Graph-Based Parallel Computing

William Cohen

Computing paradigms

1. Stream-and-sort
2. Iterative streaming ML (eg SGD)
   - with minibatch + vectorization and GPUs
3. Map-reduce (stream-and-sort + parallelism)
   - plus dataflow-language abstractions
4. Iterative parameter mixing ($\sim= 2 + 3$)
5. Spark ($\sim= 2$ + iteration + caching)
6. ….?

Many ML algorithms tend to have

- Sparse data dependencies
- Local computations
- Iterative updates

- Typical example: PageRank
  - repeat:
    - for each node, collect/combine incoming PRs
    - for each node, send outgoing PR

```
previous_pagerank =
    LOAD '$docs_in'
    USING PigStorage('\t')
    AS ( url: chararray, pagerank: float, links:{ link: ( url: chararray ) } );

outbound_pagerank =
    FOREACH previous_pagerank
    GENERATE
        pagerank / COUNT ( links ) AS pagerank,
        FLATTEN ( links ) AS to_url;

new_pagerank =
    FOREACH
        ( COGROUP outbound_pagerank BY to_url, previous_pagerank BY url INNER )
    GENERATE
        group AS url,
        ( 1 - $d ) + $d * SUM ( outbound_pagerank.pagerank ) AS pagerank,
        FLATTEN ( previous_pagerank.links ) AS links;

STORE new_pagerank
    INTO '$docs_out'
    USING PigStorage('\t');
```

lots of i/o happening here…

```python
def pageRankPlanner():

    p = Planner()

    def serialize(graphView):
        return \
            Format(graphView,
                   by=lambda(url,pagerank,outlinks):'\t'.join([url,'%g'%pagerank]+outlinks))

    # read in and create initial ranked graph
    p.edges = ReadLines(EDGEFILE) | Map(by=lambda line:line.strip().split(' '))
    p.initGraph = Group(p.edges, by=lambda (src,dst):src, retaining=lambda(src,dst):dst)
    p.initRankedGraph = Map(p.initGraph, by=lambda (url,outlinks):(url,1.0,outlinks))
    p.serializedInitRankedGraph = serialize(p.initRankedGraph)

    # one step of the update, reading the last iteration from a temp file
    p.prevGraph = \
        ReadLines(TMPFILE) \
        | Map(by=lambda line:line.strip().split("\t")) \
        | Map(by=lambda parts:(parts[0],float(parts[1]),parts[2:]))
    p.outboundPageRankMessages =  \
        FlatMap(p.prevGraph,
                by=lambda (url,pagerank,outlinks):
                map(lambda dst:(dst,pagerank/len(outlinks)), outlinks))
    p.newPageRank = \
        Group(p.outboundPageRankMessages,
              by=lambda (dst,deltaPageRank):dst,
              retaining=lambda (dst,deltaPageRank):deltaPageRank,
              reducingTo=ReduceTo(lambda:(RESET), lambda accum,delta:accum + (1-RESET)*delta))
    p.newRankedGraph = \
        Join( Jin(p.prevGraph, by=lambda (url,pagerank,outlinks):url),
              Jin(p.newPageRank, by=lambda (dst,newPageRank):dst)) \
        | Map(by=lambda((url,oldPageRank,outlinks),(url_,newPageRank)):(url,newPageRank,outlinks))
    p.serializedRankedGraph = serialize(p.newRankedGraph)

    p.setup()
    return p
```

Many ML algorithms tend to have

- Sparse data dependencies
- Local computations
- Iterative updates

- Typical example: PageRank
  - repeat:
    - for each node, collect/combine incoming PRs
    - for each node, send outgoing PR

Many Graph-Parallel Algorithms

- Collaborative Filtering
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - Tensor Factorization
- Structured Prediction
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling
- Semi-supervised ML
  - Graph SSL
  - CoEM
- Community Detection
  - Triangle-Counting
  - K-core Decomposition
  - K-Truss
- Graph Analytics
  - PageRank
  - Personalized PageRank
  - Shortest Path
  - Graph Coloring
- Classification
  - Neural Networks

Suggested architecture

- A large **mutable** graph stored in distributed memory
  - Repeat some node-centric computation until convergence
  - Node values change and edges (mostly) don't
  - Node updates depend (mostly) on their neighbors in the graph
  - Node updates are done in parallel

Sample system: Pregel

Pregel (Google, Sigmod 2010)

- Primary data structure is a graph
- Computations are sequence of *supersteps,* in each of which
  - user-defined function (UDF) is invoked (in parallel) at each vertex *v*, can get/set *value*
  - UDF can also issue requests to get/set edges
  - UDF can read *messages* sent to *v* in the last superstep and schedule messages to *send* to in the next superstep
  - Halt when every vertex *votes* to halt
- Output is directed graph
- Also: aggregators (like ALLREDUCE)
- Bulk synchronous processing (BSP) model: all vertex operations happen **simultaneously**

vertex value changes

communication

Pregel (Google, Sigmod 2010)

- One master: partitions the graph among workers
- Workers keep graph “shard” in memory
- Messages to other partitions are buffered

- Communication across partitions is expensive, within partitions is cheap
  – quality of partition makes a difference!

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

everyone computes in parallel

simplest rule: stop when *everyone* votes to halt

Figure 3: The Vertex API foundations.

Streaming PageRank: with some long rows

recap

- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^{t}$

- Store **A** as a list of edges: each line is: "i d(i) j"
- Store **v'** and **v** in memory: **v'** starts out as c**u**
- For each line "i d j"
  - **v'**[j] += (1-c)**v**[i]/d

note we need to scan through the **graph** each time

We need to get the degree of *i* and store it locally

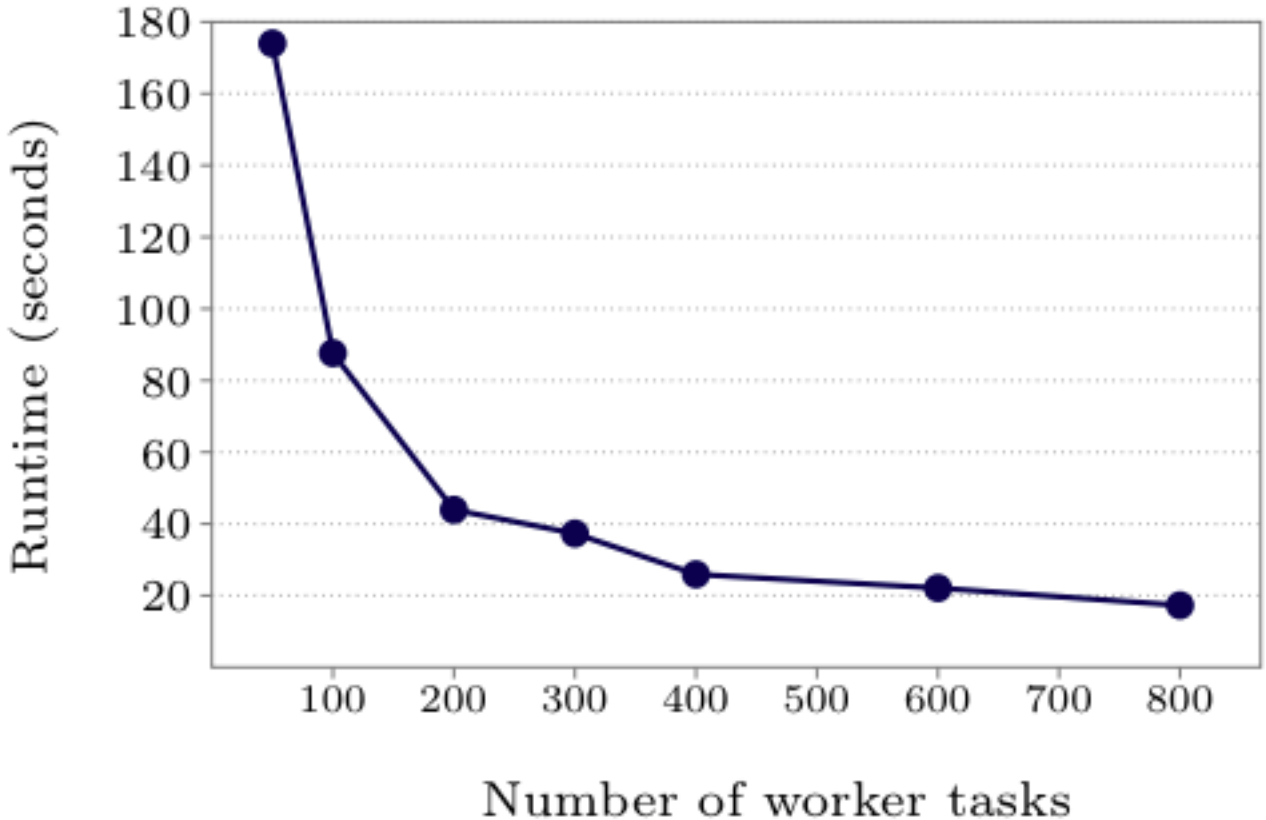
```
class PageRankVertex
    : public Vertex<double, void, double> {
 public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
          0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Another task: single source shortest path

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

edge weight



a little bit of a cheat

Figure 7: SSSP—1 billion vertex binary tree: varying number of worker tasks scheduled on 300 multicore machines

Sample system: Signal-Collect

Signal/collect model *vs* Pregel

- Integrated with RDF/SPARQL
- Vertices can be non-uniform types
- **Vertex**:
  - *id*, mutable *state*, outgoing *edges, most recent received signals* (map: neighbor id→signal), *uncollected signals*
  - user-defined *collect* function
- **Edge:** *id, source, dest*
  - user-defined *signal* function
- Allows *asynchronous* computations….via v.scoreSignal, v.scoreCollect

For "data-flow" operations

On multicore architecture: shared memory for workers

Signal/collect model

```
v.doSignal()
  lastSignalState := state
  for all (e ∈ outgoingEdges) do
    e.target.uncollectedSignals.append(e.signal())
    e.target.signalMap.put(e.sourceId, e.signal())
  end for
```

signals are made available in a list and a map

```
v.doCollect()
  state := collect()
  uncollectedSignals := Nil
```

next state for a vertex is output of the collect() operation

relax "num_iterations" soon

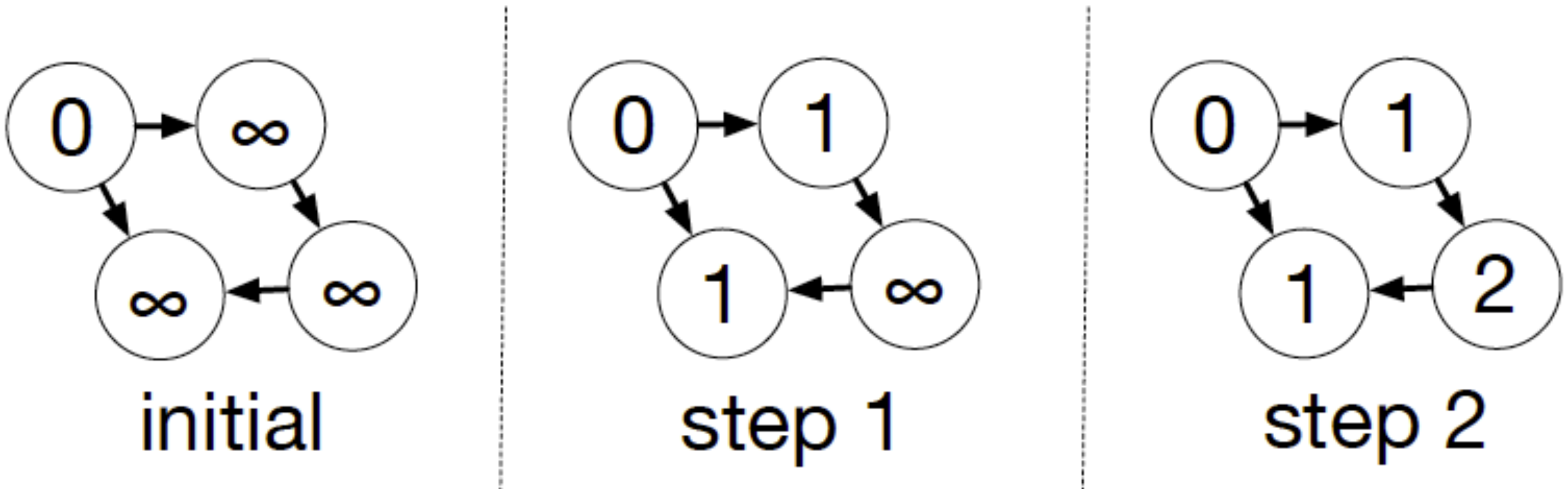
Algorithm 1 Synchronous execution

