts for parallel processing, so just using MapReduce is inefficient

At the time (2012) some distributed systems tried to address this:
- Specialized graph processing: Pregel, GraphLab
- General: Piccolo and Spark

Born out of frustration with distributed computing: if graph fits into disk, can we perform advanced graph computations on just a personal computer?

Why is it so hard to efficiently use local storage?

This class so far has covered some of the reasons :-)
- Specialized external versions of algorithms needed to use I/O efficiently

Large graphs (multiple TBs) may be sparse and irregular

Irregular -> a vertex can be connected to any other vertex; little locality

Sparse -> graphs with power-law degree distributions have a long tail of nodes with small amount of in/out edges

Sparsity and irregularity -> random access fetching small amounts of data

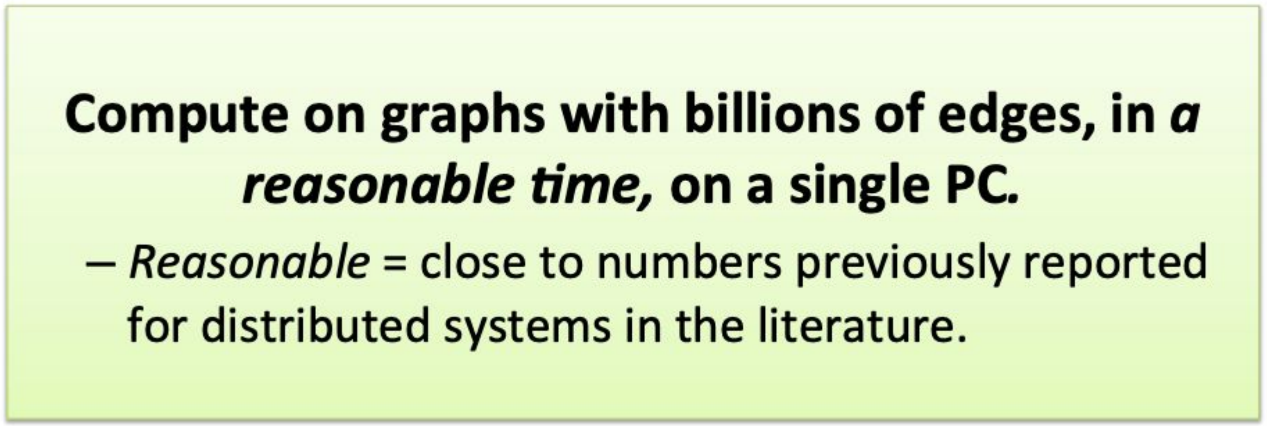
Can’t we just throw more machines at the problem?

Costly
- More engineers required to manage cluster infrastructure
- More machines -> higher total energy consumption

Utilization
- W/o efficient external memory algorithms, no guarantee of better utilization (e.g., load imbalance, idle machines waiting for stragglers)

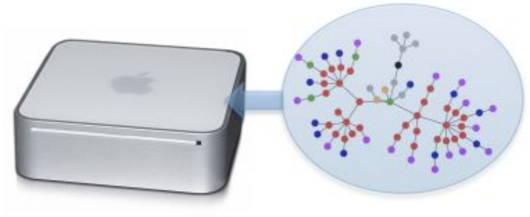
Slower
- Total time = local algo + time for message exchange and cluster coordination

Authors: let’s instead better utilize our single node



Compute on graphs with billions of edges, in *a reasonable time*, on a single PC.

– *Reasonable* = close to numbers previously reported for distributed systems in the literature.

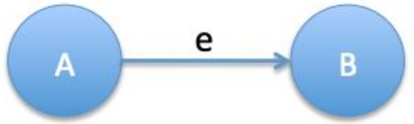


Experiment PC: Mac Mini (2012)

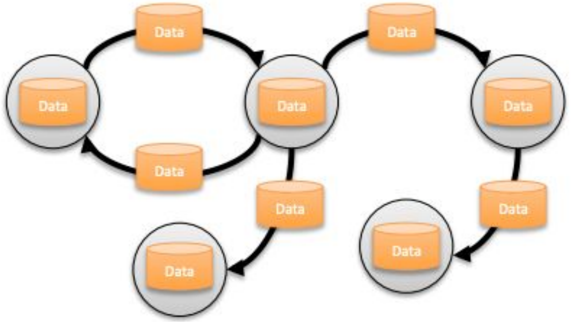
[GraphChi OSDI 2012 slides]

Computational model: vertex-centric

- Graph G = (V, E)
  - **directed edges**: e = (source, destination)
  - each edge and vertex **associated with a value** (user-defined type)
  - vertex and edge **values can be modified**
    - (structure modification also supported)

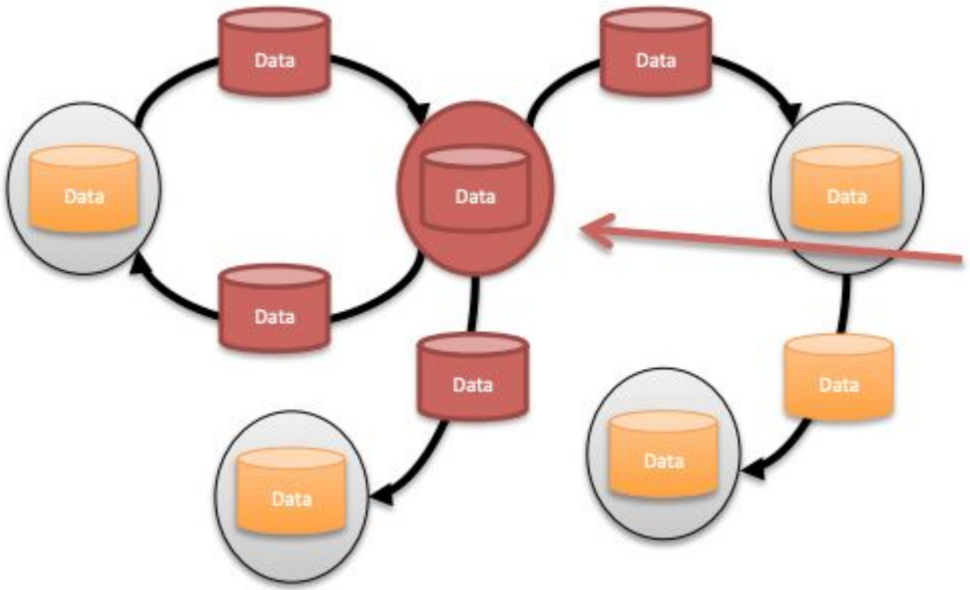


Terms: **e** is an **out-edge** of A, and **in-edge** of B.



