

GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrölä (CMU)

Guy Blelloch (CMU)

Carlos Guestrin (UW)

*In co-operation with the
GraphLab team.*

BigData with *Structure*: BigGraph



social graph



social graph



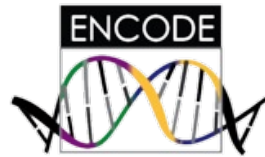
follow-graph



consumer-
products graph



user-movie
ratings
graph



DNA
interaction
graph



WWW
link graph

etc.

Big Graphs != Big Data

Data size:



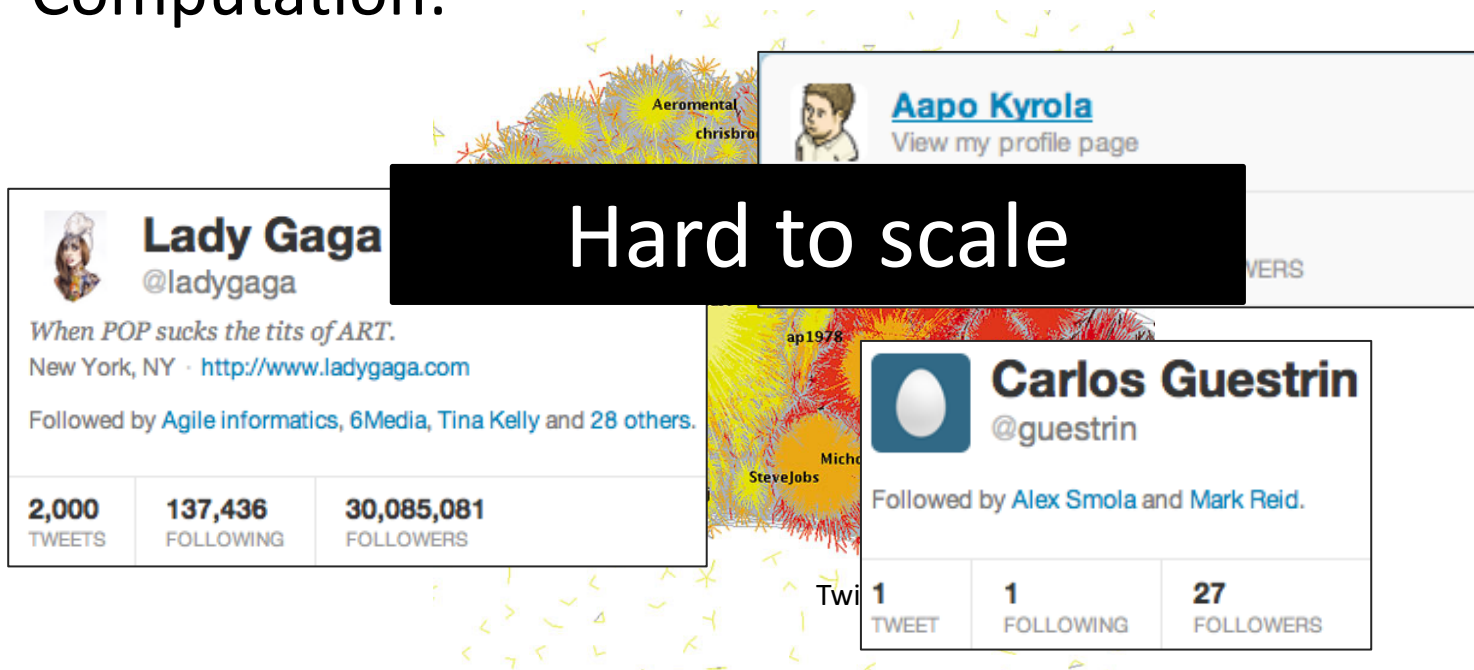
facebook

140 billion
connections

≈ 1 TB

Not a problem!

Computation:



Could we compute **Big Graphs** on a *single machine*?

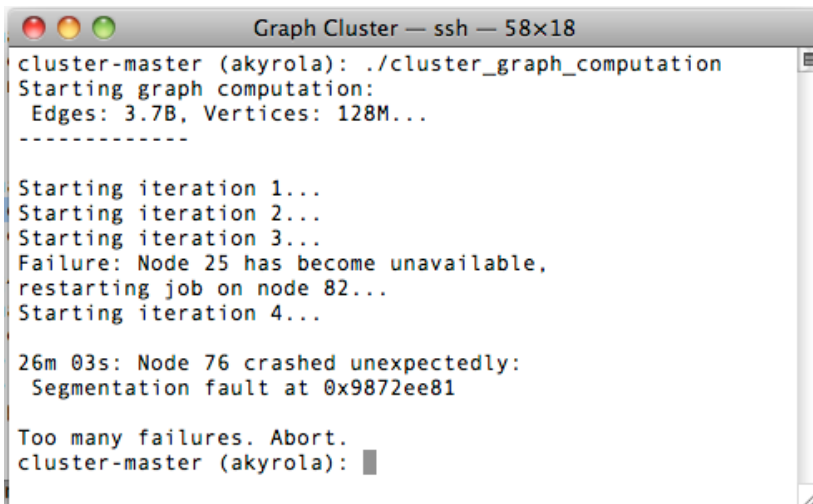
Disk-based Graph Computation



Can't we just use the
Cloud?

Distributed State is Hard to Program

Writing distributed applications remains cumbersome.

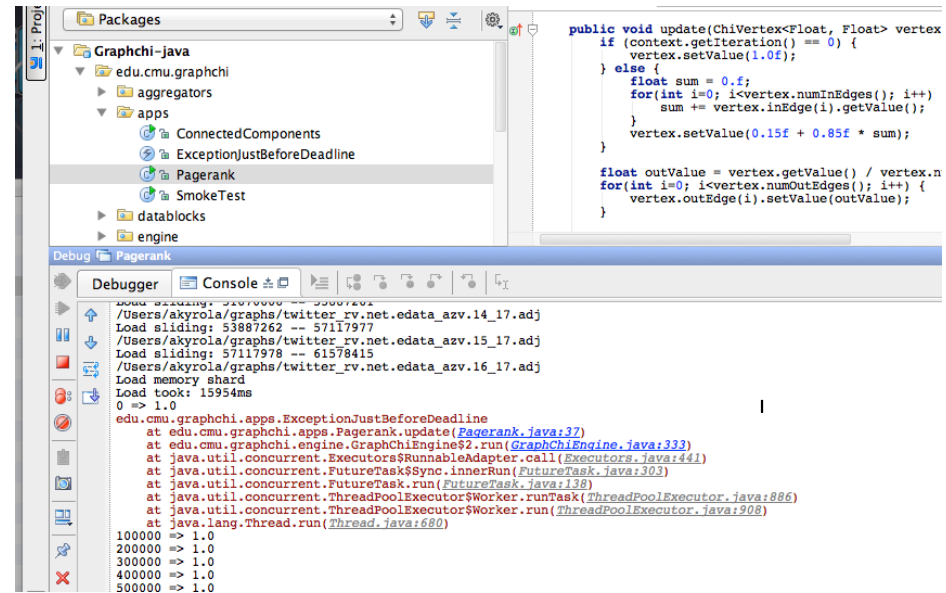


```
Graph Cluster — ssh — 58x18
cluster-master (akyrola): ./cluster_graph_computation
Starting graph computation:
Edges: 3.7B, Vertices: 128M...
-----
Starting iteration 1...
Starting iteration 2...
Starting iteration 3...
Failure: Node 25 has become unavailable,
restarting job on node 82...
Starting iteration 4...

26m 03s: Node 76 crashed unexpectedly:
Segmentation fault at 0x9872ee81

Too many failures. Abort.
cluster-master (akyrola): █
```