```
for i ← 1..num_iterations do
  for all v ∈ V parallel do
    v.doSignal()
  end for
  for all v ∈ V parallel do
    v.doCollect()
  end for
end for
```

Signal/collect examples

Single-source shortest path

`initialState`	`if (isSource) 0 else infinity`
`collect()`	`return min(oldState, min(signals))`
`signal()`	`return source.state + edge.weight`



Signal/collect examples

Life

`initialState`	`if (isInitiallyAlive) 1 else 0`
`collect()`	see below
`signal()`	`return source.state`

```
switch (sum(signals))
  case 0: return 0        // dies of loneliness
  case 1: return 0        // dies of loneliness
  case 2: return oldState // same as before
  case 3: return 1        // becomes alive if dead
  other:  return 0        // dies of overcrowding
```

PageRank

`initialState`	`baseRank`
`collect()`	`return baseRank + dampingFactor * sum(signals)`
`signal()`	`return source.state * edge.weight / sum(edgeWeights(source))`

PageRank + Preprocessing and Graph Building

Algorithm

```
class Document(id: Any) extends Vertex(id, 0.15) {
  def collect = 0.15 + 0.85 * signals[Double].foldLeft(0.0)(_ + _)
  override def processResult = if (state > 5) println(id + ": " + state)
  override def scoreSignal = (state - lastSignalState.getOrElse(0)).abs
}
```

```
class Citation(citer: Any, cited: Any) extends Edge(citer, cited) {
  override type SourceVertexType = Document
  def signal = source.state * weight / source.sumOfOutWeights
}
```

Initialization

Execution