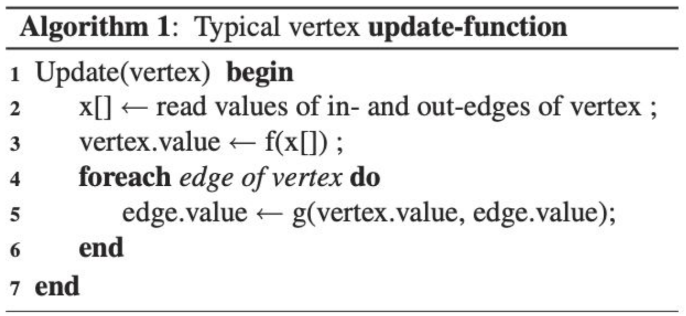
[GraphChi OSDI 2012 slides]

Computational model: vertex-centric



MyFunc(vertex)

{ // modify neighborhood }


Algorithm 1: Typical vertex **update-function**

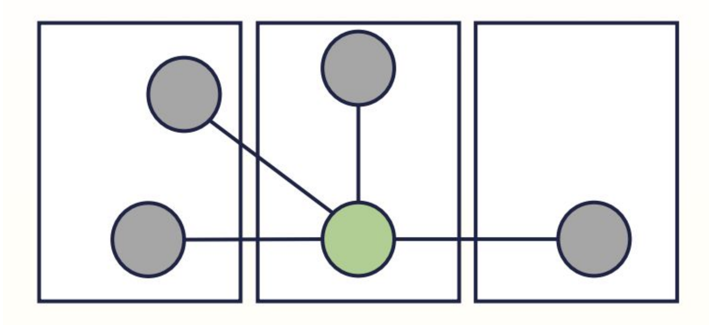
```
Update(vertex) begin
    x[] ← read values of in- and out-edges of vertex ;
    vertex.value ← f(x[]) ;
    foreach edge of vertex do
        edge.value ← g(vertex.value, edge.value);
    end
end
```

[GraphChi OSDI 2012 slides]

Problem: still have random access in vertex-centric

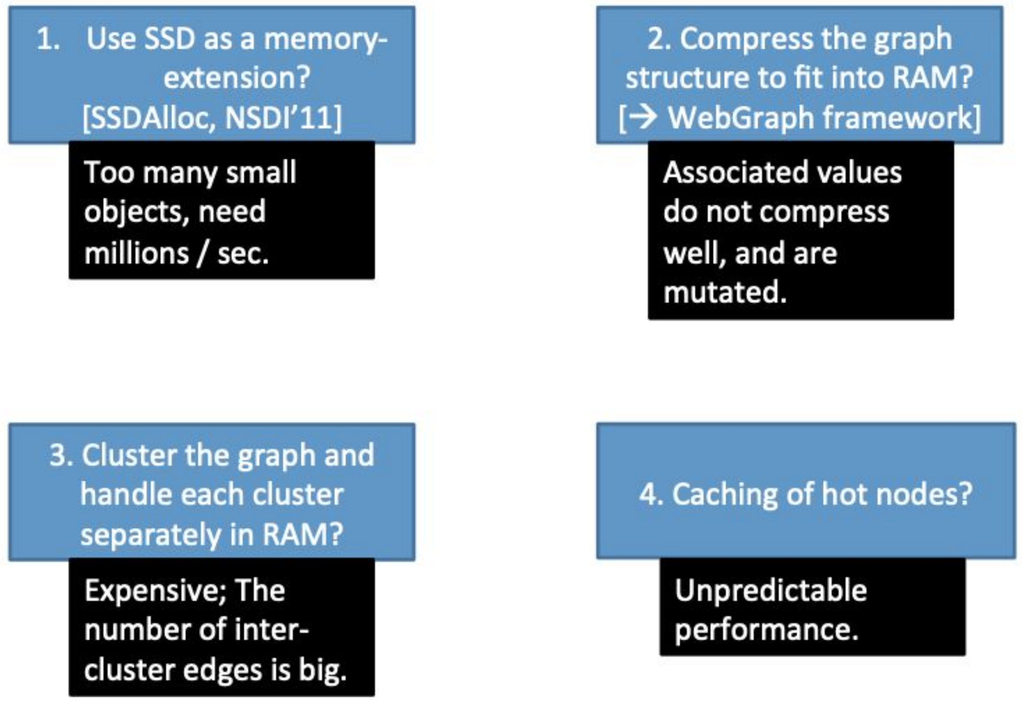
Assumptions:

- Graph is large enough to fit disk, but **not** in memory
- Data for all in/out edges of **any single vertex** fit in memory



Requires random reads/writes across vertex partitions to process in/out edges

Options to avoid random access for vertex-centric



1. Use SSD as a memory-extension? [SSDAlloc, NSDI'11]

Too many small objects, need millions / sec.

2. Compress the graph structure to fit into RAM? [→ WebGraph framework]

Associated values do not compress well, and are mutated.

3. Cluster the graph and handle each cluster separately in RAM?

Expensive; The number of inter-cluster edges is big.

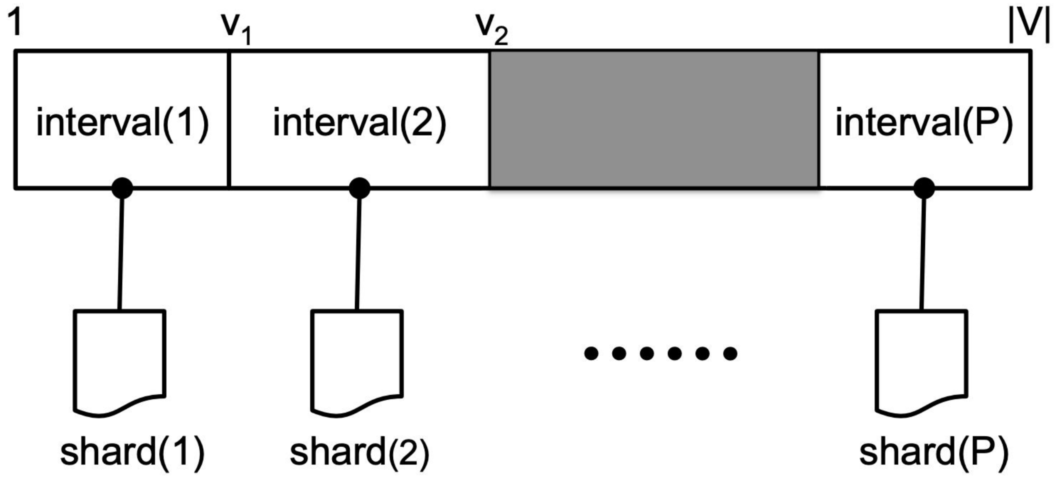
4. Caching of hot nodes?