Cluster crash



```

Packages
└─ Graphchi-java
   └─ edu.cmu.graphchi
      └─ aggregators
         └─ apps
            ├── ConnectedComponents
            ├── ExceptionJustBeforeDeadline
            └─ Pagerank
               ├── datablocks
               └─ engine

public void update(ChiVertex<Float, Float> vertex
if (context.getIteration() == 0) {
    vertex.setValue(1.0f);
} else {
    float sum = 0.f;
    for(int i=0; i<vertex.numInEdges(); i++)
        sum += vertex.inEdge(i).getValue();
    vertex.setValue(0.15f + 0.85f * sum);
}

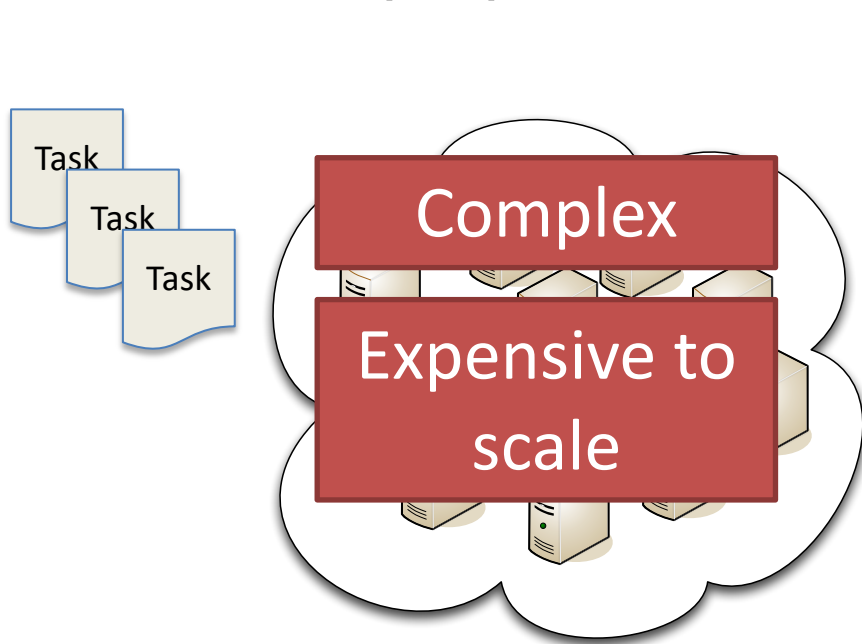
float outValue = vertex.getValue() / vertex.n
for(int i=0; i<vertex.numOutEdges(); i++) {
    vertex.outEdge(i).setValue(outValue);
}

Debugger
edu.cmu.graphchi.apps.ExceptionJustBeforeDeadline
at edu.cmu.graphchi.apps.Pagerank.update(Pagerank.java:37)
at edu.cmu.graphchi.engine.GraphChiEngine$2.run(GraphChiEngine.java:333)
at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:441)
at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:303)
at java.util.concurrent.FutureTask.run(FutureTask.java:138)
at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
at java.lang.Thread.run(Thread.java:680)
100000 => 1.0
200000 => 1.0
300000 => 1.0
400000 => 1.0
500000 => 1.0
```

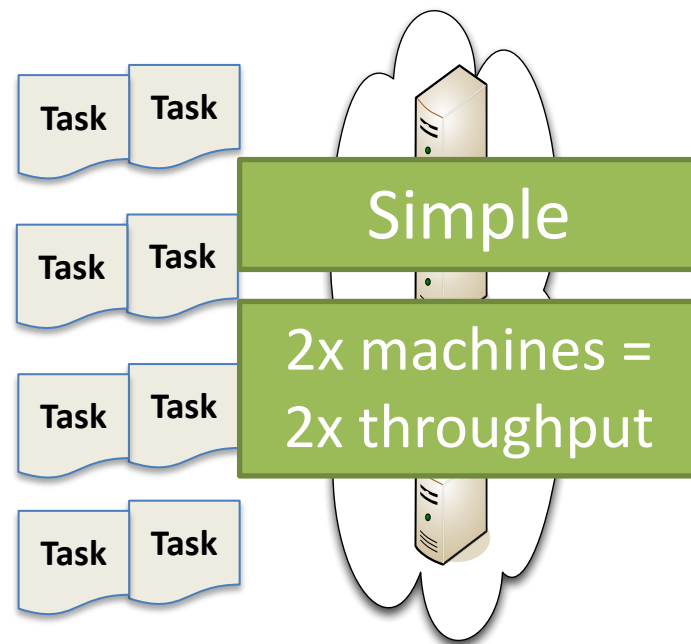
Crash in your IDE

Efficient Scaling

- Businesses need to compute hundreds of distinct tasks on the same graph
 - Example: personalized recommendations.



Parallelize each task

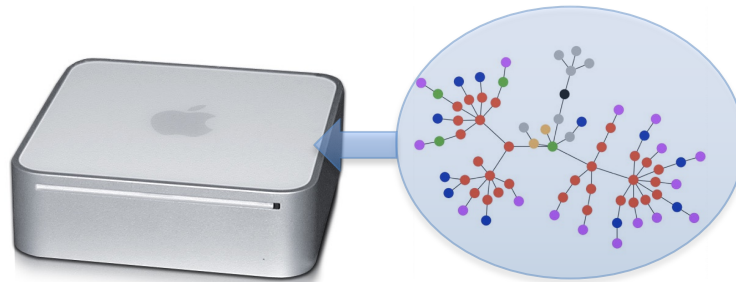


Parallelize across tasks

Research Goal

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.

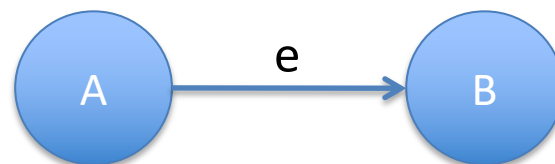


Experiment PC: Mac Mini (2012)

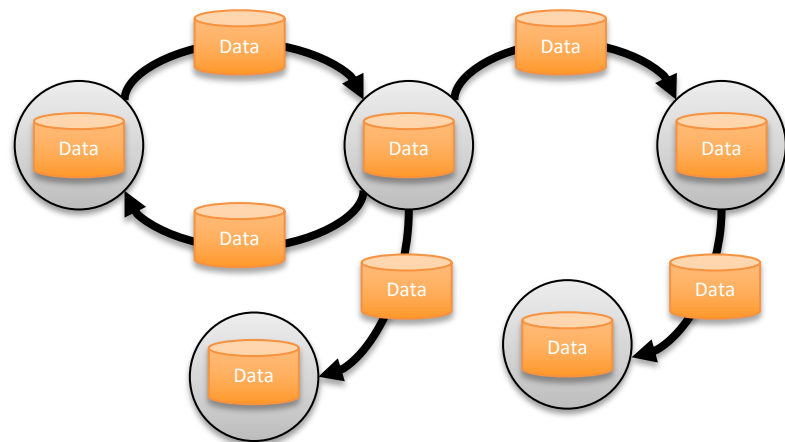
Computational Model

Computational Model

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

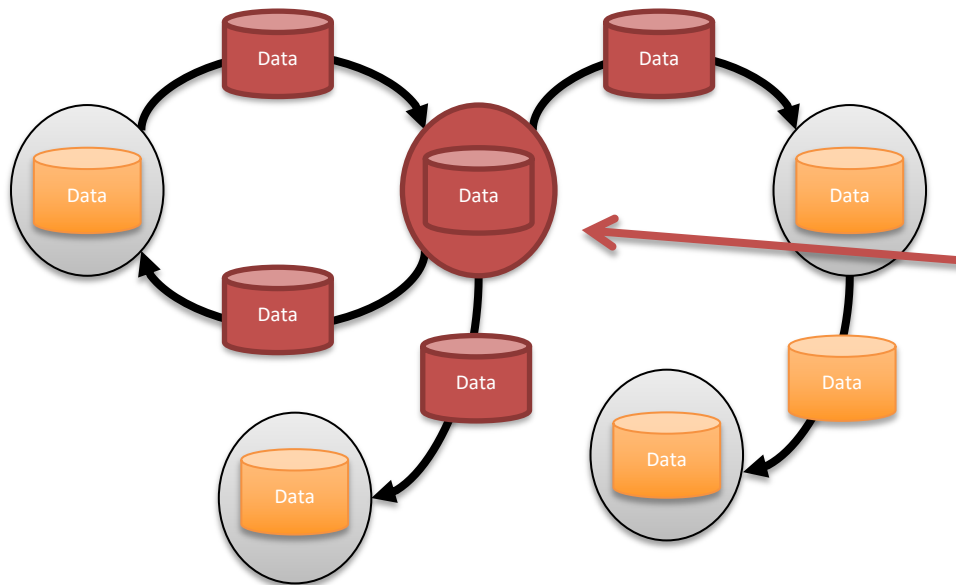


Terms: **e** is an **out-edge** of A, and **in-edge** of B.