```
object Algorithm {
  def executeCitationRank(db: SparqlAccessor) {
    val computeGraph = new ComputeGraph(ScoreGuidedSynchronous)
    val citations = new SparqlTuples(db, "select ?source ?target where {"
      + "?source <http://lsdis.cs.uga.edu/projects/semdis/opus#cites> ?target}")
    citations foreach {
      case (citer, cited) =>
        computeGraph.addVertex[Document](citer)
        computeGraph.addVertex[Document](cited)
        computeGraph.addEdge[Citation](citer, cited)
    }
    computeGraph.execute(signalThreshold = 0)
  }
}
```

Signal/collect examples

Co-EM/wvRN/Harmonic fields

`initialState`	`if (isTrainingData) trainingData else avgProbDist`

`collect()`	`if (isTrainingData)`
	`  return oldState`
	`else`
	`  return signals.sum.normalise`

`signal()`	`return source.state`

`initialState`	`Set(id)`
`collect()`	`return union(oldState, union(signals))`
`signal()`	`return source.state`

Fig. 8. Transitive closure (data-graph/data-flow).

`initialState`	`randomColour`

`collect()`	`if (contains(signals, oldState))`
	`  return randomColorExcept(oldState)`
	`else`
	`  return oldState`

`signal()`	`return source.state`

Fig. 9. Vertex colouring (data-graph).

`initialState`	`0`
`collect()`	`return 1 / (1 +` $e^{-\mathbf{signals.sum}}$`)`
`signal()`	`return source.state * edge.weight`

Fig. 15. Artificial neural networks (data-graph).

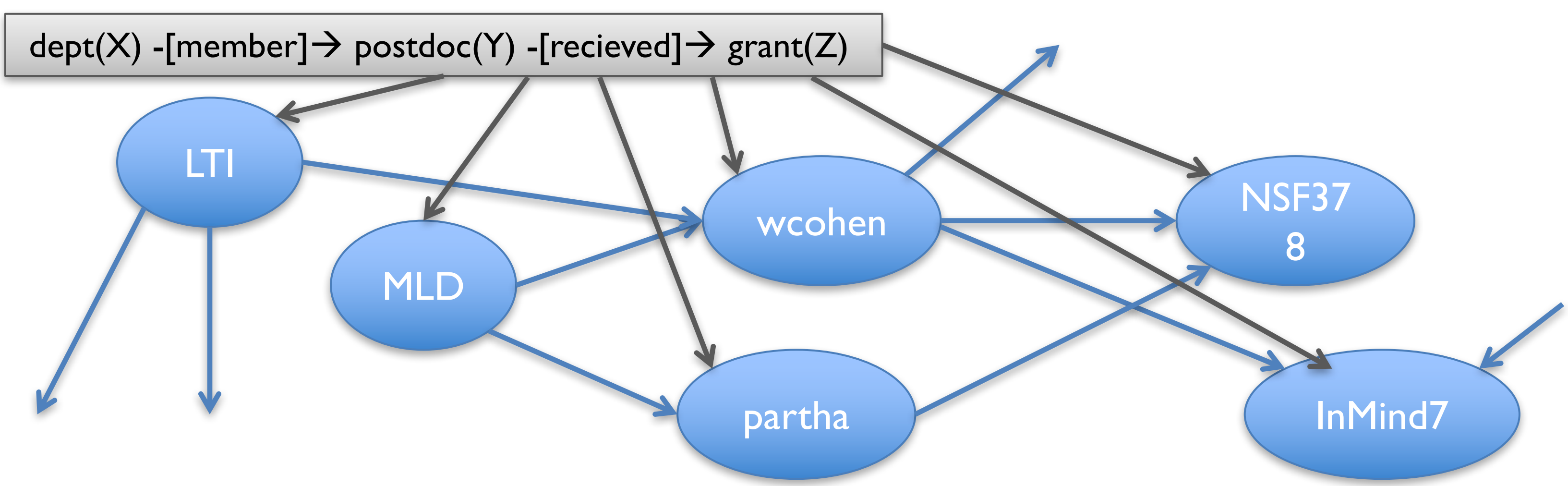
Signal/collect examples

Matching path queries:
dept(X) -[member]→ postdoc(Y) -[recieved]→ grant(Z)

`initialState`	`emptySet`

`collect()`	`matched = successfulMatchesWithVertex(signals)`
	`(fullyMatched, partiallyMatched) = partition(matched)`
	`reportResults(fullyMatched)`
	`return union(oldState - lastSignalState, partiallyMatched)`

`signal()`	`return successfulMatchesWithEdge(source.state)`



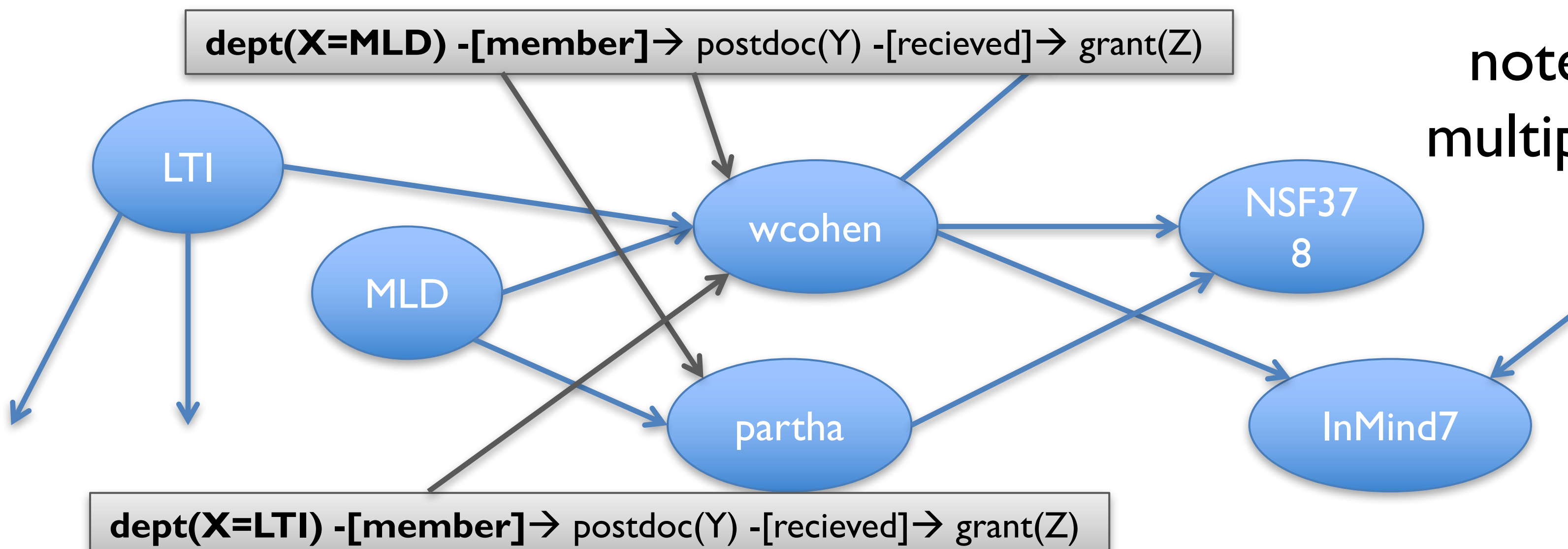
Signal/collect examples: data flow

Matching path queries:
dept(X) -[member]→ postdoc(Y) -[recieved]→ grant(Z)

`initialState`	`emptySet`

`collect()`	`matched = successfulMatchesWithVertex(signals)`
	`(fullyMatched, partiallyMatched) = partition(matched)`
	`reportResults(fullyMatched)`
	`return union(oldState - lastSignalState, partiallyMatched)`

`signal()`	`return successfulMatchesWithEdge(source.state)`



note: can be multiple input signals

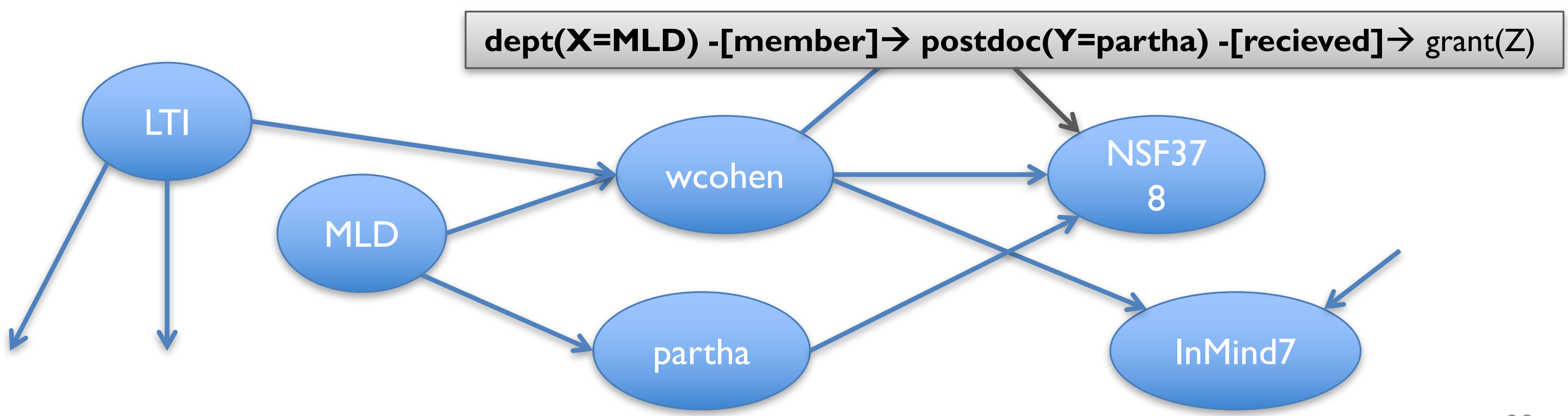
Signal/collect examples

Matching path queries:
dept(X) -[member]→ postdoc(Y) -[recieved]→ grant(Z)

`initialState`	`emptySet`

`collect()`	`matched = successfulMatchesWithVertex(signals)`
	`(fullyMatched, partiallyMatched) = partition(matched)`