Unpredictable performance.

[GraphChi OSDI 2012 slides]

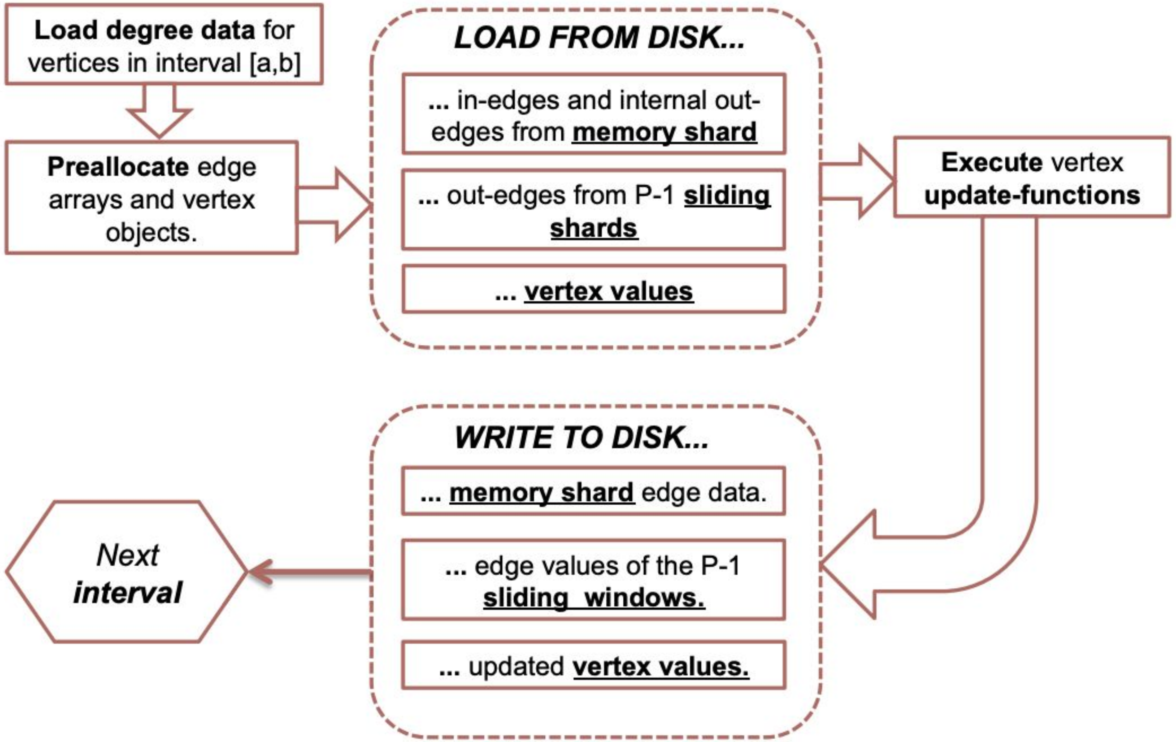
This paper: Parallel Sliding Windows (PSW)

Reduce random access by collocating vertex data with edge data:

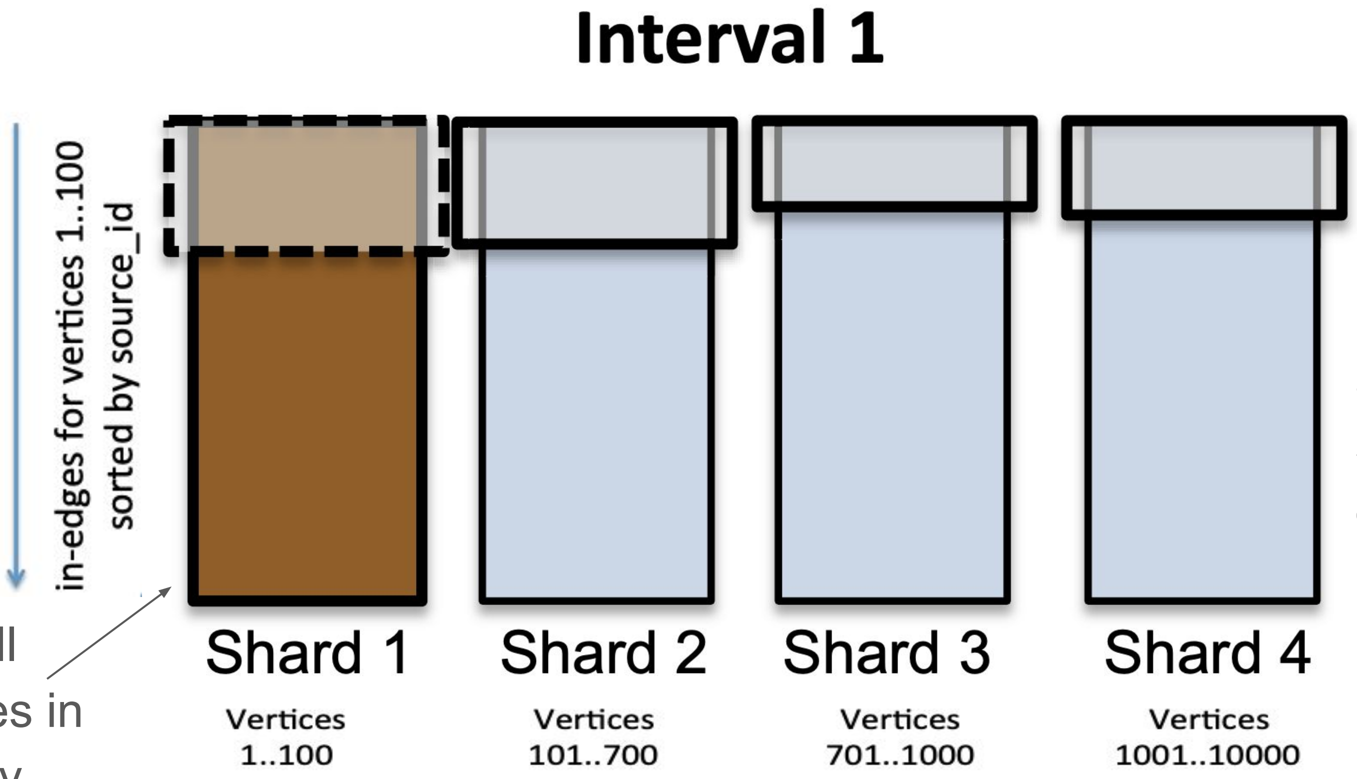


Process one subgraph at a time in 3 stages: (1) load subgraph from disk, (2) update vertices and edges, and (3) write updated values to disk

