PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

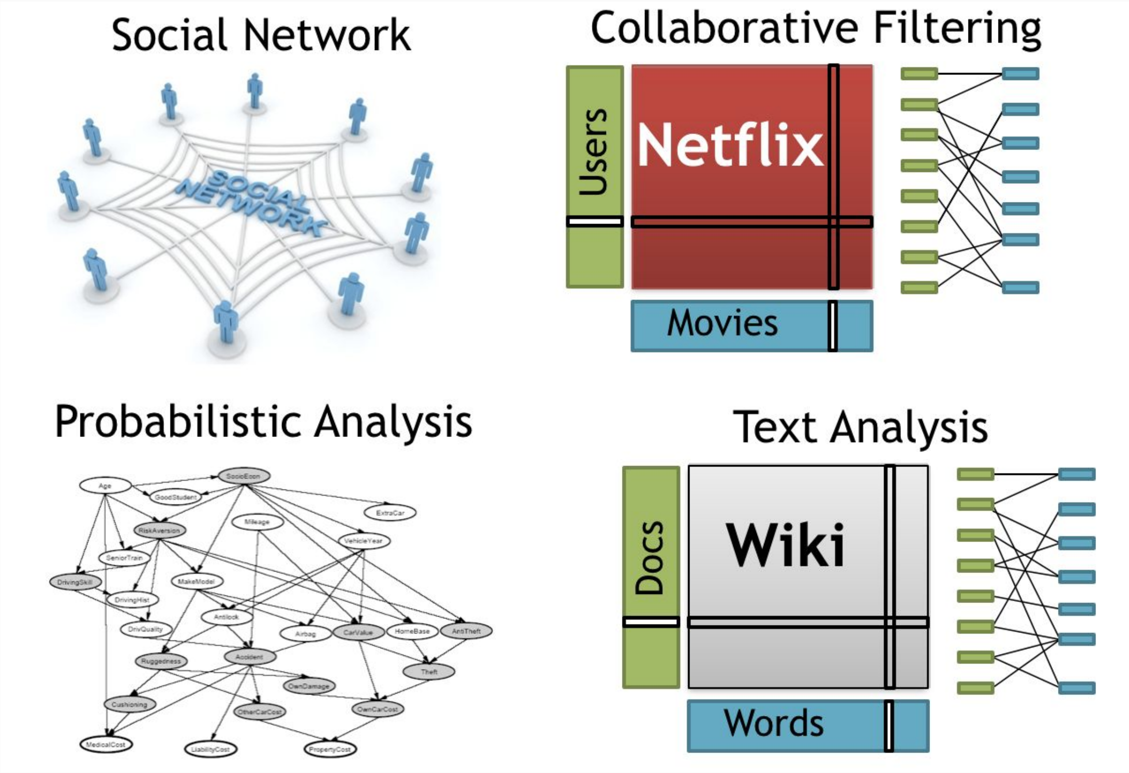
Eric Hsin, Kevin Chen, Wen Jen Hsieh

Oct 23

Background

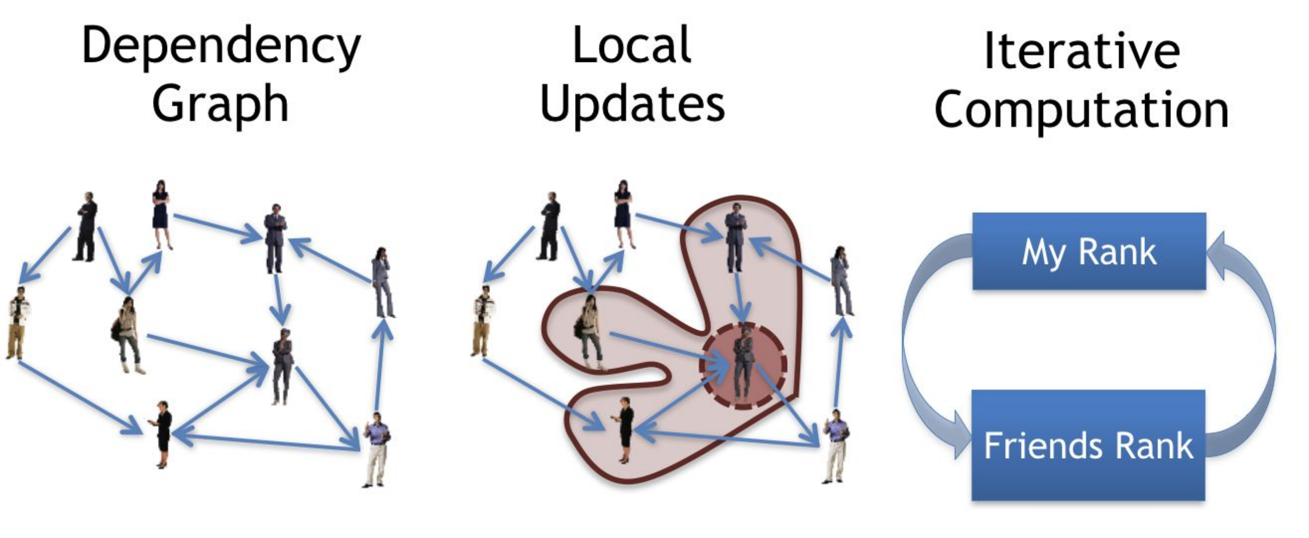
- Rapidly growing datasets
- Machine learning and Data mining Problems
    - Sparse data dependencies
    - Local computations
    - Iterative updates
- We used graph-parallel abstractions to describe large-scale of data

Graphs are Everywhere



Joseph Gonzalez et. al. GraphLab