	`reportResults(fullyMatched)`
	`return union(oldState - lastSignalState, partiallyMatched)`

`signal()`	`return successfulMatchesWithEdge(source.state)`



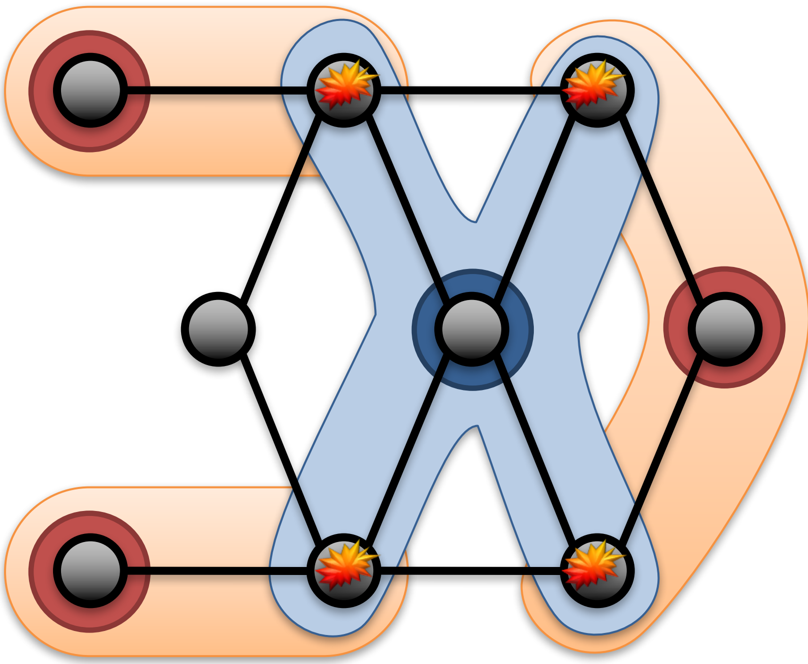
Signal/collect model *vs* Pregel

- Integrated with RDF/SPARQL
- Vertices can be non-uniform types
- **Vertex**:
  - *id*, mutable *state*, outgoing *edges, most recent received signals* (map: neighbor id→signal), *uncollected signals*
  - user-defined *collect* function
- **Edge:** *id, source, dest*
  - user-defined *signal* function
- Allows *asynchronous* computations….via **v.scoreSignal, v.scoreCollect**

For "data-flow" operations

Asynchronous Parallel Computation

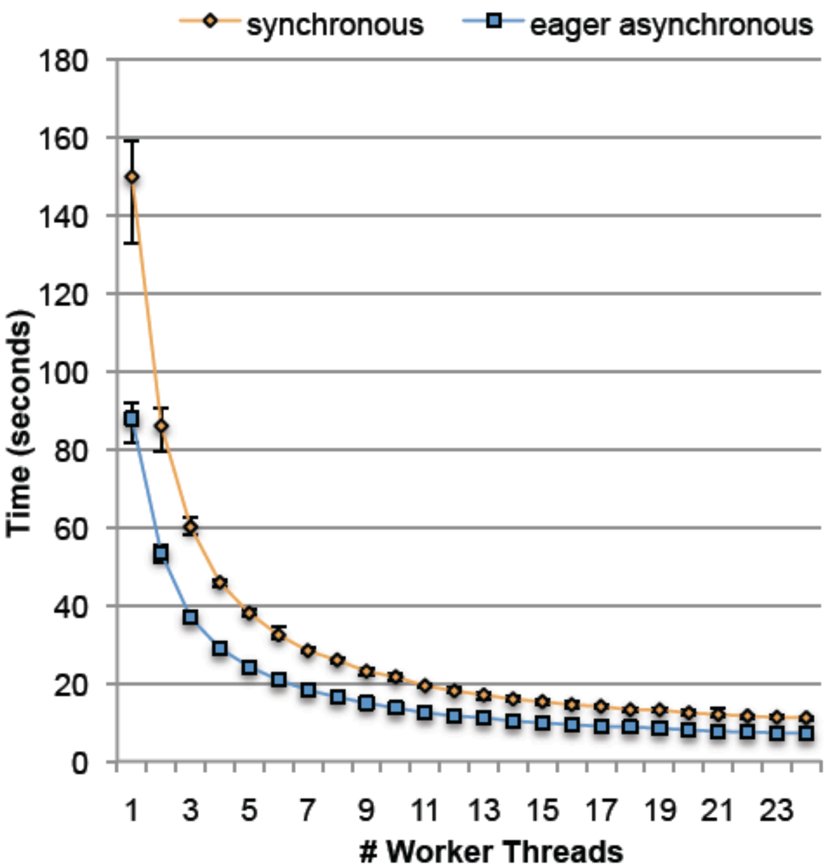
- **Bulk-Synchronous**: All vertices update in parallel
  - need to keep copy of "old" and "new" vertex values
- **Asynchronous**:
  - Reason 1: if two vertices are not connected, can update them in any order
    - more flexibility, less storage
  - Reason 2: not all updates are equally *important*
    - parts of the graph converge quickly, parts slowly

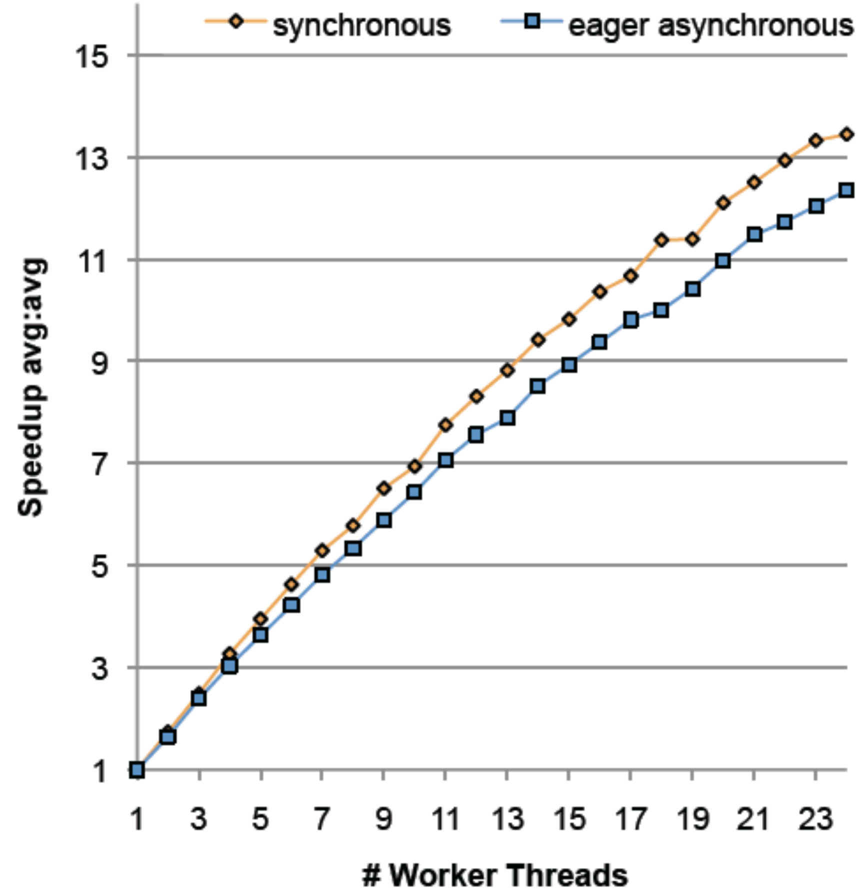


Algorithm 2 Score-guided synchronous execution

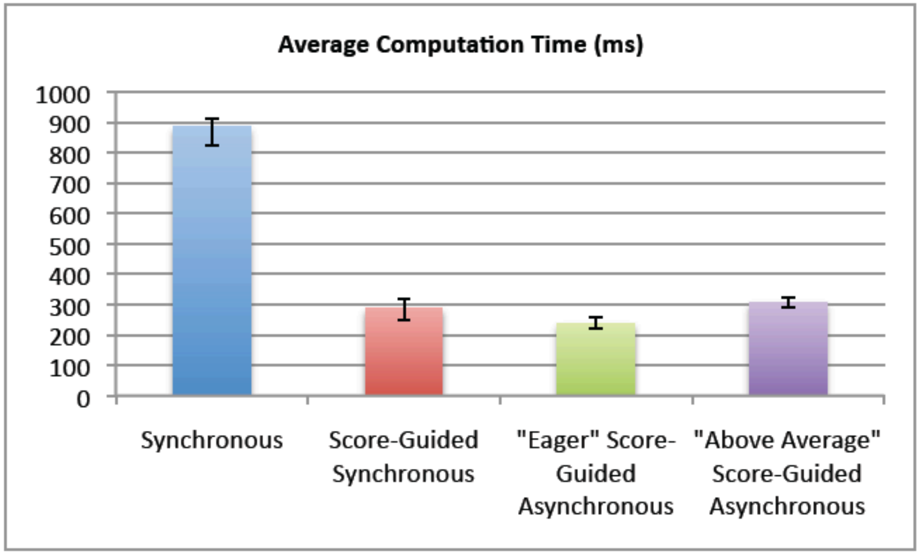
```
done := false
iter := 0
while iter < max_iter and !done do
    done := true
    iter := iter +1
    for all v ∈ V parallel do
        if (v.scoreSignal() > s_threshold) then
            done := false
            v.doSignal()
        end if
    end for
    for all v ∈ V parallel do
        if (v.scoreCollect() > c_threshold)
        then
            done := false
            v.doCollect()
        end if
    end for
end while
```

***using*:**
- *v.scoreSignal*
- *v.scoreCollect*

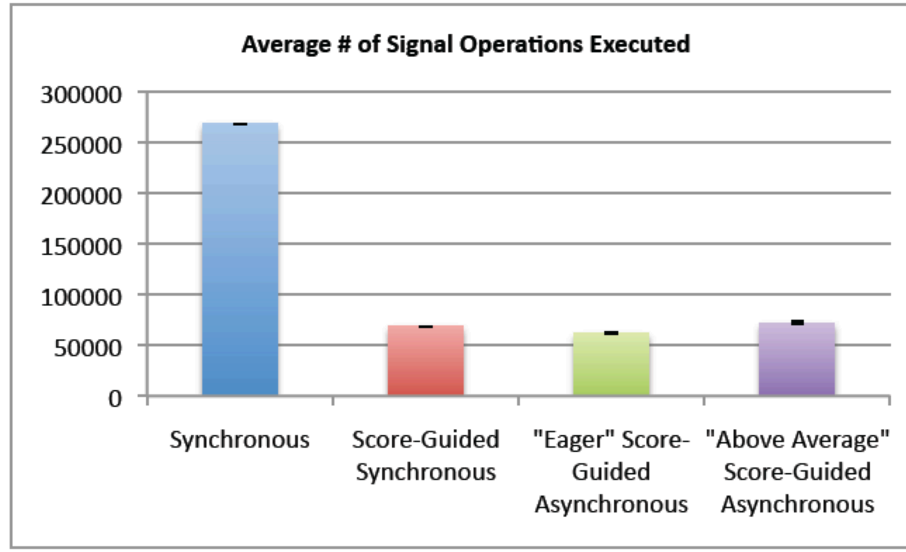

synchronous
eager asynchronous
Time (seconds)
Worker Threads


synchronous
eager asynchronous
Speedup avg:avg
Worker Threads

SSSP



PageRank



Algorithm 3 Score-guided asynchronous execution

ops := 0
while
ops < max_ops **and** $\exists$v $\in V$(
  v.scoreSignal() > s_threshold **or**
  v.scoreCollect() > c_threshold)
do
  S := choose subset of V
  for all v $\in$ S **parallel do**
    Randomly call either v.doSignal() or v.doCollect() iff respective threshold is reached; increment ops if an operation was executed.
  end for
end while

Sample system: GraphLab

GraphLab

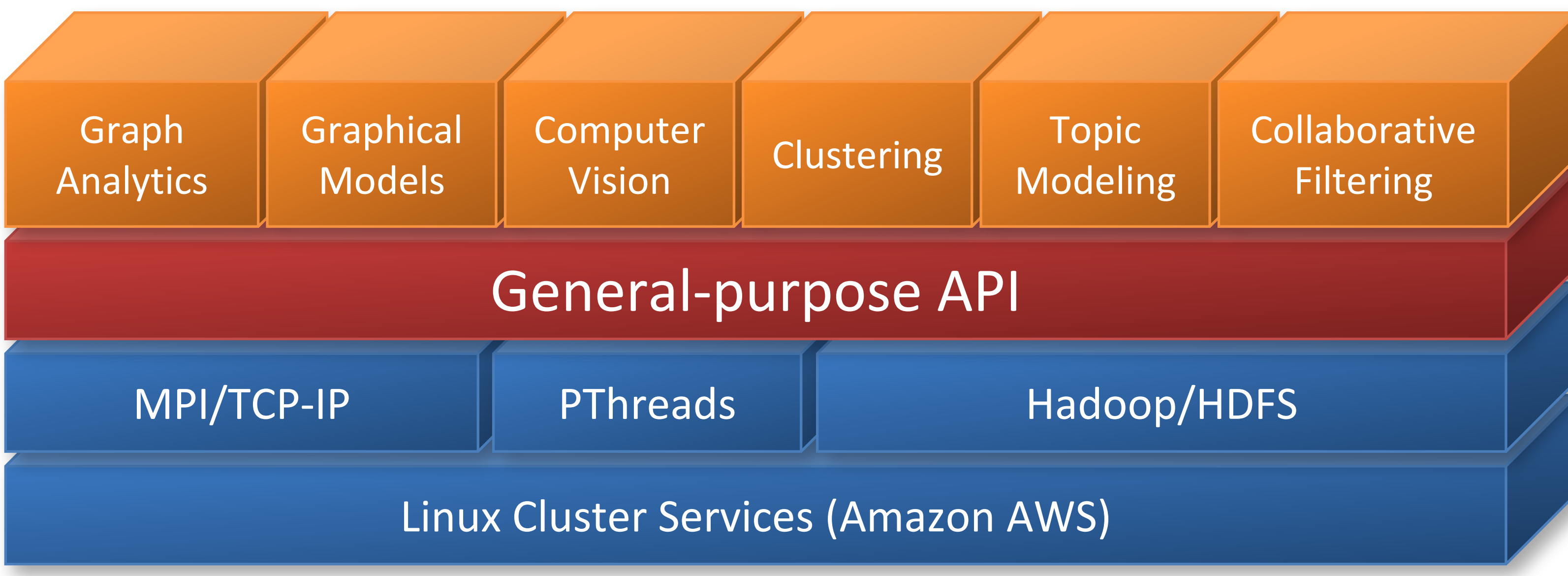
- Data in graph, UDF vertex function
- Differences:
  - some control over scheduling
    - vertex function can insert new tasks in a queue
  - messages must follow graph edges: can access adjacent vertices only