GraphChi main execution flow


Load degree data for vertices in interval [a,b]
Preallocate edge arrays and vertex objects.
LOAD FROM DISK...
... in-edges and internal out-edges from **memory shard**
... out-edges from P-1 **sliding shards**
... **vertex values**
Execute vertex **update-functions**
WRITE TO DISK...
... **memory shard** edge data.
... edge values of the P-1 **sliding windows.**
... updated **vertex values.**
Next ***interval***

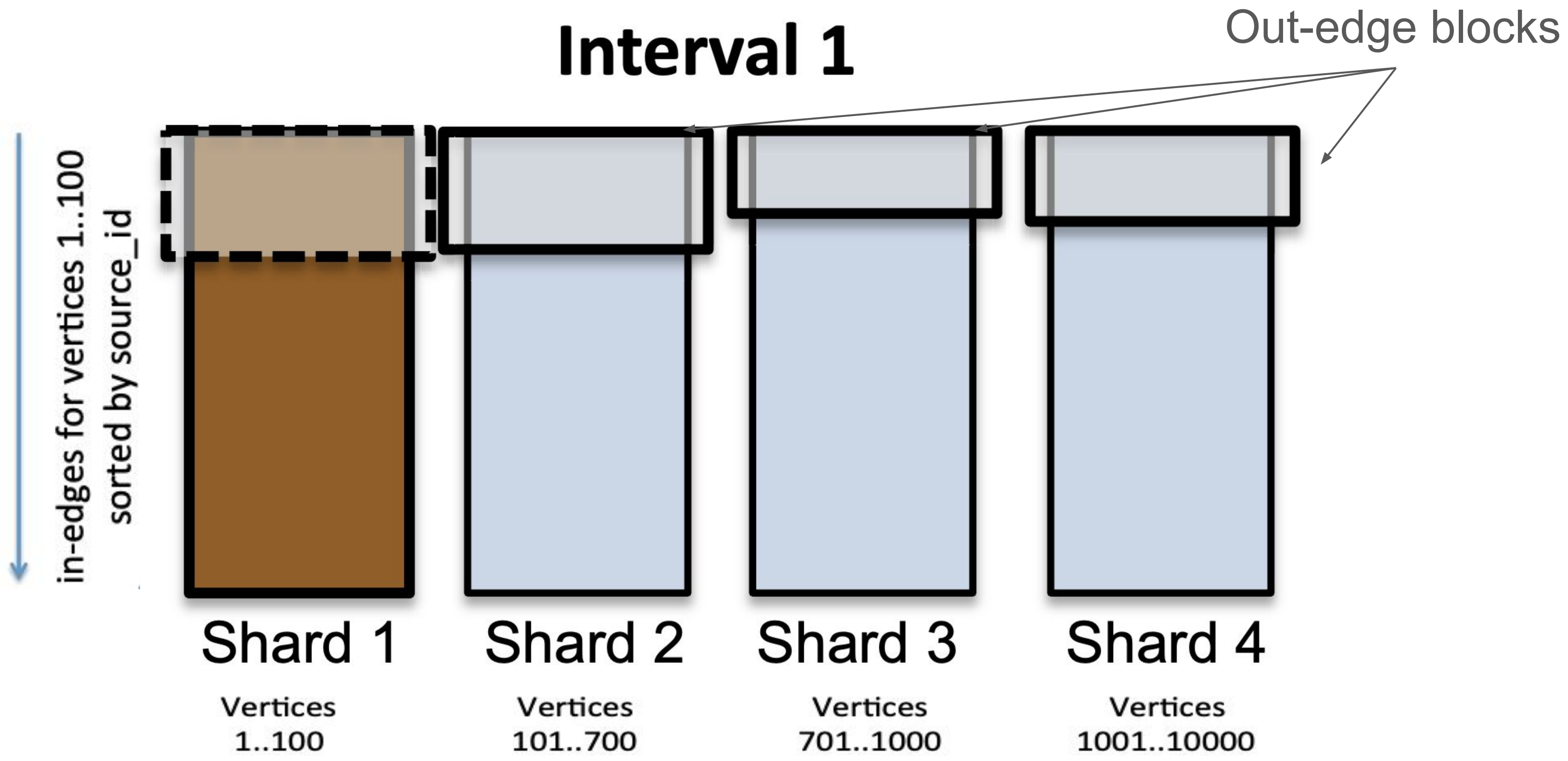
Load stage: example PSW with 4 intervals