Vertex-centric Programming

- “Think like a vertex”
- Popularized by the Pregel and GraphLab projects
 - Historically, systolic computation and the Connection Machine

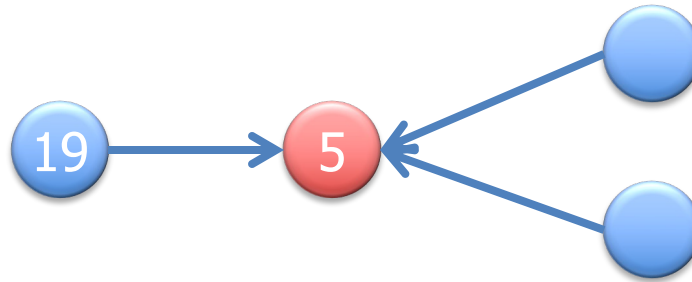


MyFunc(vertex)
{ // modify neighborhood }

The Main Challenge of Disk-based Graph Computation:

Random Access

Random Access Problem



- Symmetrized adjacency file with values,

vertex	in-neighbors	out-neighbors
5	3:2.3, 19: 1.3 , 49: 0.65	781: 2.3, 881: 4.2..
....		
19	3: 1	2, ...

Random
write

- ... or with for SSD, this is too much.

vertex	in-neighbor-ptr	out-neighbors
5	3: <u>881</u> , 19: <u>10092</u> , 49: <u>20763</u> ,...	781: 2.3, 881: 4.2..
....		
19	3: <u>882</u> , 9: <u>2872</u> , ...	5: 1.3 , 28: 2.2, ...

read

Random
read

Possible Solutions

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?

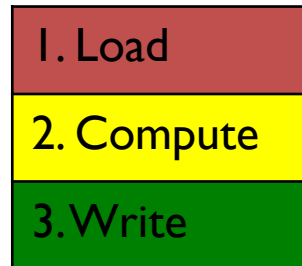
Unpredictable performance.

Our Solution

Parallel Sliding Windows (PSW)

Parallel Sliding Windows: Phases

- PSW processes the graph one **sub-graph** a time:



- In one **iteration**, the whole graph is processed.
 - And typically, next iteration is started.

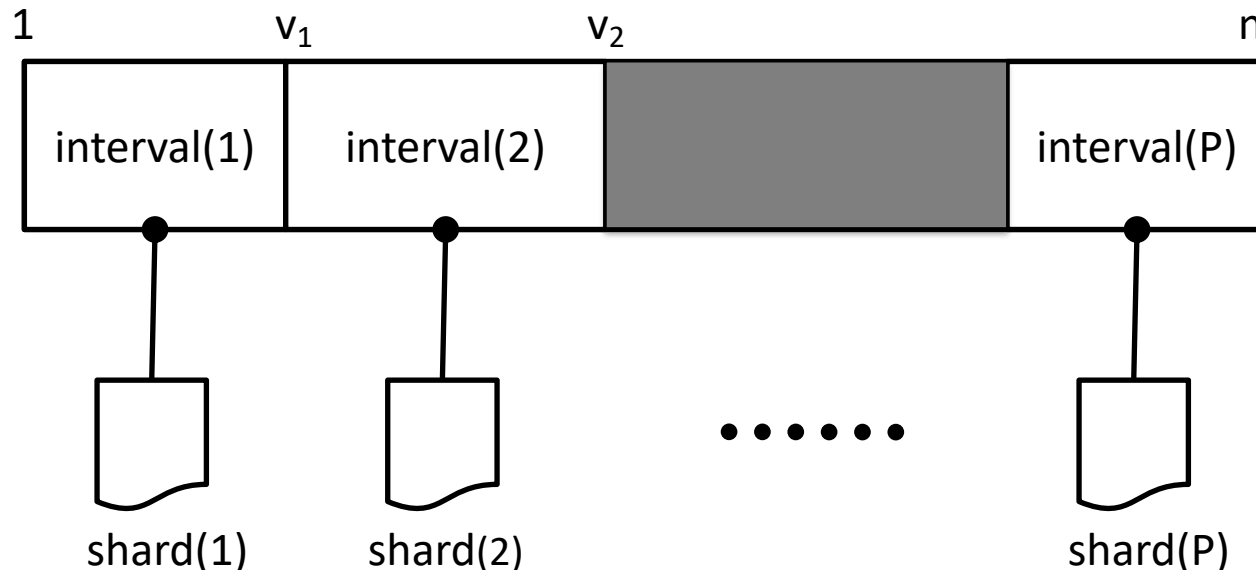
1. Load

2. Compute

3. Write

PSW: Shards and Intervals

- Vertices are numbered from 1 to n
 - P intervals, each associated with a **shard** on disk.
 - **sub-graph** = interval of vertices