  - “shared data table” for global data
  - library algorithms for matrix factorization, coEM, SVM, Gibbs, …
  - GraphLab → Now Dato

GraphLab's descendents

- PowerGraph
- **GraphChi**
- GraphX

GraphLab con't

- PowerGraph
- **GraphChi**
  - Goal: use graph abstraction on-disk, not in-memory, on a conventional workstation



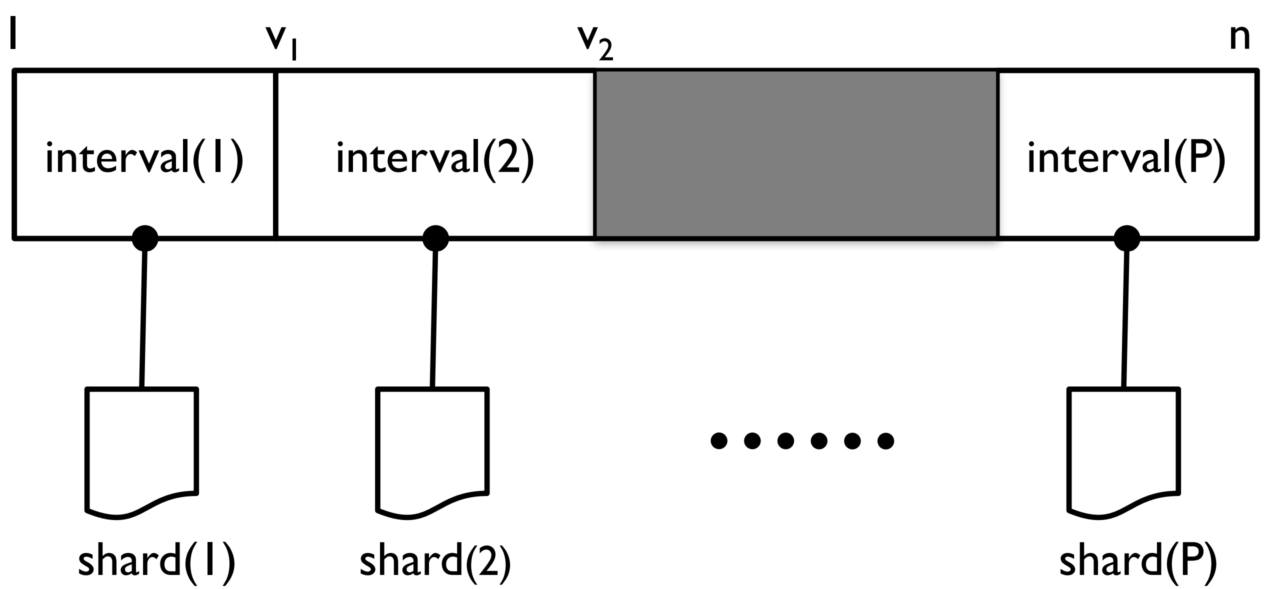
GraphLab con't

- **GraphChi**
  - Key **insight**:
    - in general we can't easily stream the graph because neighbors will be scattered
    - but maybe we can *limit the degree* to which they're scattered … enough to make streaming possible?
      - "almost-streaming": keep P cursors in a file instead of one

1. Load
2. Compute
3. Write

GraphChi: Shards and Intervals

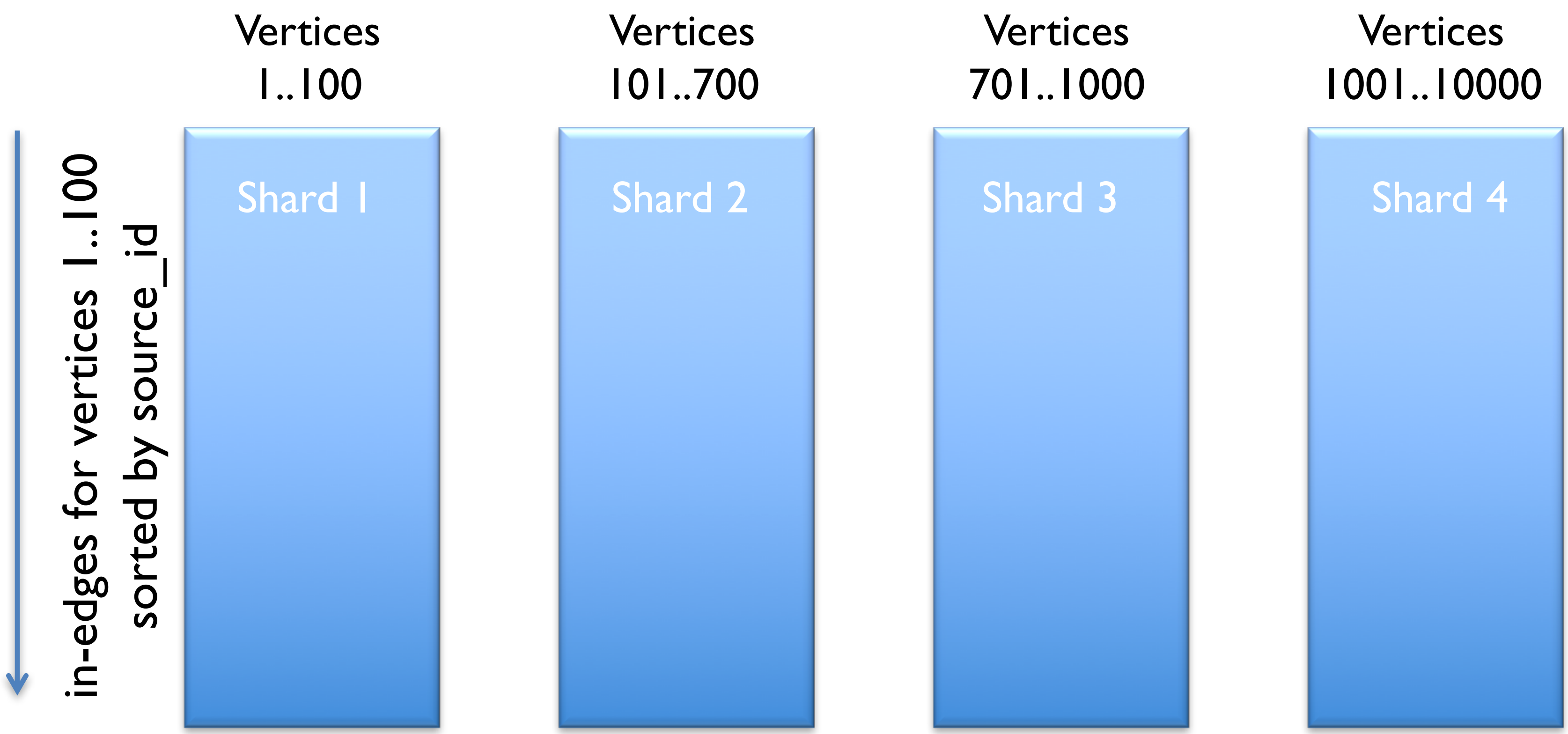
- Vertices are numbered from 1 to n
  - P intervals, each associated with a **shard** on disk.
  - **sub-graph =** interval of vertices



GraphChi: Layout

1. Load
2. Compute
3. Write

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

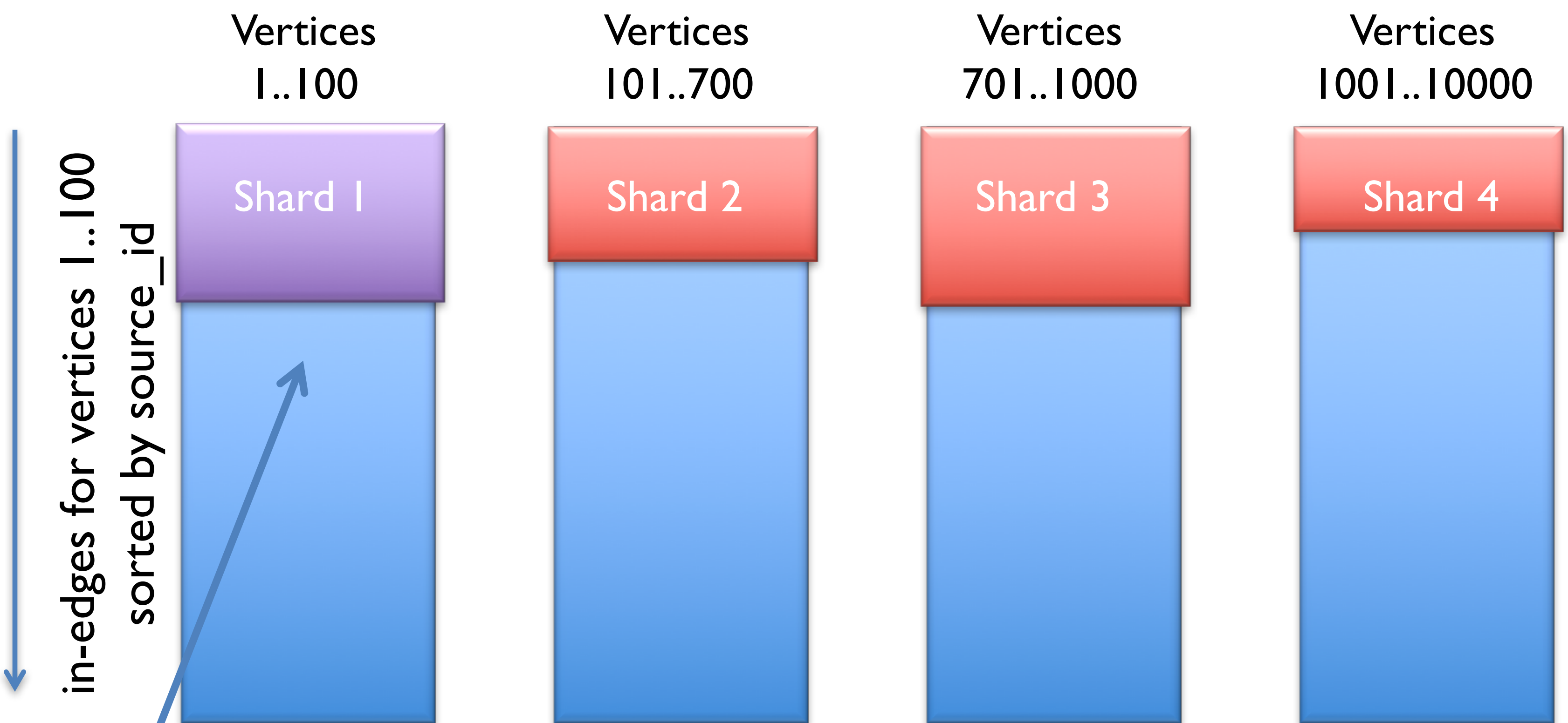


Shards small enough to fit in memory; balance size of shards

GraphChi: Loading Sub-graph

1. Load
2. Compute
3. Write

Load subgraph for vertices 1..100



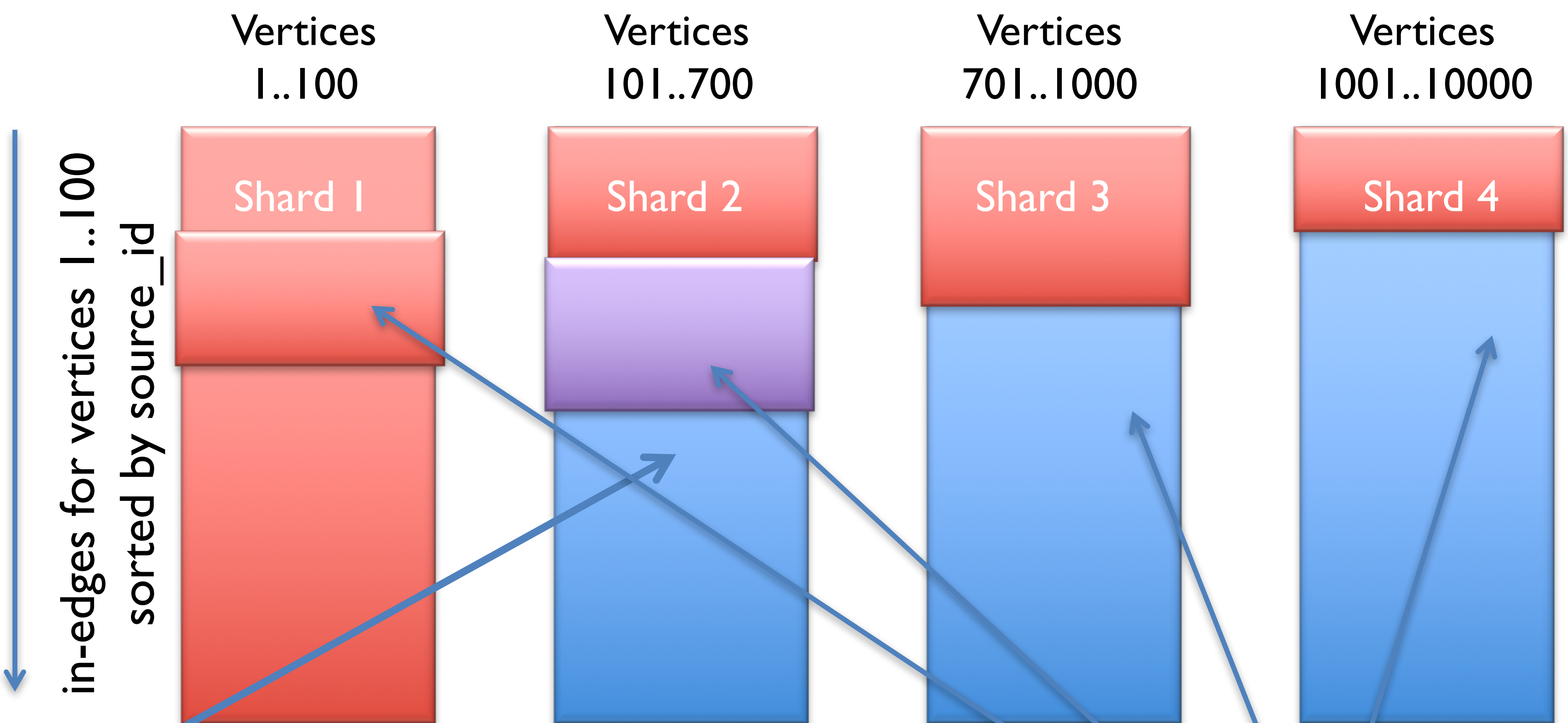
Load all in-edges in memory

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

GraphChi: Loading Sub-graph

1. Load
2. Compute
3. Write

Load subgraph for vertices 101..700



Load all in-edges in memory

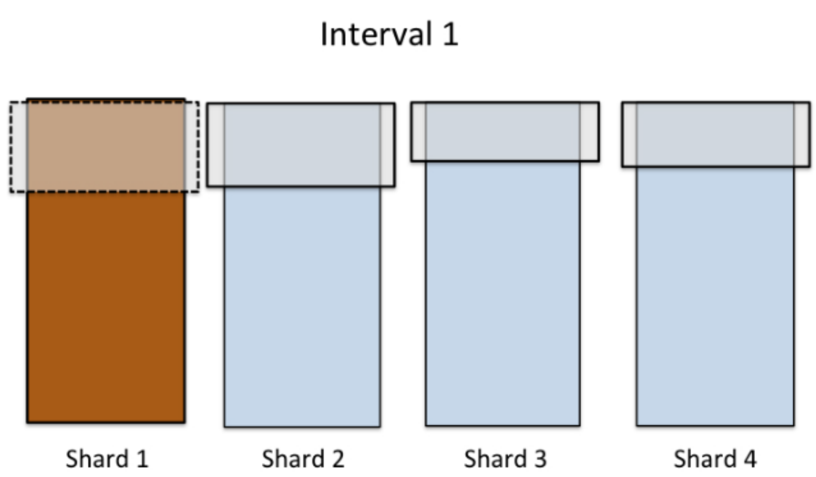
Out-edge blocks in memory

GraphChi Load-Phase

1. Load
2. Compute
3. Write

Only P large reads for each interval.

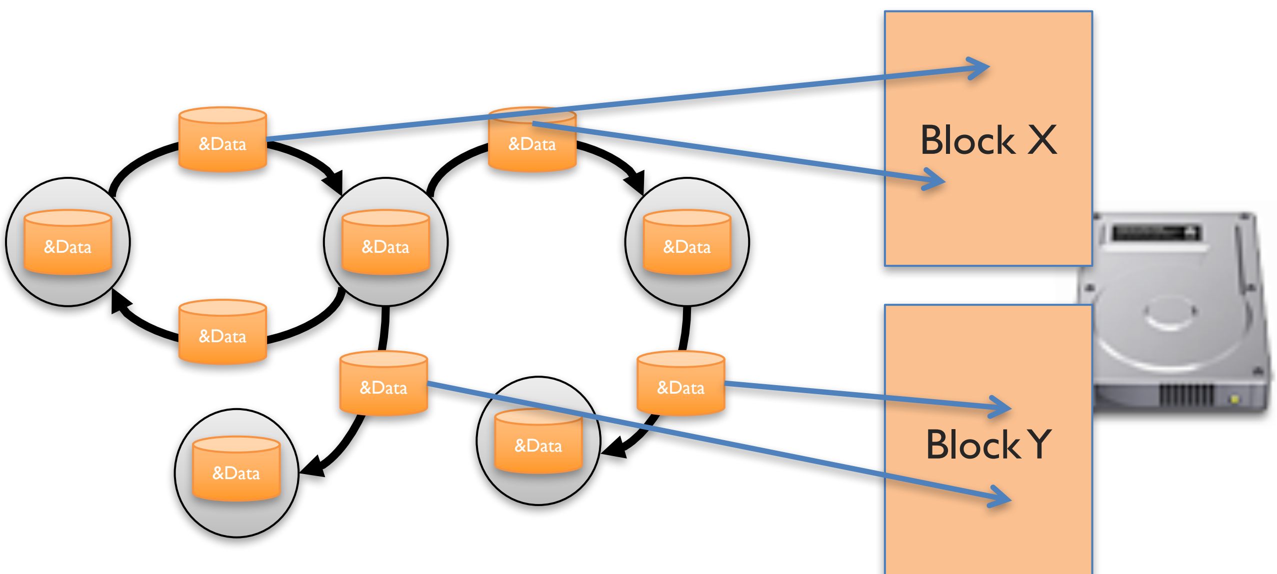
P^2 reads on one full pass.



GraphChi: Execute updates

1. Load
2. Compute
3. Write

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

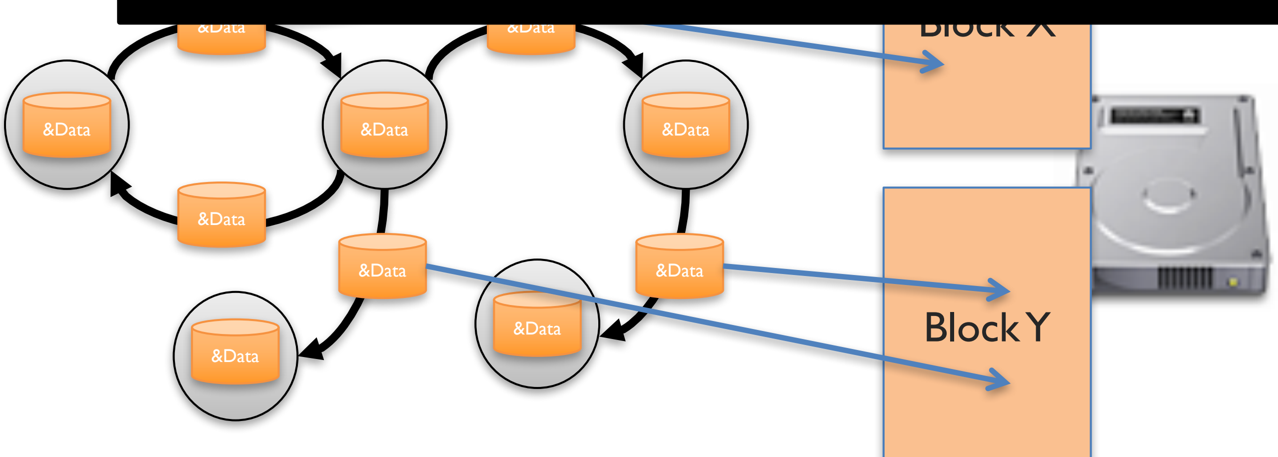


GraphChi: Commit to Disk

1. Load
2. Compute
3. Write

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

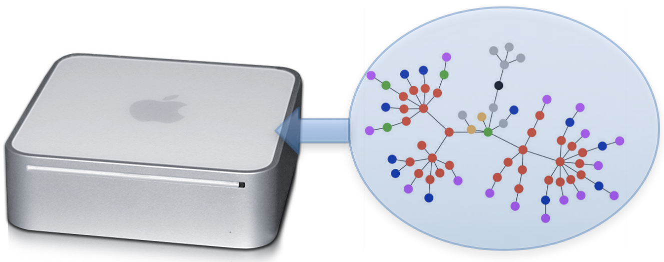
In total:
P^2 reads and writes / full pass on the graph.
→ Performs well on *both* SSD and hard drive.



To make this work: the **size** of a vertex state can't change when it's updated (at last, as stored on disk).

Experiment Setting

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



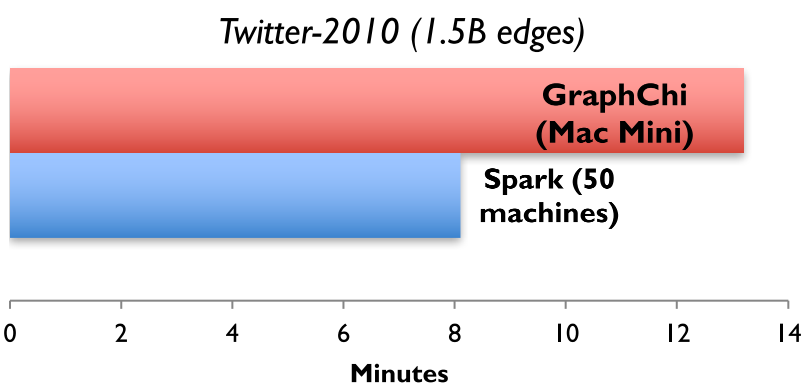
- Experiment graphs:

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

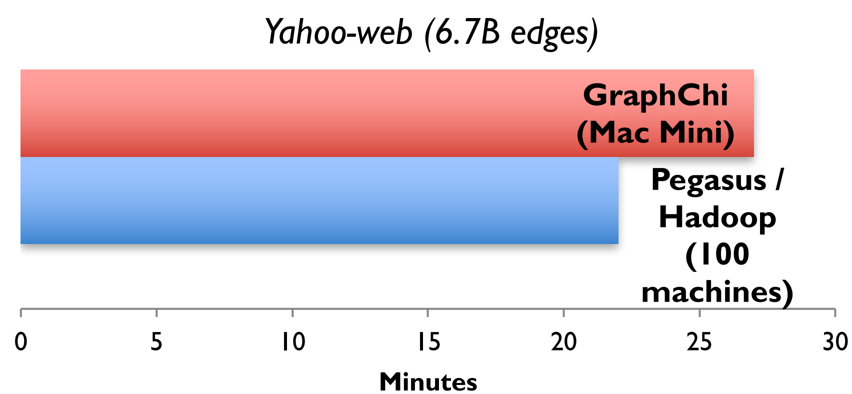
See the paper for more comparisons.

Comparison to Existing Systems

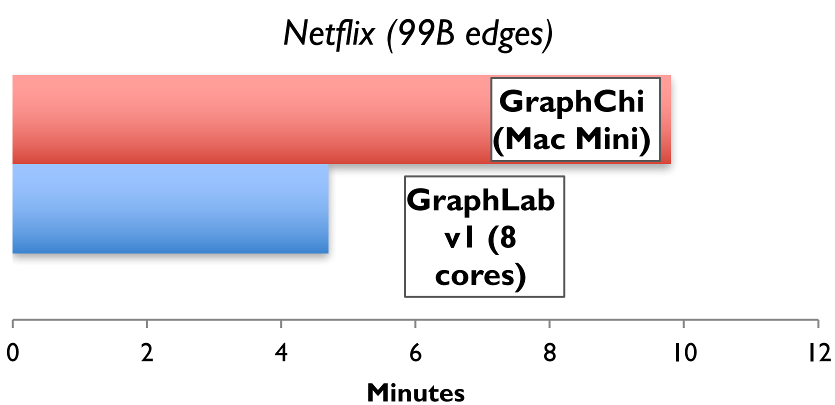