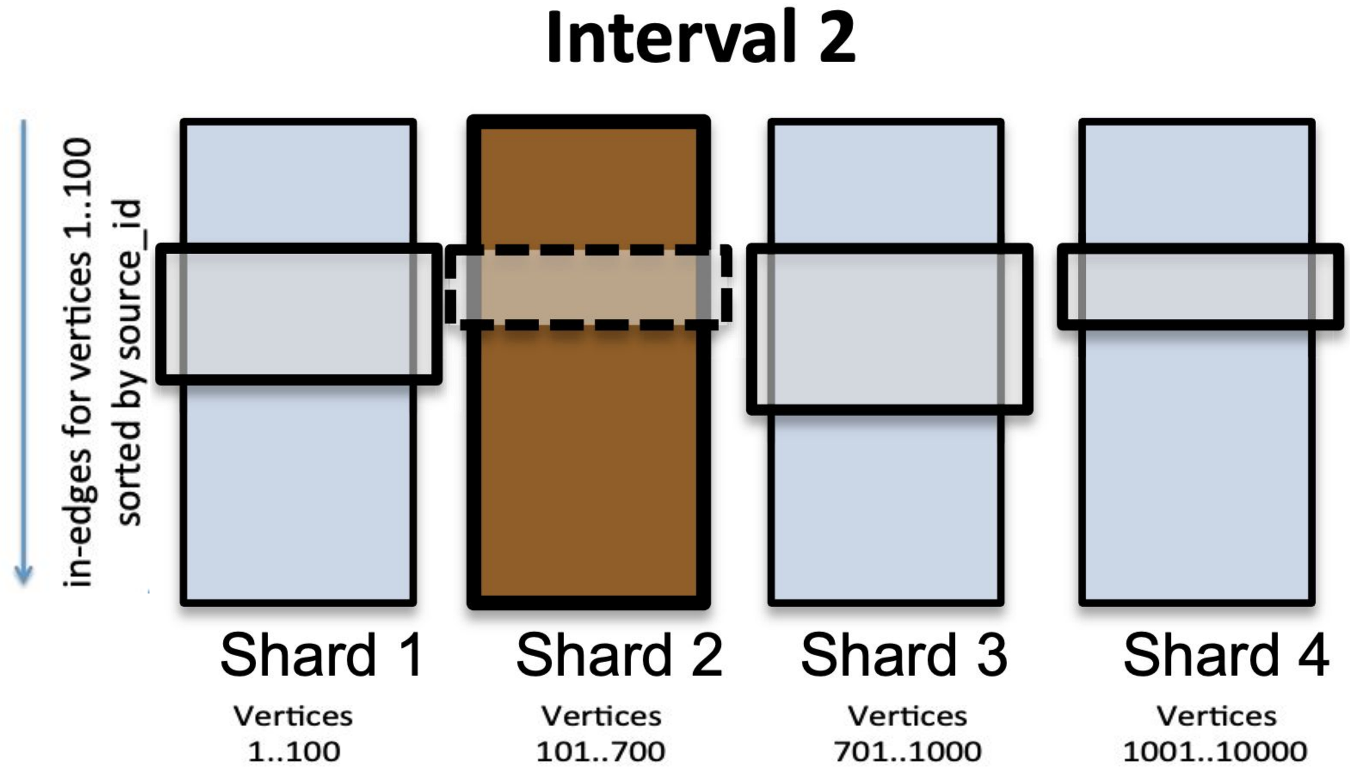
Out-edges arranged in sequence in other shards

Load all in-edges in memory

Load stage: example PSW with 4 intervals



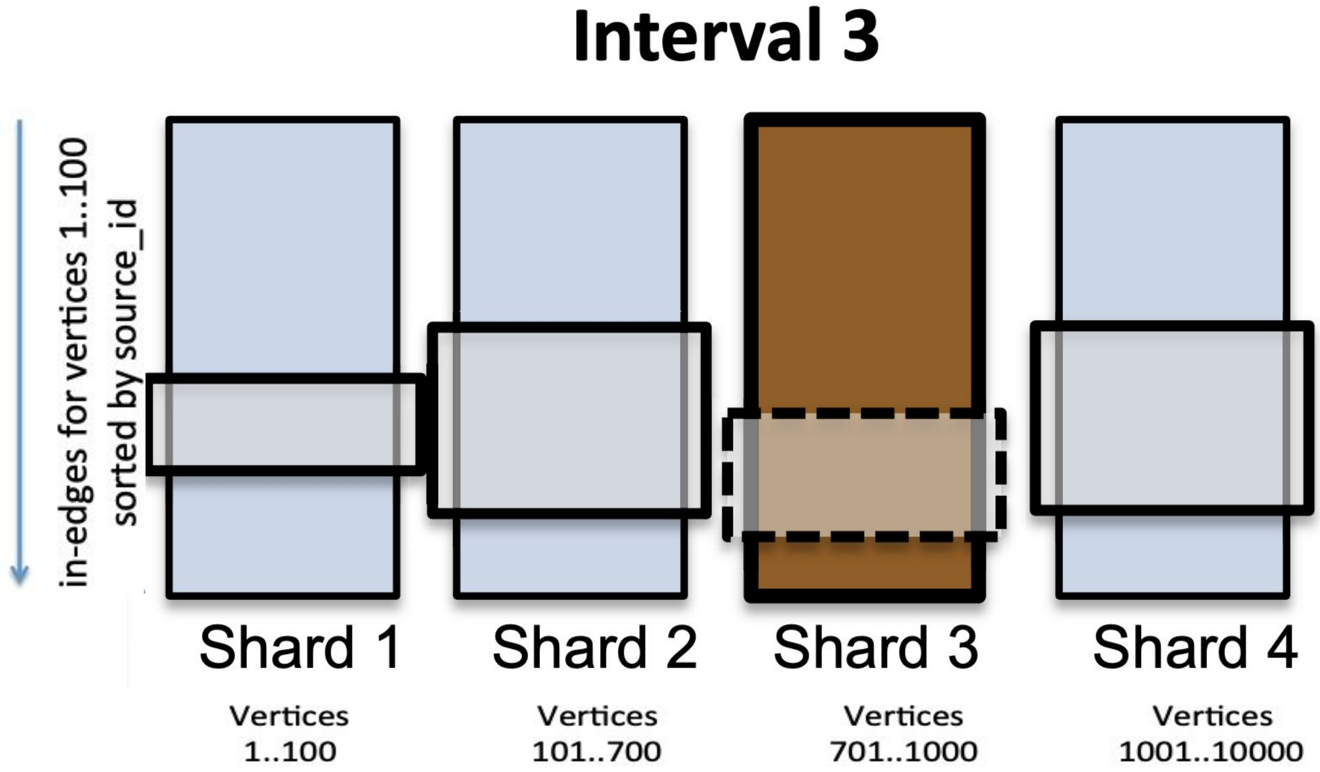
Load stage: example PSW with 4 intervals