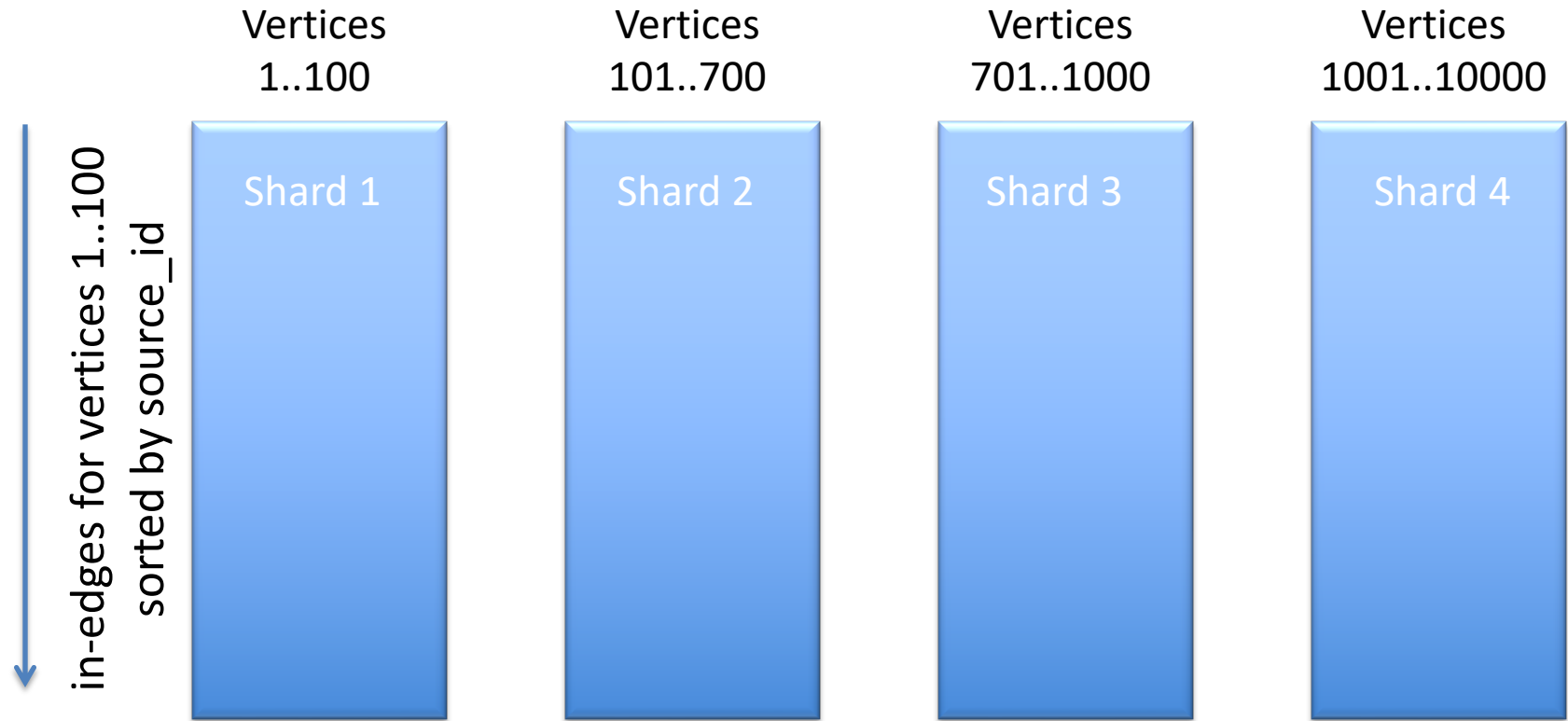
PSW: Layout

1. Load

2. Compute

3. Write

Shard: in-edges for **interval** of vertices; sorted by source-id



Shards small enough to fit in memory; balance size of shards

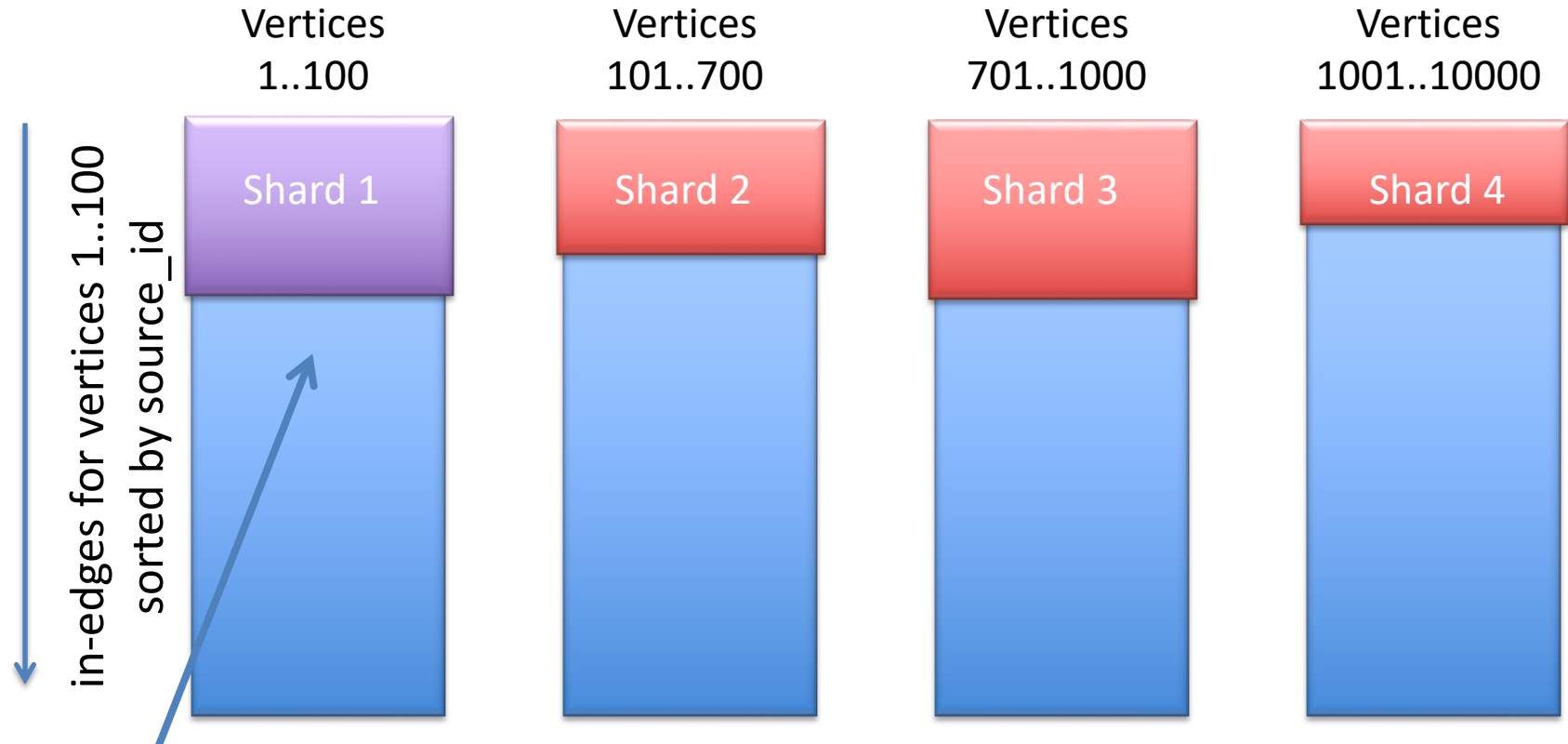
PSW: Loading Sub-graph

1. Load

2. Compute

3. Write

Load subgraph for vertices 1..100



Load all in-edges
in memory

What about out-edges?
Arranged in sequence in other shards

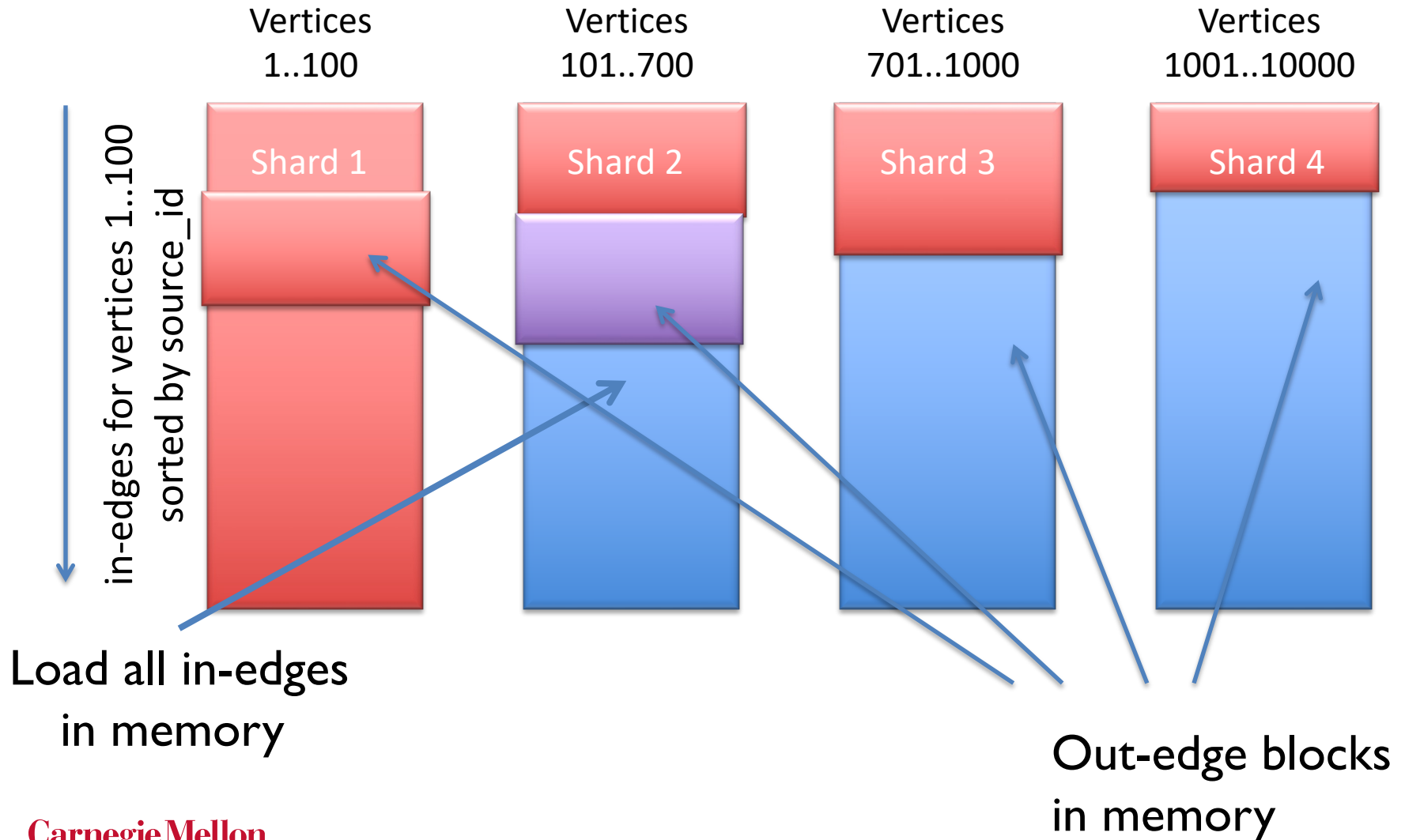
PSW: Loading Sub-graph

Load subgraph for vertices 101..700

1. Load

2. Compute

3. Write



PSW Load-Phase

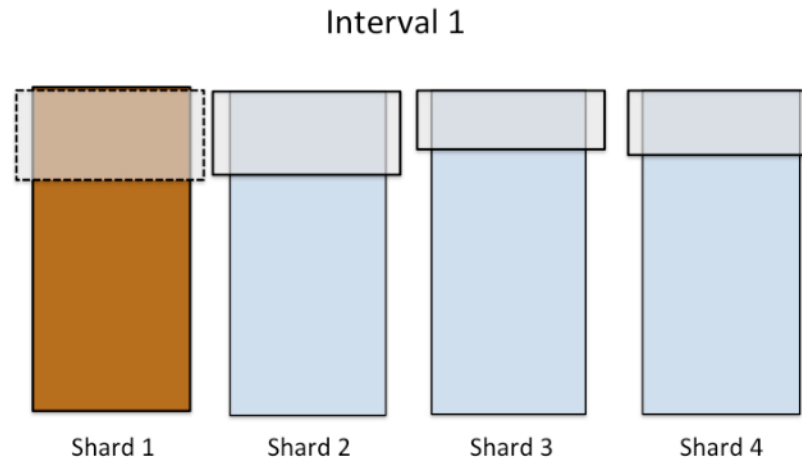
1. Load

2. Compute

3. Write

Only P large reads for each interval.