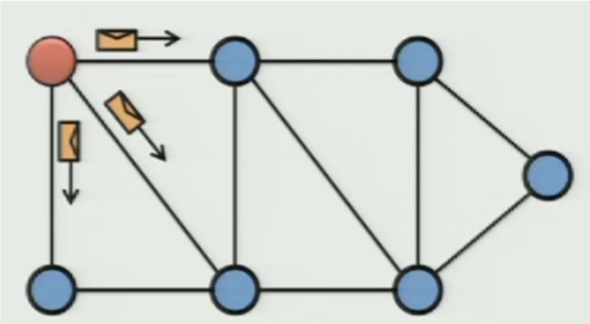
Properties of Computation on Graphs

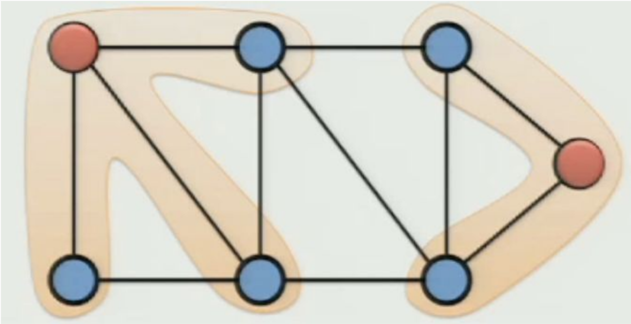


Slides from: http://www.sfbayacm.org/event/graphlab-distributed-abstraction-machine-learning-cloud

Graph-Parallel abstraction

- A user-defined Vertex-Program runs on each vertex
- Graph constrains interaction along edges
  - Using messages: Pregel
  - Using shared states: GraphLab
- Parallelism: run multiple vertex programs simultaneously