Next shard is loaded into memory

Sliding shards move forward to match it

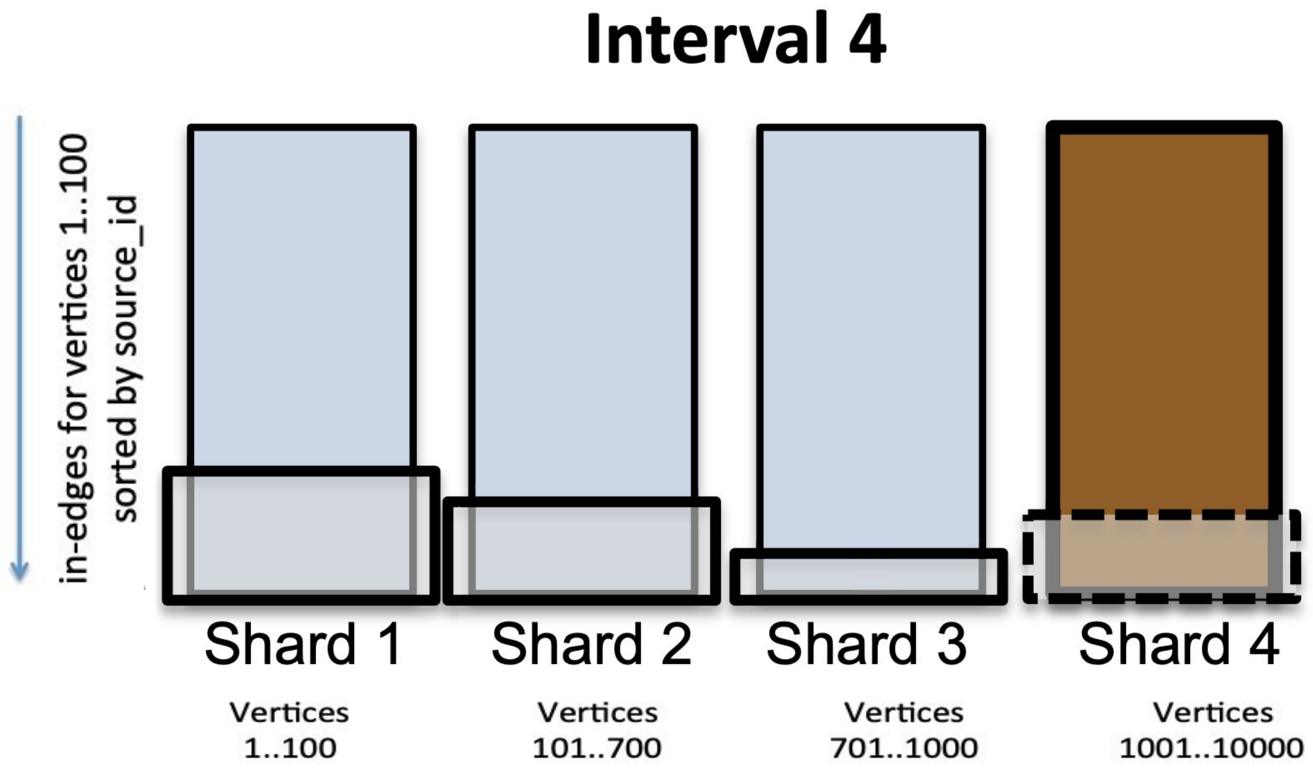
Load stage: example PSW with 4 intervals



Next shard is loaded into memory

Sliding shards move forward to match it

Load stage: example PSW with 4 intervals



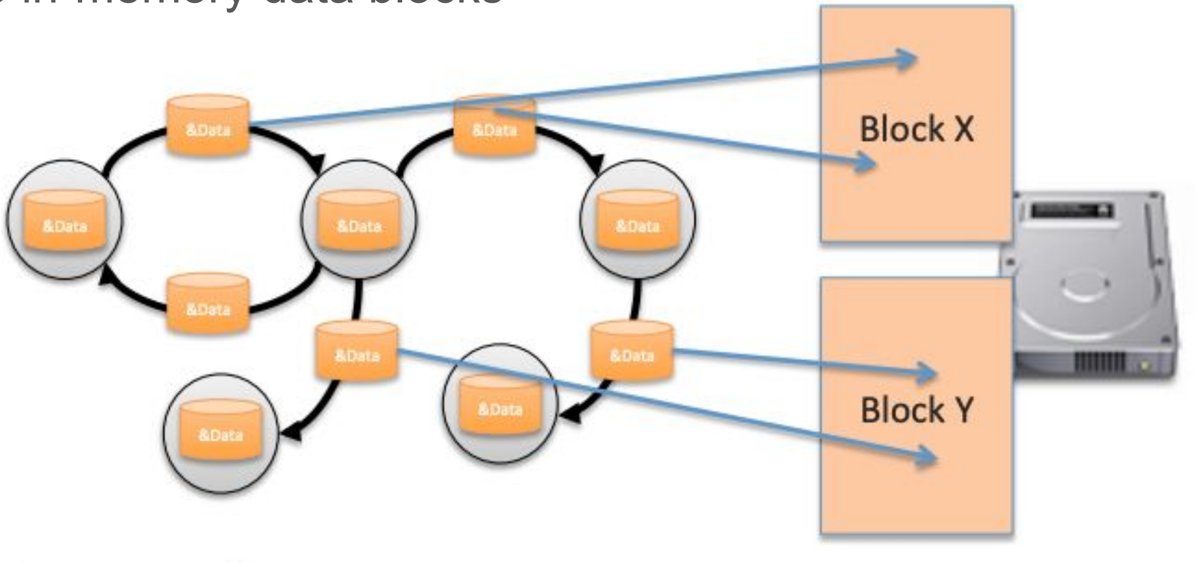
Next shard is loaded into memory

Sliding shards move forward to match it

Update stage

Update-function is executed on vertices in the interval

Edges point to in-memory data blocks

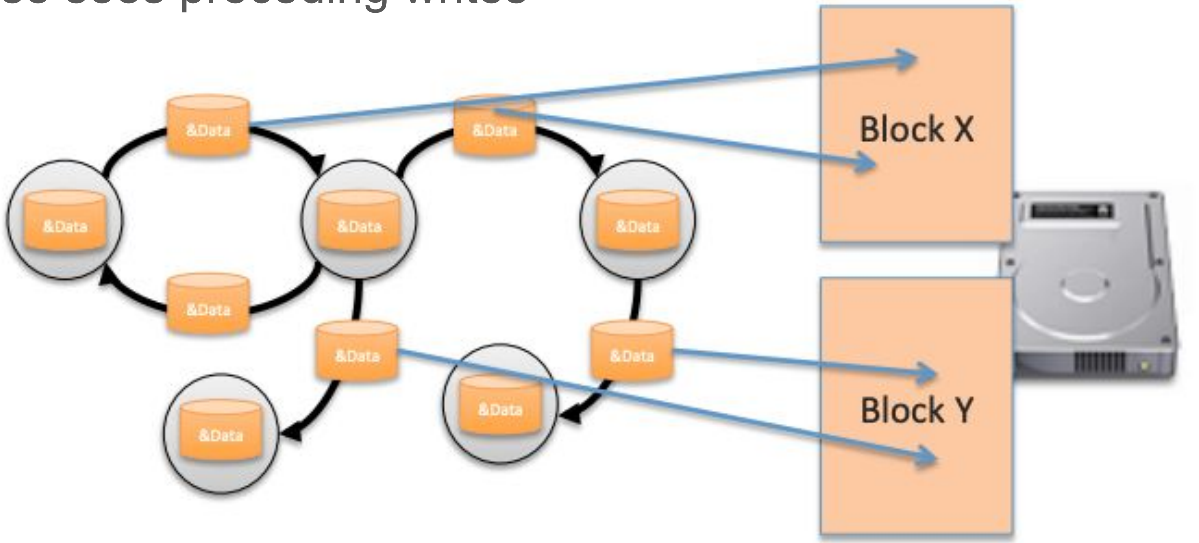


[GraphChi OSDI 2012 slides]