P^2 reads on one full pass.



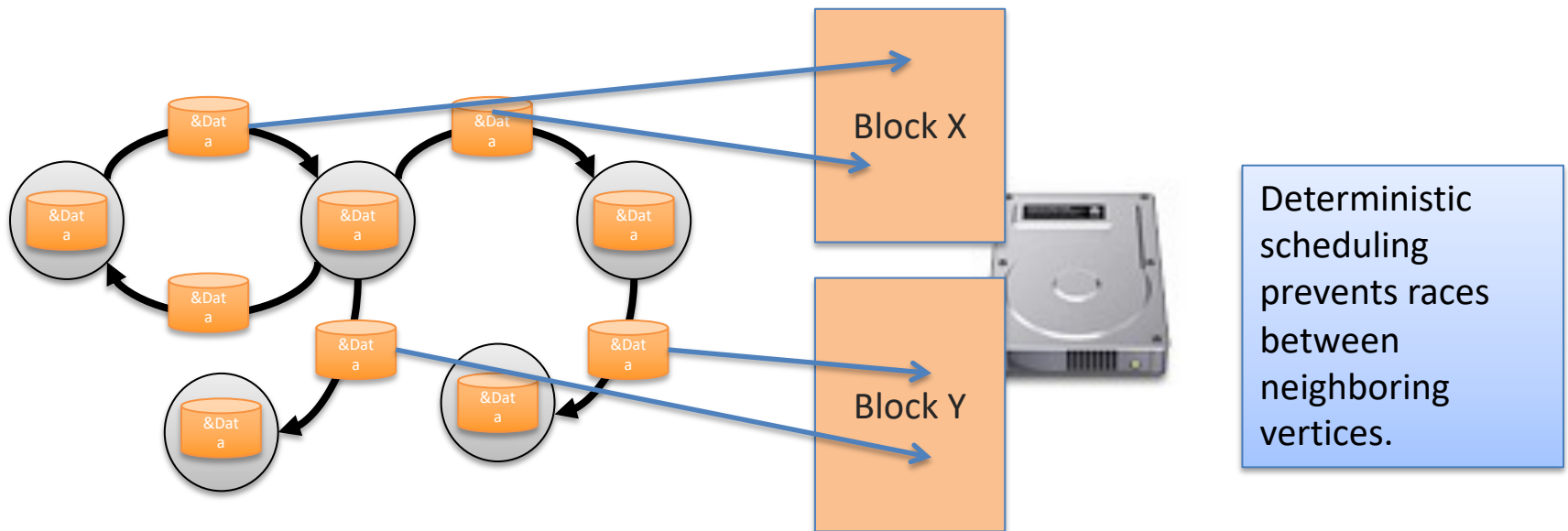
1. Load

2. Compute

3. Write

PSW: Execute updates

- Update-function is executed on interval's vertices
- Edges have pointers to the loaded data blocks
 - Changes take effect immediately → asynchronous.

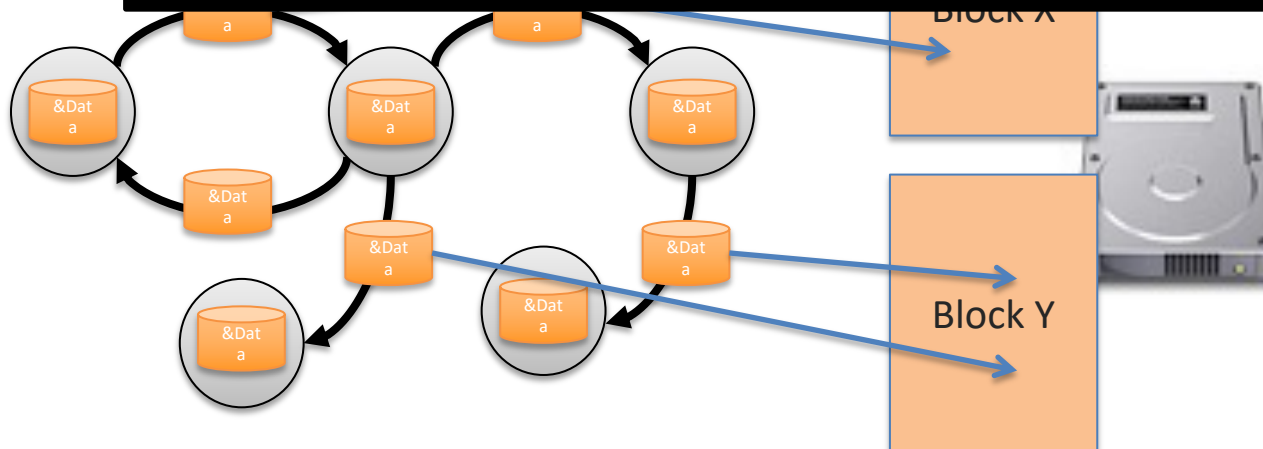


1. Load
2. Compute
3. Write

PSW: Commit to Disk

- In write phase, the blocks are written *back* to disk
 - Next load-phase sees the preceding writes → asynchronous.

In total:
 P^2 reads and writes / full pass on the graph.
→ Performs well on *both* SSD and hard drive.



GraphChi: Implementation

Evaluation & Experiments

GraphChi

- C++ implementation: 8,000 lines of code
 - Java-implementation also available (~ 2 -3x slower), with a Scala API.
- Several optimizations to PSW (see paper).

Source code and
examples:

<http://graphchi.org>

EVALUATION: APPLICABILITY

Evaluation: Is PSW expressive enough?

Graph Mining

- Connected components
- Approx. shortest paths
- Triangle counting
- Community Detection

SpMV

- PageRank
- Generic

Recommendations

- Random walks

Collaborative Filtering

(by Danny Bickson)

- ALS
- SGD
- Sparse-ALS
- SVD, SVD++
- Item-CF

Probabilistic Graphical Models

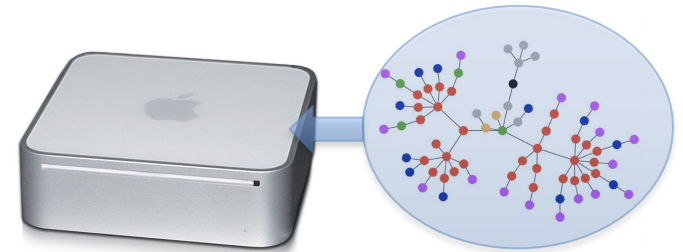
- Belief Propagation

Comparisons to existing systems

IS GRAPHCHI FAST ENOUGH?

Experiment Setting

- Mac Mini (Apple Inc.)
 - 8 GB RAM
 - 256 GB SSD, 1TB hard drive
 - Intel Core i5, 2.5 GHz
- Experiment graphs:



Graph	Vertices	Edges	P (shards)	Preprocessing
live-journal	4.8M	69M	3	0.5 min
netflix	0.5M	99M	20	1 min
twitter-2010	42M	1.5B	20	2 min
uk-2007-05	106M	3.7B	40	31 min
uk-union	133M	5.4B	50	33 min
yahoo-web	1.4B	6.6B	50	37 min

Comparison to Existing Systems

PageRank

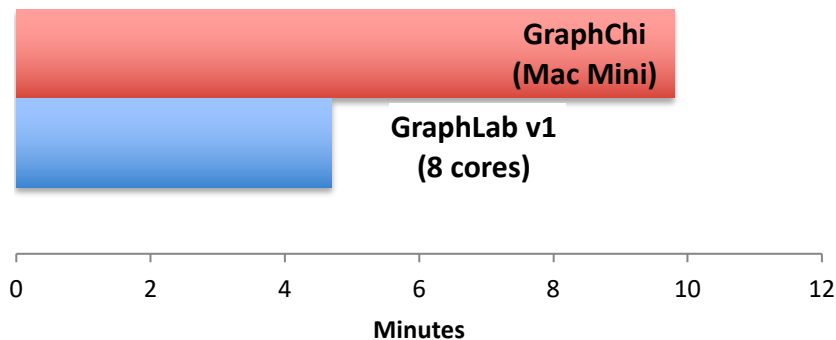
WebGraph Belief Propagation (U Kang et al.)