PageRank



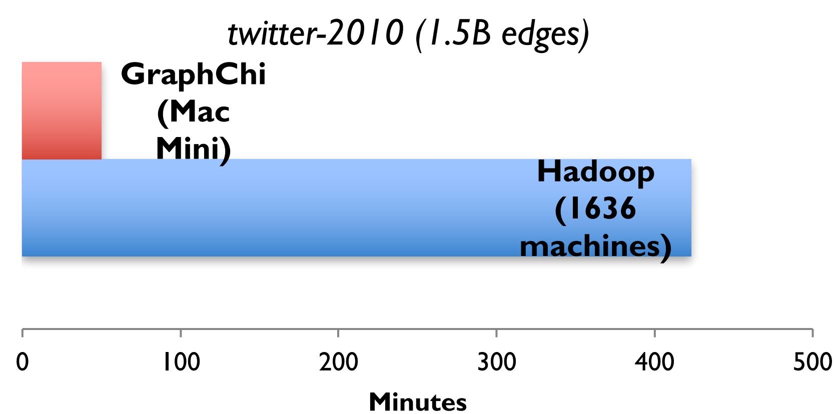
WebGraph Belief Propagation (U Kang



Matrix Factorization (Alt. Least



Triangle Counting



Notes: comparison results do not include time to transfer the data to cluster, preprocessing, or the time to load the graph from disk. GraphChi computes asynchronously, while all but GraphLab synchronously.

GraphLab's descendents

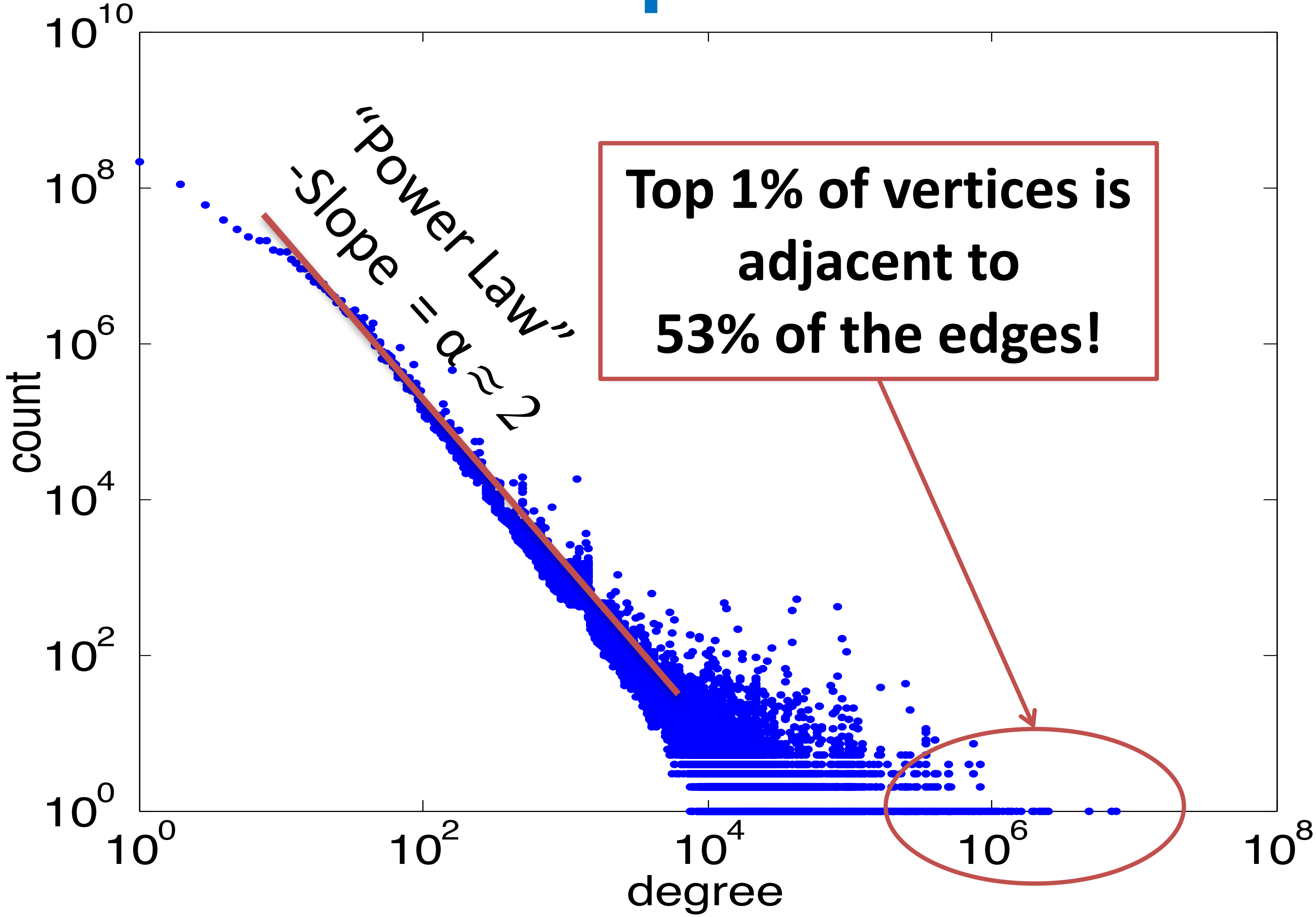
- **PowerGraph**
- GraphChi
- GraphX

~~On multicore architecture: shared memory for workers~~

On cluster architecture (like Pregel): different memory spaces

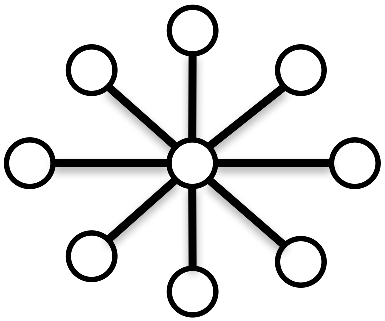
What are the challenges moving away from shared-memory?

Natural Graphs → Power Law

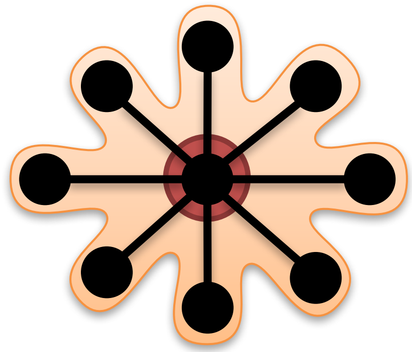


Altavista Web Graph: 1.4B Vertices, 6.7B Edges

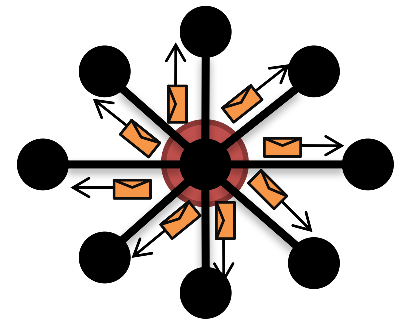
Problem:
High Degree Vertices Limit Parallelism



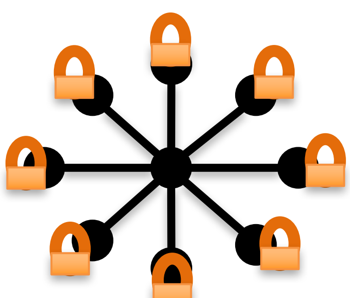
Edge information too large for single machine



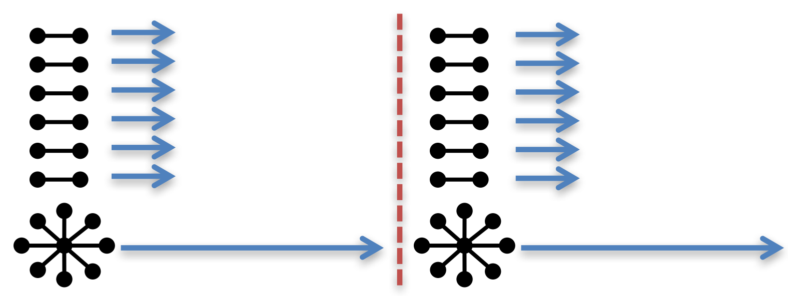
Touches a large fraction of graph (GraphLab 1)



Produces many messages (Pregel, Signal/Collect)



Asynchronous consistency requires heavy locking (GraphLab 1)

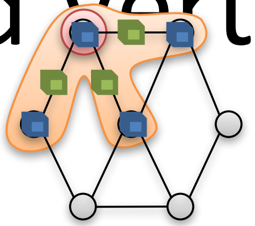


Synchronous consistency is prone to stragglers (Pregel)

PowerGraph

- Problem: GraphLab's localities can be large
  - "all neighbors of a node" can be large for hubs, high indegree nodes
- Approach:
  - new graph partitioning algorithm
    - can **replicate** data
  - gather-apply-scatter API: finer-grained parallelism
    - gather ~ combiner
    - apply ~ vertex UDF (for all replicates)
    - scatter ~ messages from vertex to edges

Factorized Vertex Updates

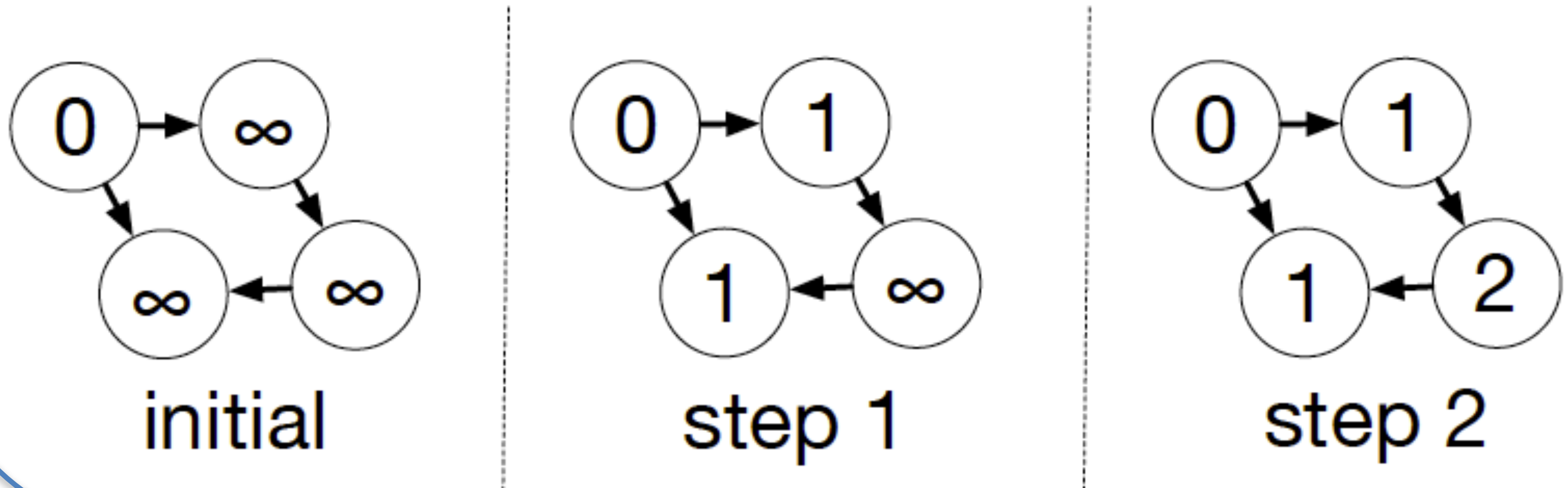


Split update into 3 phases

Signal/collect examples

Single-source shortest path

`initialState`	`if (isSource) 0 else infinity`
`collect()`	`return min(oldState, min(signals))`
`signal()`	`return source.state + edge.weight`



Signal/collect examples

Life

`initialState`	`if (isInitiallyAlive) 1 else 0`

`collect()`	`switch (sum(signals))`
	`case 0: return 0 // dies of loneliness`
	`case 1: return 0 // dies of loneliness`
	`case 2: return oldState // same as before`
	`case 3: return 1 // becomes alive if dead`
	`other: return 0 // dies of overcrowding`

`signal()`	`return source.state`

PageRank

`initialState`	`baseRank`
`collect()`	`return baseRank + dampingFactor * sum(signals)`
`signal()`	`return source.state * edge.weight / sum(edgeWeights(source))`

PageRank + Preprocessing and Graph Building

Algorithm