Commit stage

Blocks written back to disk

Next load-phase sees preceding writes



[GraphChi OSDI 2012 slides]

Optimization: dynamic selective scheduling

When active set is sparse (e.g., traversal algos), it is inefficient to process all edges

Their solution: coarse-grained selection:

- Further splits shards into sub-indices
- When neighbors are activated, sets a bit mask
- Loops through bit-mask to figure out which sub-indices to split

Other optimizations in the paper, e.g., buffered updates for evolving graphs

Advantages of PSW

Less random access when compared to other solutions:

- Most reads are performed over sequential chunks

P shards requires only P^2 random reads (across sliding shards)

Each edge in a single graph full scan:

- Only read up to 2 times
- Only written up to 2 times

Drawbacks of PSW

Costly initial pre-processing to ensure sorting order within shards

- E.g., takes 10 min to load twitter 2010 graph (|V| = 42M, |E| = 1.5B)

Storage overhead: vertex value is duplicated

Results with selective scheduling not included in the paper

Does not do well when compared to competing system at the time

- PowerGraph (distributed version of Graphlab), which uses Gather-Apply-Scatter (GAS) for vertex-partitioning better suited for power-law

Evaluation

Comparison against inconsistent system configurations

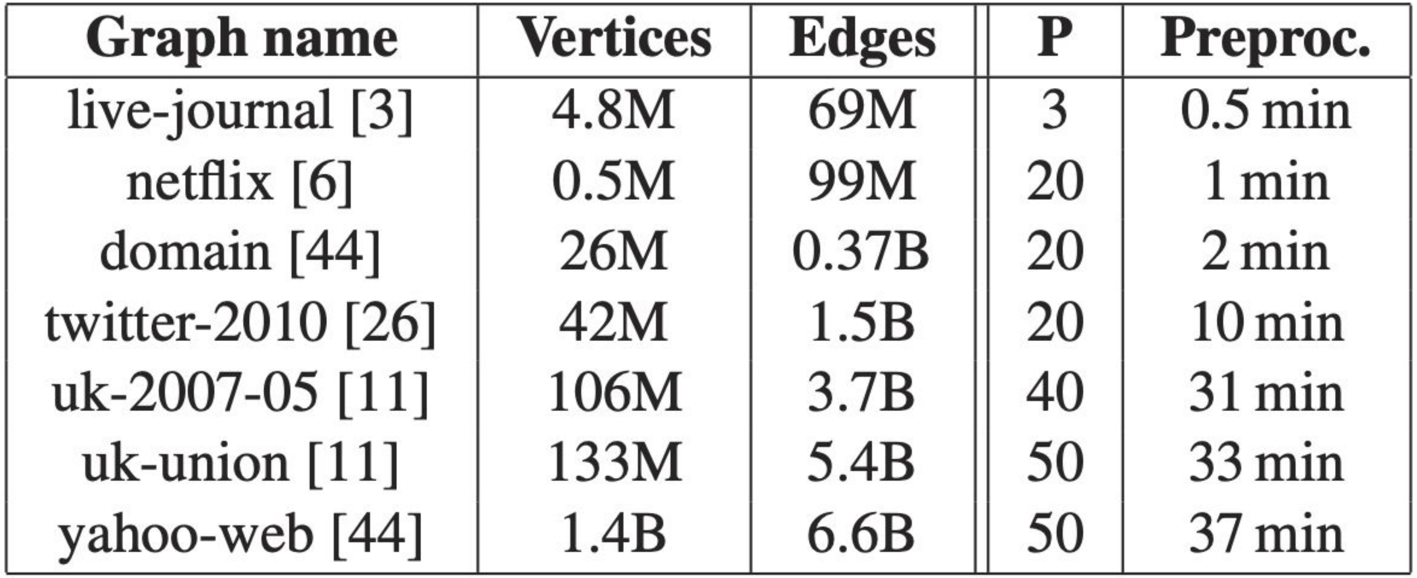
Versus Hadoop-based Pegasus

- Single node GraphChi: 27min; Pegasus w/ 100 machines: 22 min

Versus in-memory systems

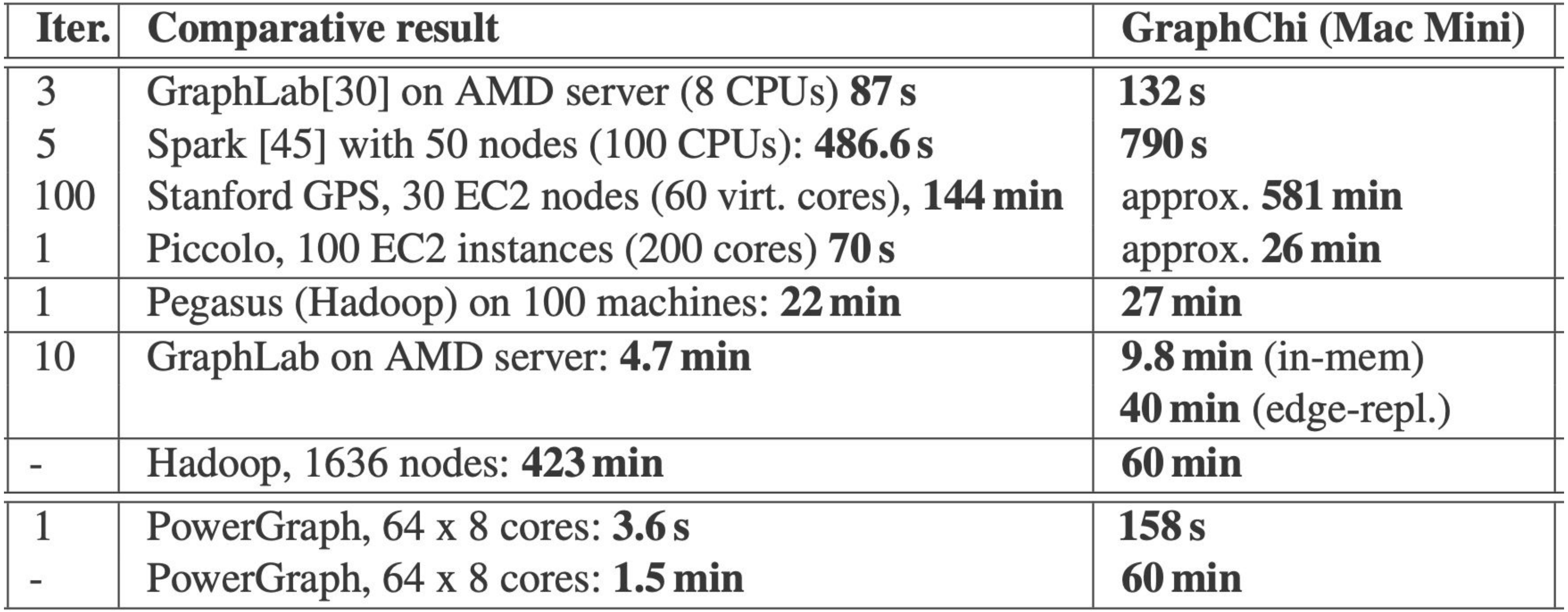
- 2x slower than single-node GraphLab
- 2x slower than 50-node Spark

Evaluation: datasets



Graph name	Vertices	Edges	P	Preproc.
live-journal [3]	4.8M	69M	3	0.5 min