On a Mac Mini:

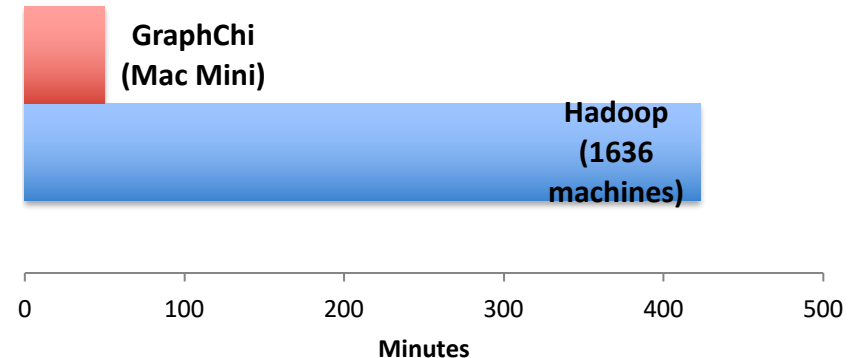
- ✓ GraphChi can solve as big problems as existing large-scale systems.
- ✓ Comparable performance.

Matrix F

Netflix (99B edges)



twitter-2010 (1.5B edges)



PowerGraph Comparison

- **PowerGraph / GraphLab 2** outperforms previous systems by a wide margin on natural graphs.
- With 64 more machines, 512 more CPUs:
 - **Pagerank**: 40x faster than GraphChi
 - **Triangle counting**: 30x faster than GraphChi.

GraphLab 2



vs.



GraphChi

GraphChi has state-of-the-art performance / CPU.

Consult the paper for a comprehensive evaluation:

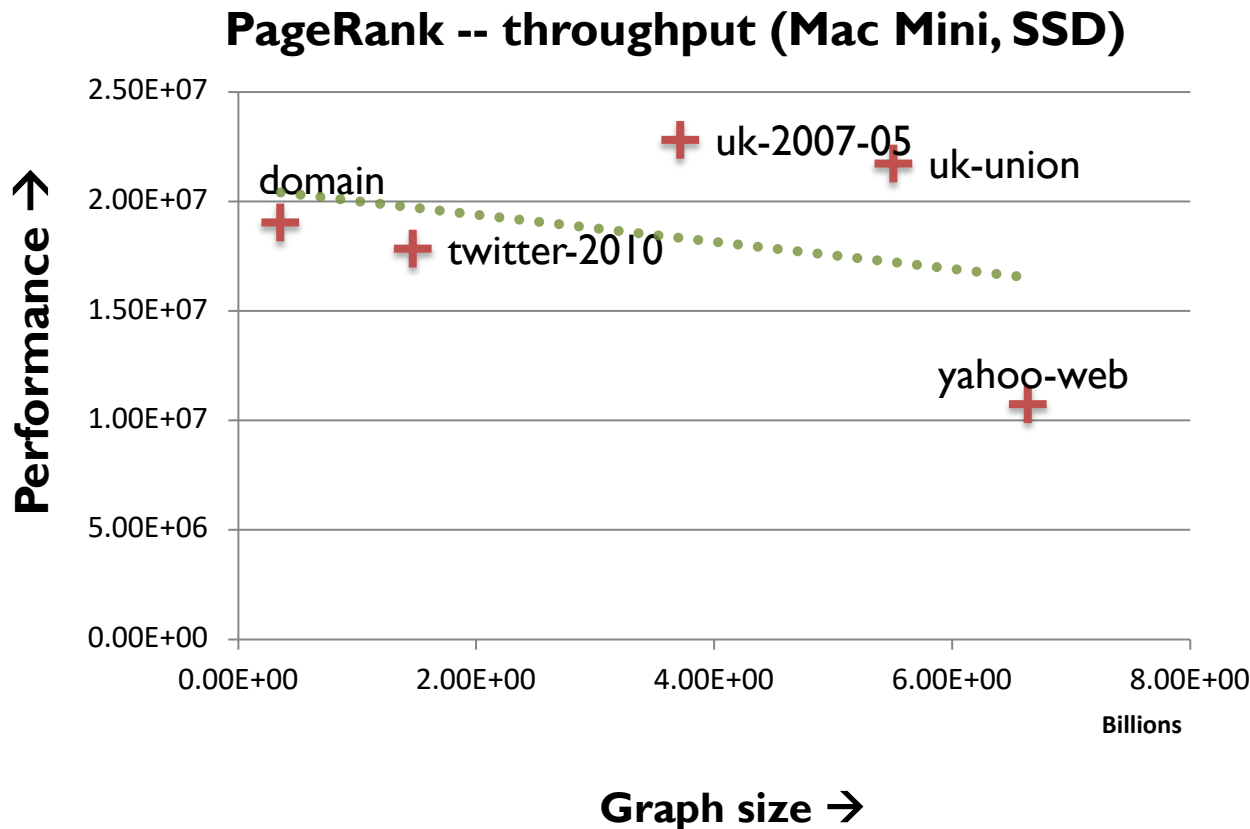
- *HD vs. SSD*
- *Striping data across multiple hard drives*
- *Comparison to an in-memory version*
- *Bottlenecks analysis*
- *Effect of the number of shards*
- *Block size and performance.*

Sneak peek

SYSTEM EVALUATION

Scalability / Input Size [SSD]

- Throughput: number of edges processed / second.

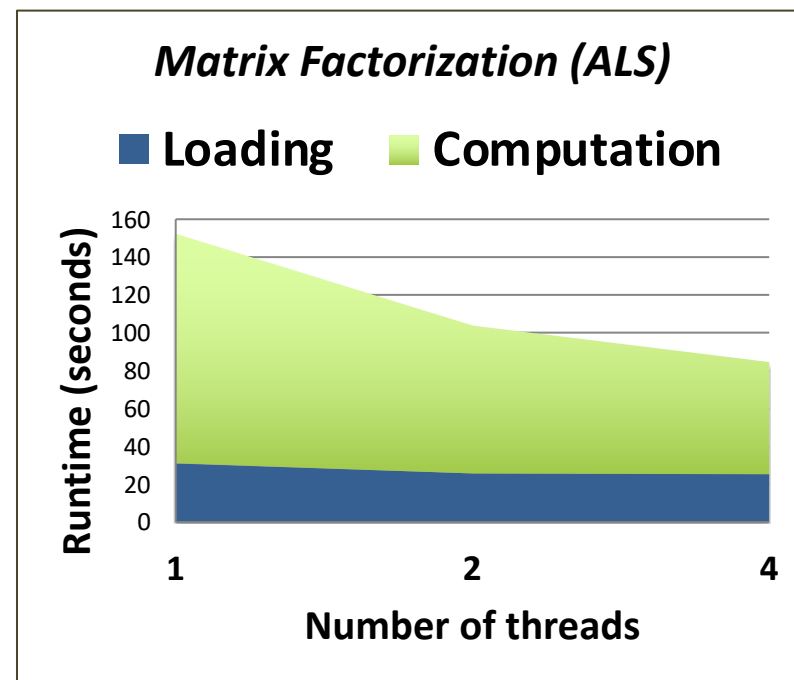
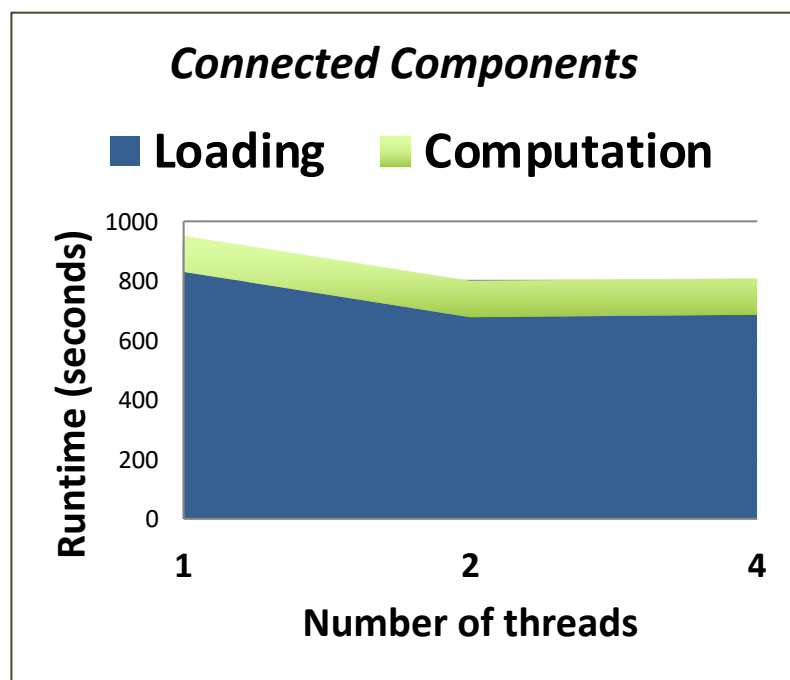