```
class Document(id: Any) extends Vertex(id, 0.15) {
  def collect = 0.15 + 0.85 * signals[Double].foldLeft(0.0)(_ + _)
  override def processResult = if (state > 5) println(id + ": " + state)
  override def scoreSignal = (state - lastSignalState.getOrElse(0)).abs
}
```

```
class Citation(citer: Any, cited: Any) extends Edge(citer, cited) {
  override type SourceVertexType = Document
  def signal = source.state * weight / source.sumOfOutWeights
}
```

Initialization

Execution

```
object Algorithm {
  def executeCitationRank(db: SparqlAccessor) {
    val computeGraph = new ComputeGraph(ScoreGuidedSynchronous)
    val citations = new SparqlTuples(db, "select ?source ?target where {"
      + "?source <http://lsdis.cs.uga.edu/projects/semdis/opus#cites> ?target}")
    citations foreach {
      case (citer, cited) =>
        computeGraph.addVertex[Document](citer)
        computeGraph.addVertex[Document](cited)
        computeGraph.addEdge[Citation](citer, cited)
    }
    computeGraph.execute(signalThreshold = 0)
  }
}
```

Signal/collect examples

Co-EM/wvRN/Harmonic fields

`initialState`	`if (isTrainingData) trainingData else avgProbDist`

`collect()`	`if (isTrainingData)`
	`  return oldState`
	`else`
	`  return signals.sum.normalise`

`signal()`	`return source.state`

PageRank in PowerGraph

$$R[i] = \beta + (1-\beta) \boxed{\sum_{(j,i)\in E} w_{ji} R[j]}$$

gather/sum like a group by … reduce or *collect*

PageRankProgram(i)

Gather(j → i) : return w_{ji} * R[j]

sum(a, b) : return a + b;

Apply(i, Σ) : R[i] = $\beta + (1 - \beta) * \Sigma$

Scatter(i → j) :

if (R[i] changes) then *activate*(j)

j edge
i vertex

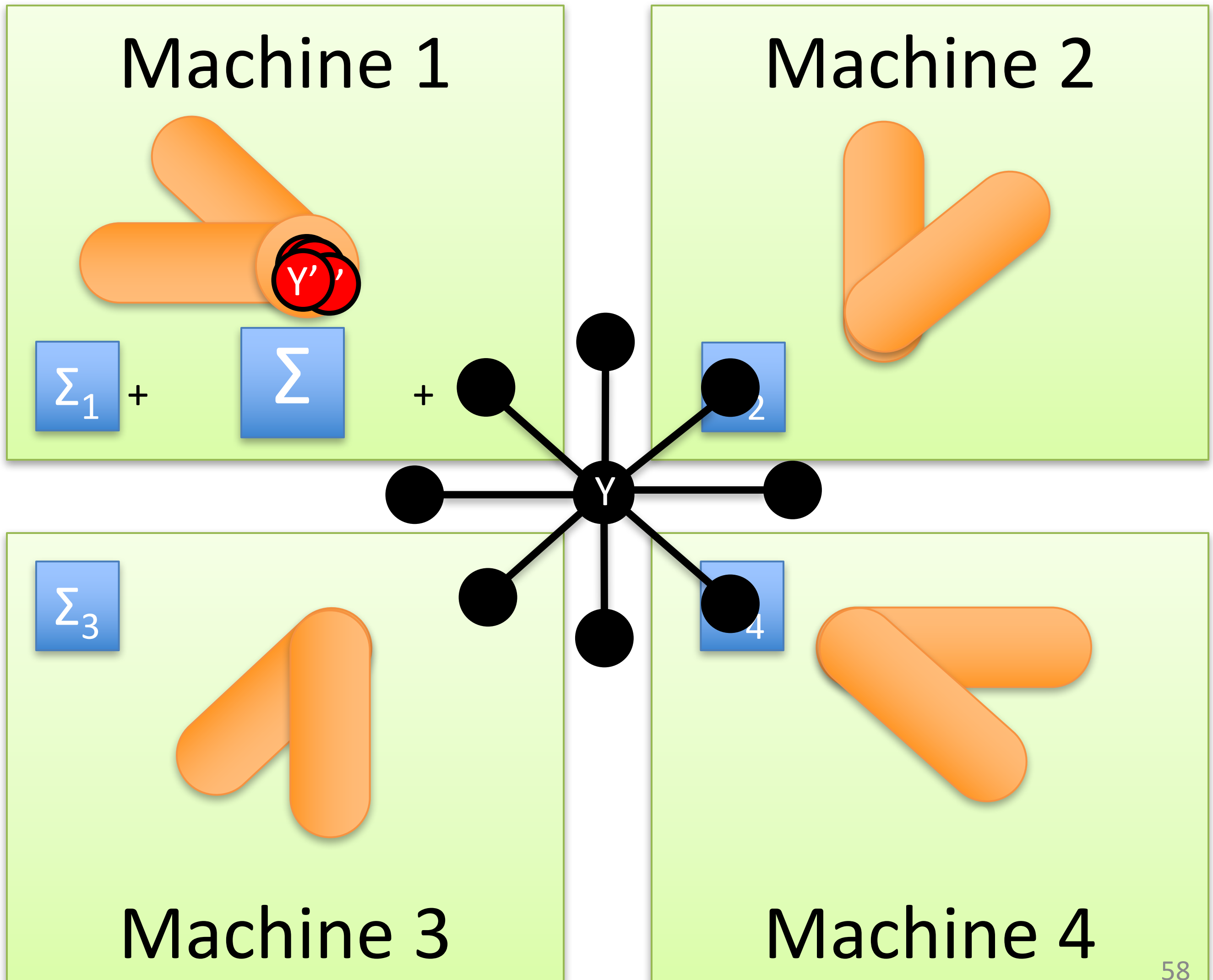
scatter is like a *signal*

Distributed Execution of a PowerGraph Vertex-Program

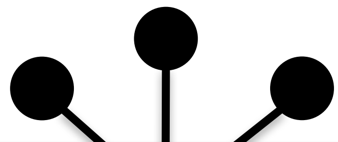
Gather

Apply

Scatter



Minimizing Communication in PowerGraph



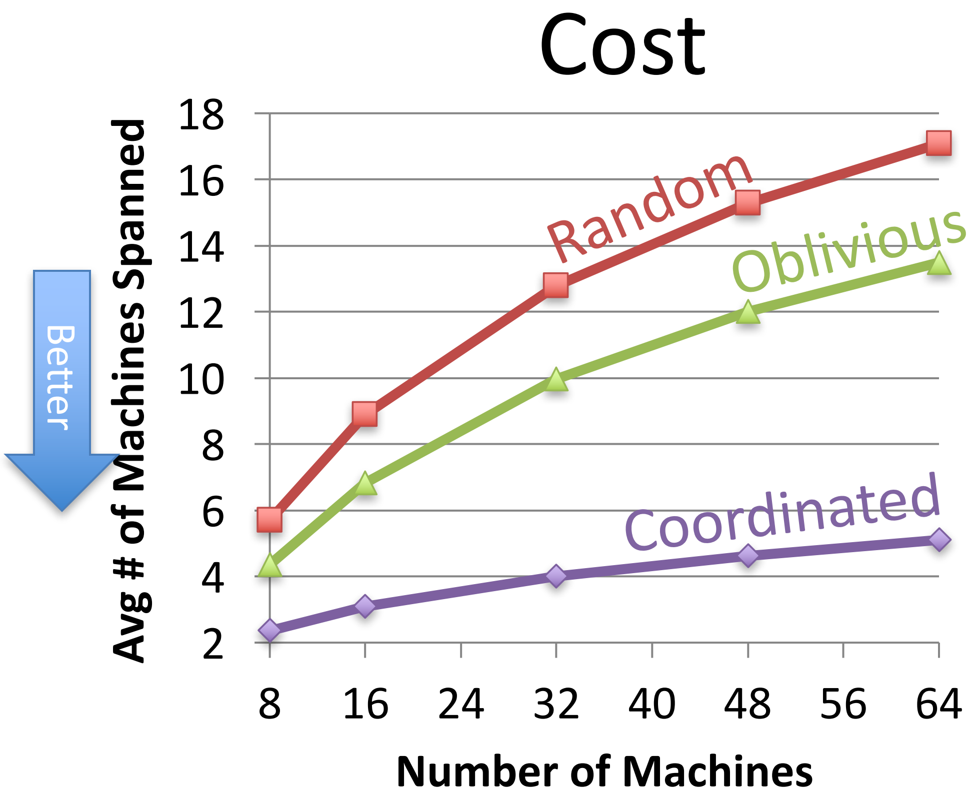
Communication is linear in the number of machines each vertex spans

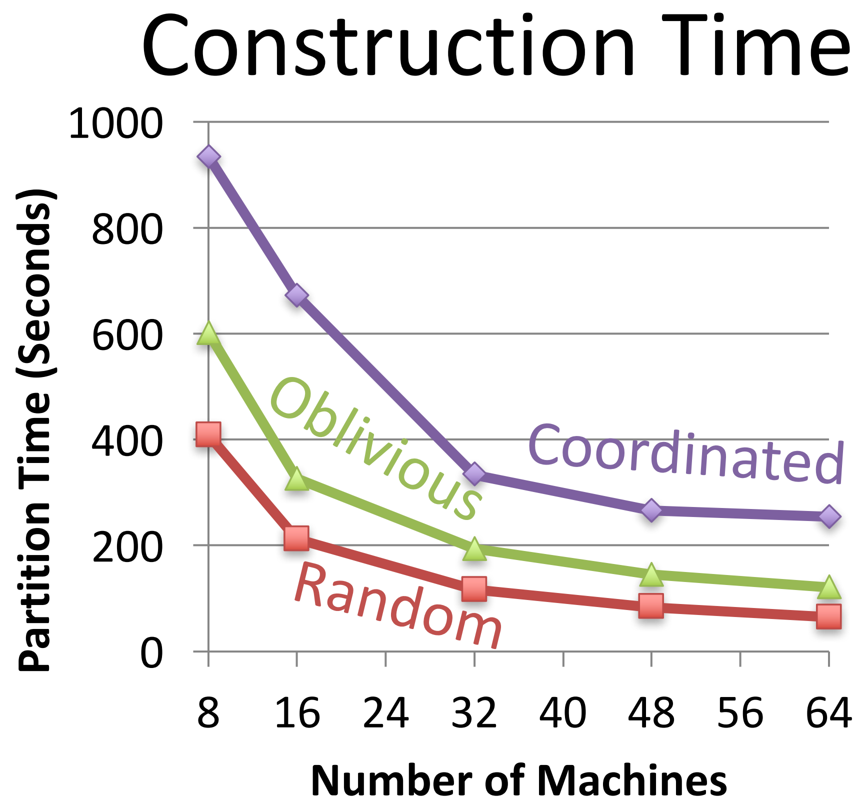
A **vertex-cut** minimizes machines each vertex spans

Percolation theory suggests that power law graphs have good vertex cuts. [Albert et al. 2000]

Partitioning Performance

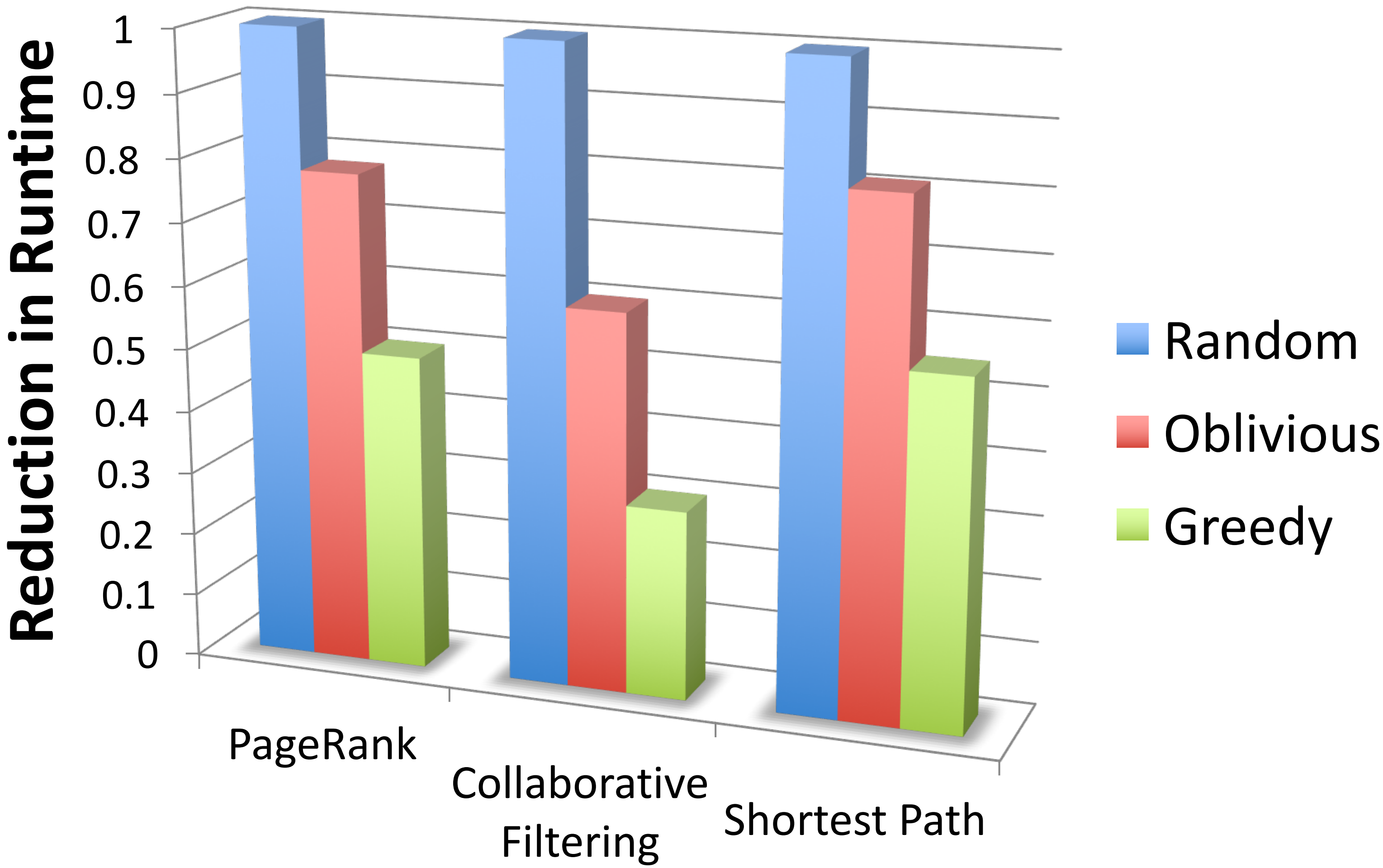
Twitter Graph: 41M vertices, 1.4B edges





Oblivious balances partition quality and partitioning time.

Partitioning matters…



GraphLab’s descendents

- PowerGraph
- GraphChi
- **GraphX**
  - implementation of GraphLabs API on top of Spark
  - Motivations:
    - avoid transfers between subsystems
    - leverage larger community for common infrastructure
  - What’s different:
    - Graphs are now *immutable* and operations transform one graph into another (RDD ➔ RDG, resiliant distributed graph)

Idea 1: Graph as Tables

Property Graph

Under the hood things can be split even more finely: eg a **vertex map table + vertex data table.** Operators maximize structure sharing and minimize communication.

Vertex Property Table

Id	Property (V)
Rxin	(Stu., Berk.)
Jegonzal	(PstDoc, Berk.)
Franklin	(Prof., Berk)
Istoica	(Prof., Berk)

Edge Property Table

SrcId	DstId	Property (E)
rxin	jegonzal	Friend
franklin	rxin	Advisor
istoica	franklin	Coworker
franklin	jegonzal	PI

Operators

- Table (RDD) operators are inherited from Spark:

`map`	`reduce`	`sample`
`filter`	`count`	`take`
`groupBy`	`fold`	`first`
`sort`	`reduceByKey`	`partitionBy`
`union`	`groupByKey`	`mapWith`
`join`	`cogroup`	`pipe`
`leftOuterJoin`	`cross`	`save`
`rightOuterJoin`	`zip`	`...`

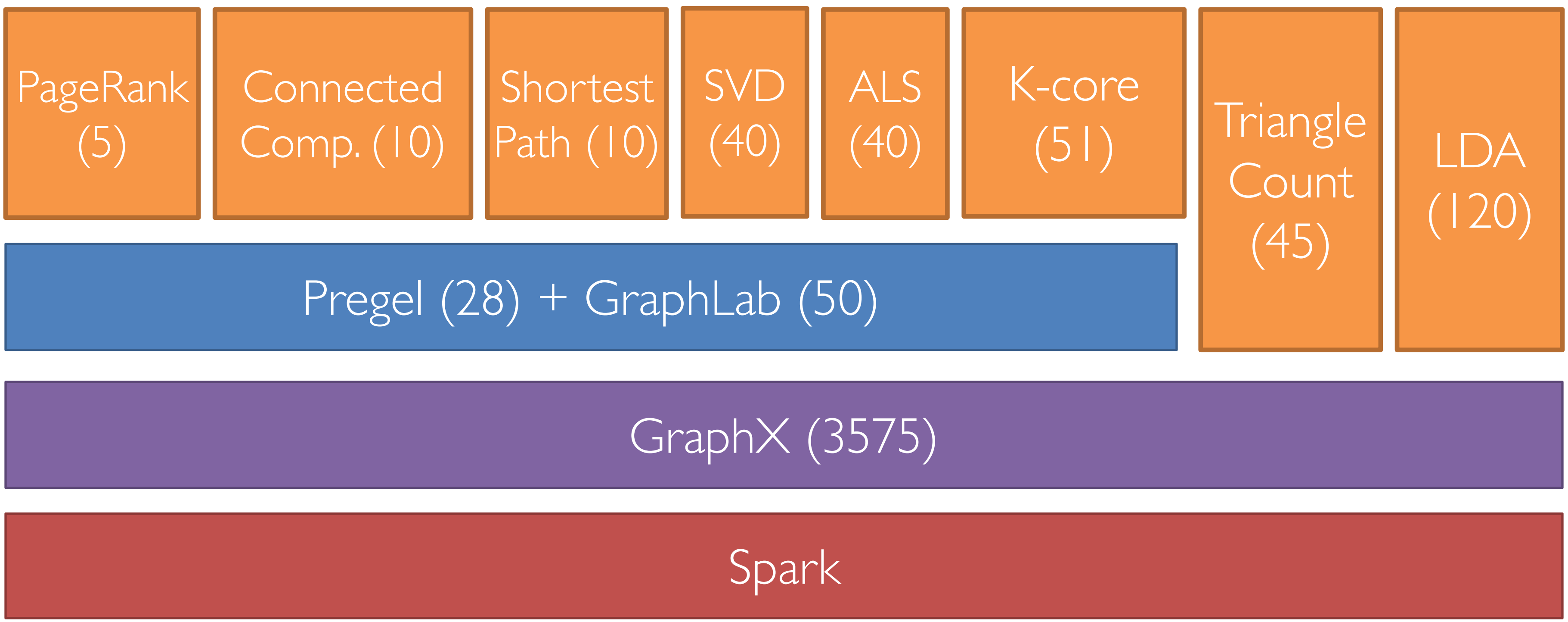
Graph Operators