netflix [6]	0.5M	99M	20	1 min
domain [44]	26M	0.37B	20	2 min
twitter-2010 [26]	42M	1.5B	20	10 min
uk-2007-05 [11]	106M	3.7B	40	31 min
uk-union [11]	133M	5.4B	50	33 min
yahoo-web [44]	1.4B	6.6B	50	37 min

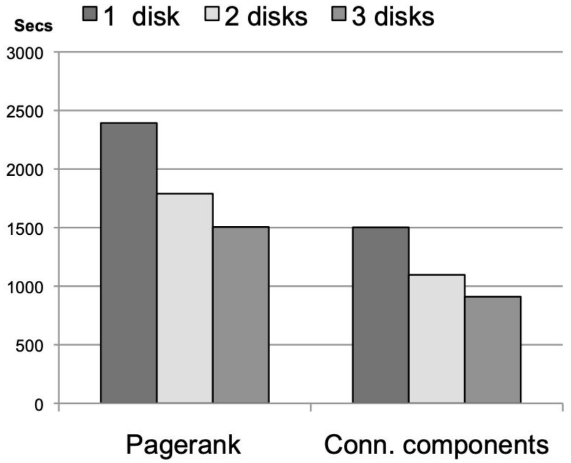
Evaluation: vs other systems (inconsistent configs)

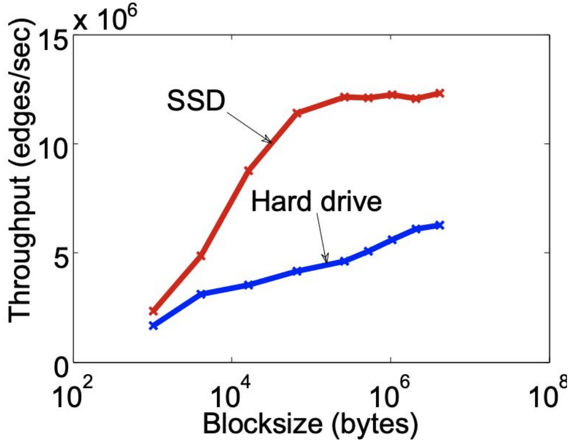


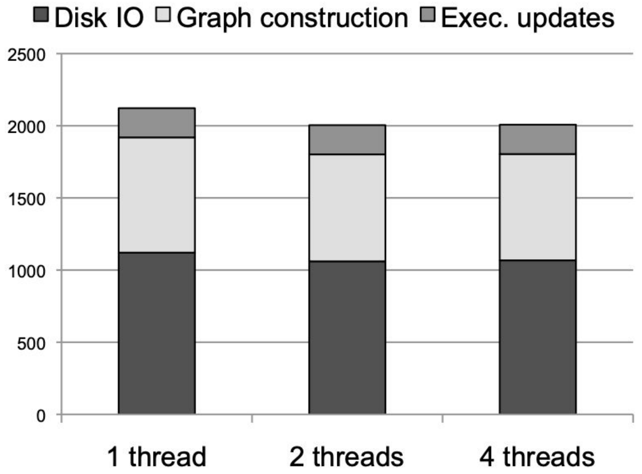
Iter.	Comparative result	GraphChi (Mac Mini)
3	GraphLab[30] on AMD server (8 CPUs) **87 s**	**132 s**
5	Spark [45] with 50 nodes (100 CPUs): **486.6 s**	**790 s**
100	Stanford GPS, 30 EC2 nodes (60 virt. cores), **144 min**	approx. **581 min**
1	Piccolo, 100 EC2 instances (200 cores) **70 s**	approx. **26 min**
1	Pegasus (Hadoop) on 100 machines: **22 min**	**27 min**
10	GraphLab on AMD server: **4.7 min**	**9.8 min** (in-mem)
		40 min (edge-repl.)
-	Hadoop, 1636 nodes: **423 min**	**60 min**
1	PowerGraph, 64 x 8 cores: **3.6 s**	**158 s**
-	PowerGraph, 64 x 8 cores: **1.5 min**	**60 min**

Evaluation: micro-benchmarks

- Performance scales linearly as function of disks
- Little benefits from multithreading when comp. complexity is low (saturates I/O)
- Large benefits when using larger blocks, as less I/O is required







Conclusion

Processing graphs that fit in disk but do not fit in memory is challenging

Authors propose Parallel Sliding Windows (PSW) as a way to reduce random access for graph computation

Core contribution: on-disk graph partitioning targeted for vertex-centric model that minimizes random reads / writes

Results were encouraging: beats distributed systems with 10s of nodes

However, distributed systems with alternative partitioning can still do better (e.g., PowerGraph using GAS for vertex partitioning)