Conclusion: the throughput remains roughly constant when graph size is increased.

GraphChi with hard-drive is ~ 2x slower than SSD (if computational cost low).

Bottlenecks / Multicore

- Computationally intensive applications benefit substantially from parallel execution.
- GraphChi saturates SSD I/O with 2 threads.



Experiment on MacBook Pro with 4 cores / SSD.

Evolving Graphs

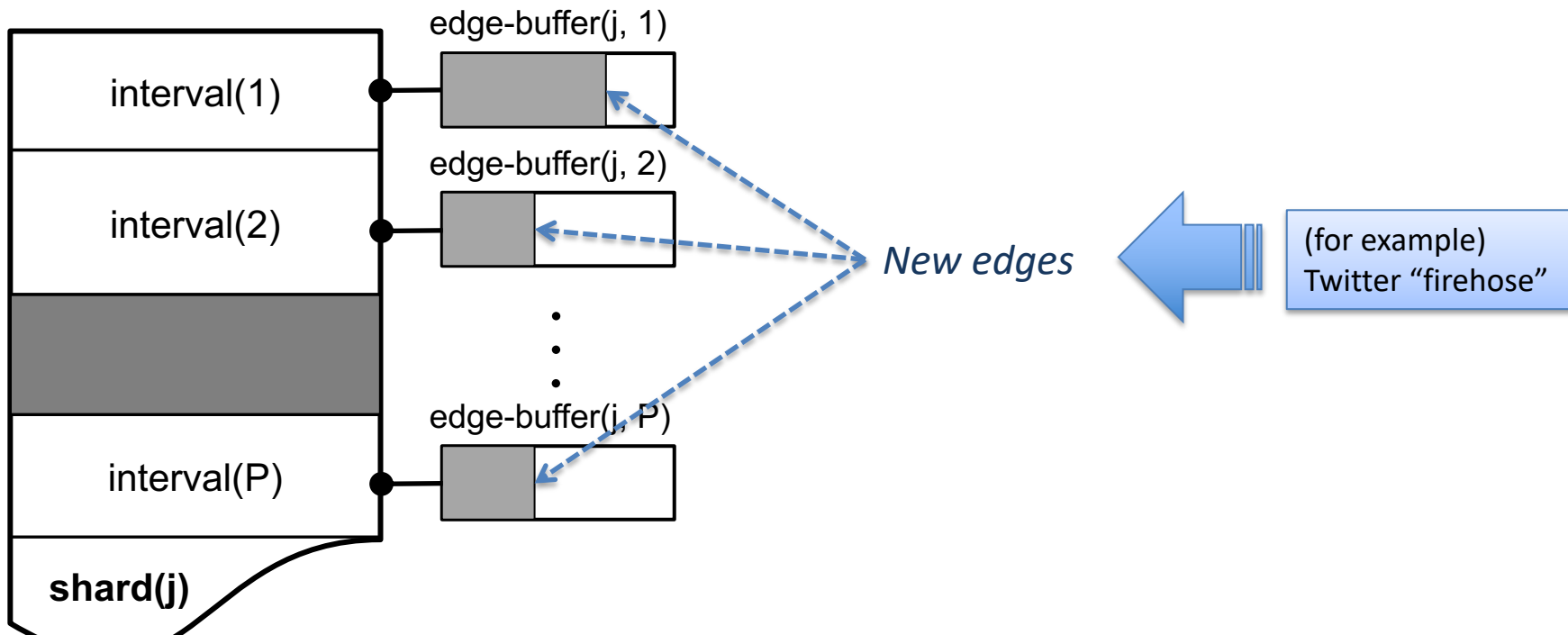
Graphs whose structure changes
over time

Evolving Graphs: Introduction

- Most interesting networks grow continuously:
 - New connections made, some ‘unfriended’.
- Desired functionality:
 - Ability to add and remove edges in streaming fashion;
 - ... while continuing computation.
- Related work:
 - *Kineograph* (EuroSys ‘12), distributed system for computation on a changing graph.

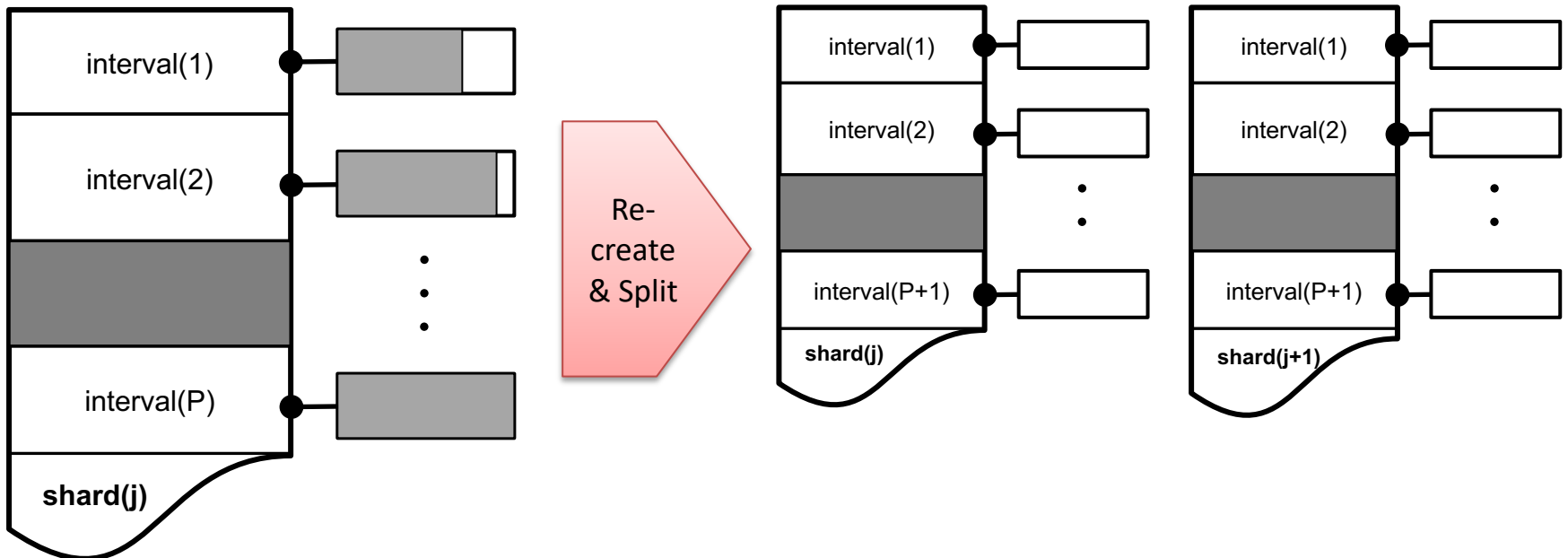
PSW and Evolving Graphs

- Adding edges
 - Each (shard, interval) has an associated edge-buffer.
- Removing edges: Edge flagged as “removed”.



Recreating Shards on Disk

- When buffers fill up, shards are **recreated** on disk
 - Too big shards are **split**.
- During recreation, deleted edges are permanently removed.