```
class Graph [ V, E ] {
  def Graph(vertices: Table[ (Id, V) ],
            edges: Table[ (Id, Id, E) ])
  // Table Views -----------------
  def vertices: Table[ (Id, V) ]
  def edges: Table[ (Id, Id, E) ]
  def triplets: Table [ ((Id, V), (Id, V),
  // Transformations ------------------------------
  def reverse: Graph[V, E]
  def subgraph(pV: (Id, V) => Boolean,
               pE: Edge[V,E] => Boolean): Graph[V,E]
  def mapV(m: (Id, V) => T ): Graph[T,E]
  def mapE(m: Edge[V,E] => T ): Graph[V,T]
  // Joins ----------------------------------------
  def joinV(tbl: Table [(Id, T)]): Graph[(V, T), E ]
  def joinE(tbl: Table [(Id, Id, T)]): Graph[V, (E, T)]
  // Computation ----------------------------------
  def mrTriplets(mapF: (Edge[V,E]) => List[(Id, T)],
                 reduceF: (T, T) => T): Graph[T, E]
}
```

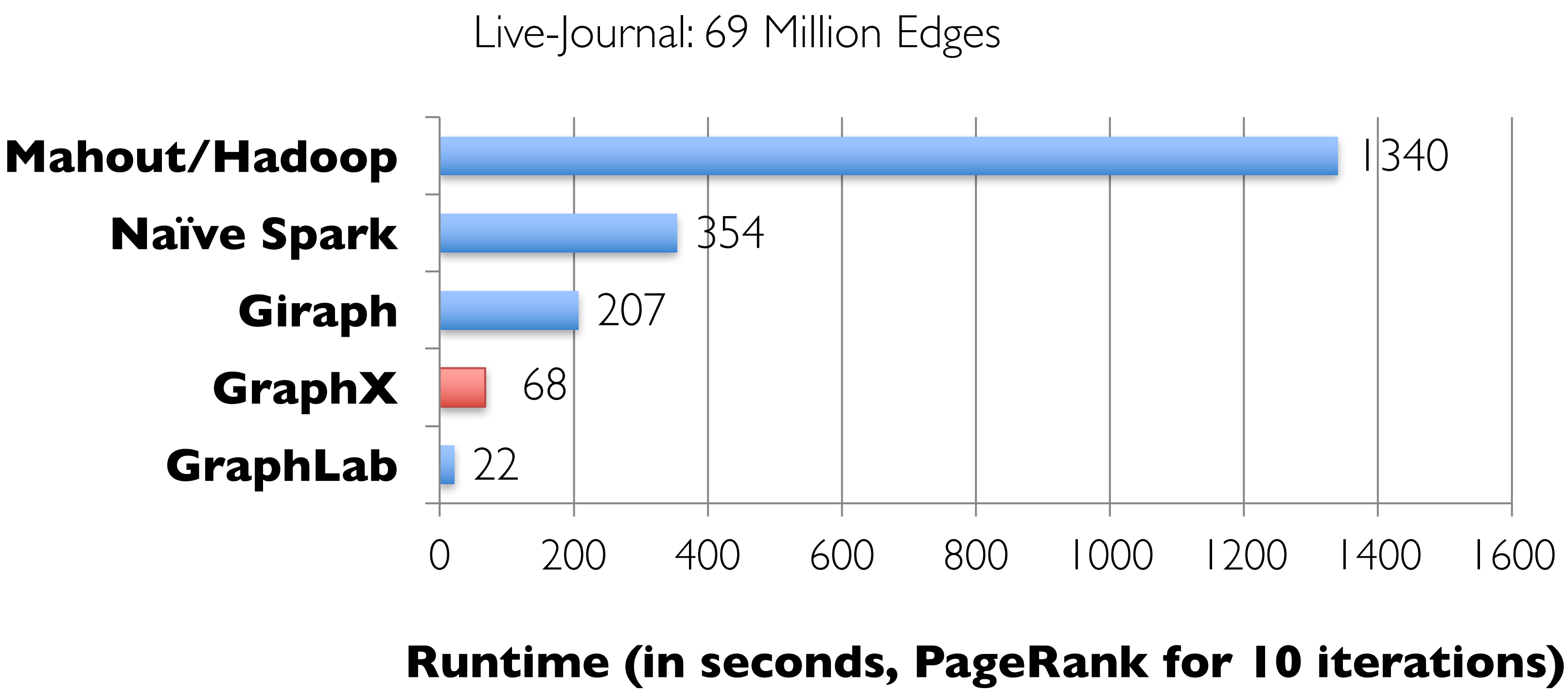
Idea 2: mrTriplets: low-level routine similar to scatter-gather-apply.

Evolved to <u>aggregateNeighbors</u>, <u>aggregateMessages</u>

The GraphX Stack (Lines of Code)



Performance Comparisons



GraphX is roughly 3x slower than GraphLab

Wrapup

Summary

- Large immutable data structures on (distributed) disk, processing by sweeping through then and creating new data structures:
  - stream-and-sort, Hadoop, PIG, Hive, …
- Large immutable data structures in distributed memory:
  - Spark – distributed tables
- Large mutable data structures in distributed memory:
  - parameter server: structure is a *hashtable*
  - Pregel, GraphLab, GraphChi, GraphX: structure is a graph

Summary

- APIs for the various systems vary in detail but have a similar flavor
  - Typical algorithms iteratively update vertex state
  - Changes in state are communicated with messages which need to be aggregated from neighbors
- Biggest wins are
  - on problems where graph is fixed in each iteration, but vertex data changes
  - on graphs small enough to fit in (distributed) memory

Some things to take away

- Platforms for iterative operations on graphs
  - GraphX: if you want to integrate with Spark
  - GraphChi: if you don't have a cluster
  - GraphLab/Dato: if you don't need free software and performance is crucial
  - Pregel: if you work at Google
  - Giraph, Signal/collect, … ??
- Important differences
  - Intended architecture: shared-memory and threads, distributed cluster memory, graph on disk
  - How graphs are partitioned for clusters
  - If processing is synchronous or asynchronous