Streaming Graph experiment

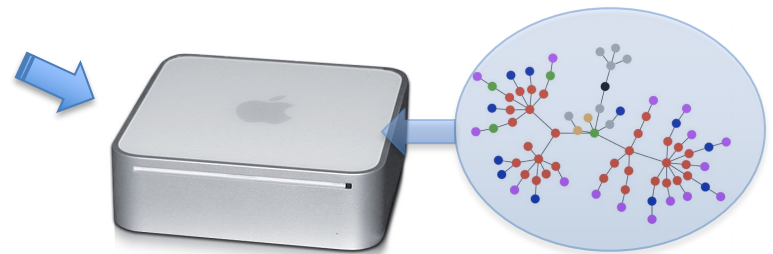
EVALUATION: EVOLVING GRAPHS

Streaming Graph Experiment

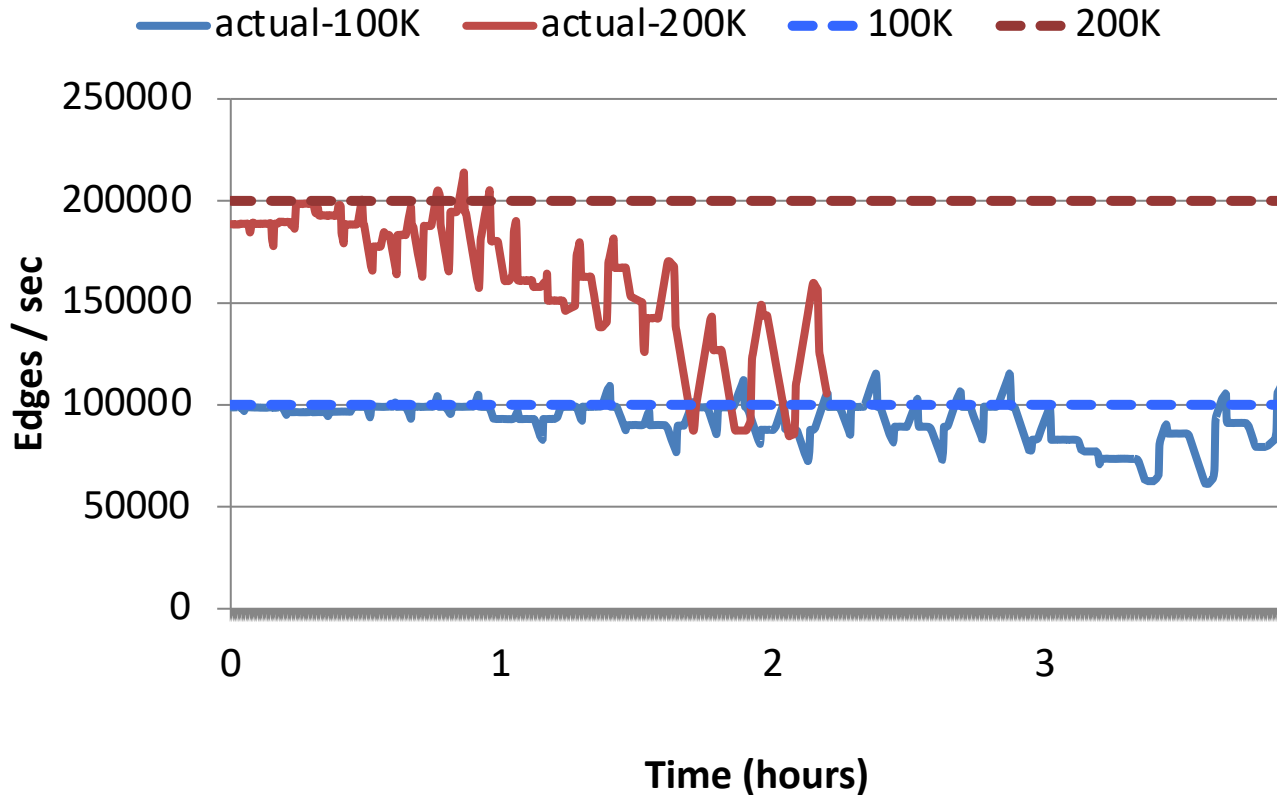
- On the Mac Mini:
 - Streamed edges in random order from the *twitter-2010* graph (1.5 B edges)
 - With maximum rate of 100K or 200K edges/sec. (very high rate)
 - Simultaneously run PageRank.
 - Data layout:
 - Edges were streamed from hard drive
 - Shards were stored on SSD.



edges

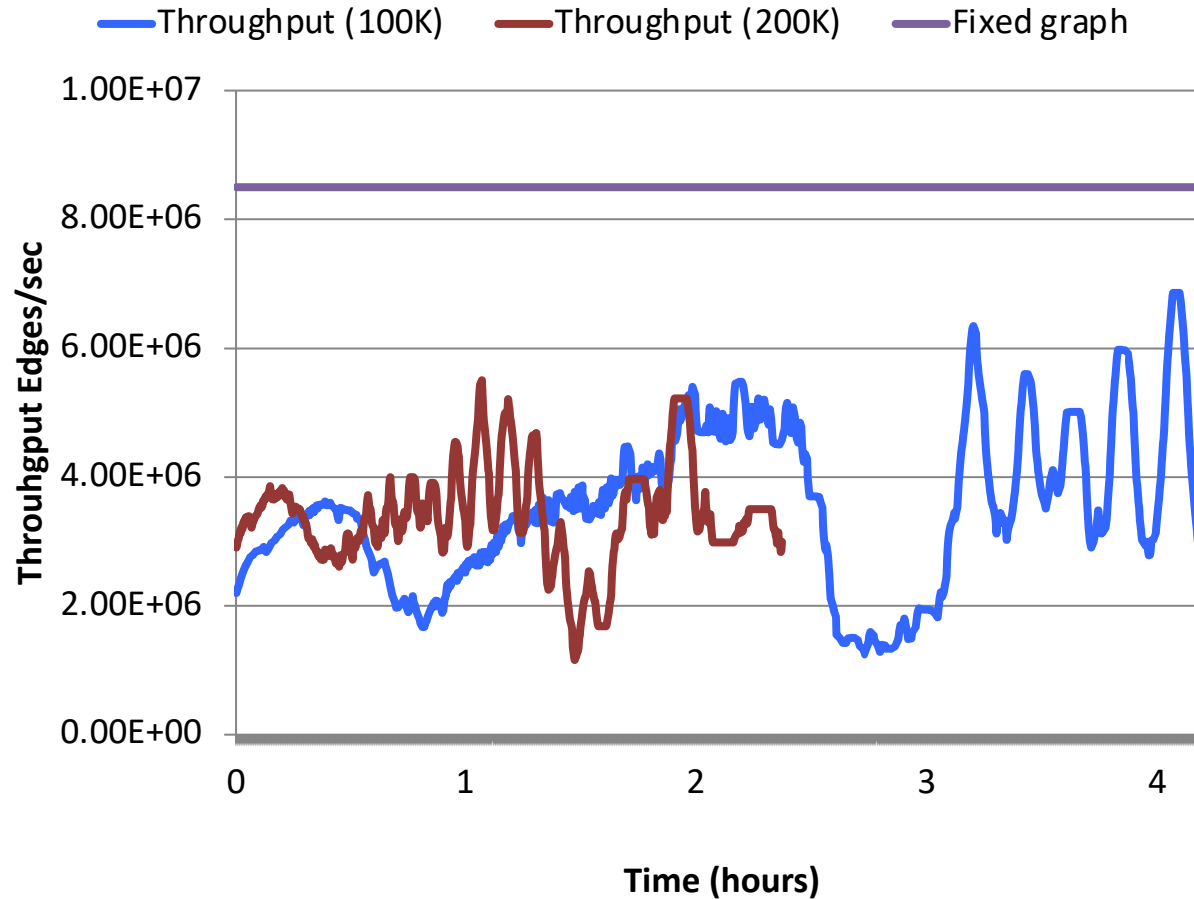


Ingest Rate



When graph grows, shard recreations become more expensive.

Streaming: Computational Throughput



Throughput varies strongly due to shard rewrites and asymmetric computation.

Conclusion

Conclusion

- Parallel Sliding Windows algorithm enables processing of large graphs with very few non-sequential disk accesses.
- For the system researchers, GraphChi is a solid baseline for system evaluation
 - It can solve as big problems as distributed systems.
- Takeaway: Appropriate data structures as an alternative to scaling up.

Source code and examples: <http://graphchi.org>
License: Apache 2.0