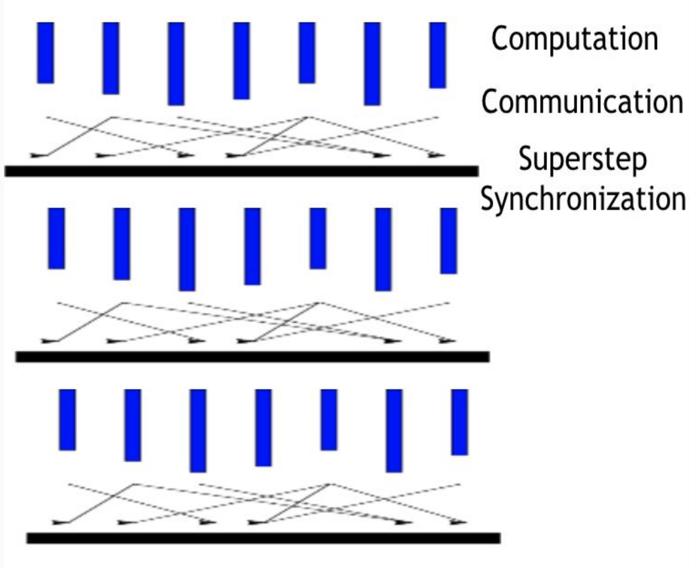


Joseph Gonzalez et. al. PowerGraph

Pregel

- Bulk-Synchronous: All vertices update in parallel
  - Compute
  - Communicate
  - Barrier

```
Message combiner(Message m1, Message m2) :
  return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
  float total = msg.value();
  vertex.val = 0.15 + 0.85*total;
  foreach(nbr in out_neighbors) :
     SendMsg(nbr, vertex.val/num_out_nbrs);
```



GraphLab

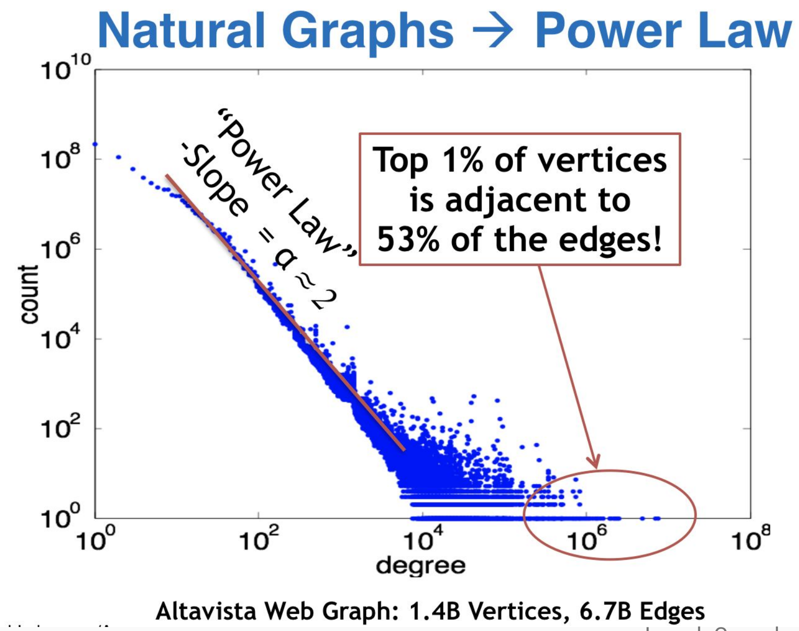
- Asynchronous
- Shared Memory
  - Each vertex-program may directly access information
- Lock on all neighbors to prevent adjacent vertex-program concurrency
  - Fine-grained locking protocol

```
void GraphLabPageRank(Scope scope) :
  float accum = 0;
  foreach (nbr in scope.in_nbrs) :
      accum += nbr.val / nbr.nout_nbrs();
  vertex.val = 0.15 + 0.85 * accum;
```

Issues

- Challenges of high-degree vertices
  - Natural graphs have skewed power-law degree distribution
  - Lead to a few highly-loaded servers

$$\mathbf{P}(d) \propto d^{-\alpha}$$



Joseph Gonzalez et. al. GraphLab

High-Degreed Vertex

- Workload Imbalance
- Partitioning
  - Pregel and GraphLab both use hash-based (random) partitioning
- Communication
  - Pregel: Sending many identical messages
  - GraphLab: Locking scheme is unfair to high degree vertices
- Storage
  - High-degree vertices can exceed the memory capacity of a single machine
- Computation
  - Multiple vertex-programs may execute in parallel
  - Existing abstractions do not parallelize within individual vertex-programs

GAS decomposition

- Gather: Information about adjacent vertices and edges is collected

$$\Sigma \leftarrow \bigoplus_{v \in \mathbf{Nbr}[u]} g\left(D_u, D_{(u,v)}, D_v\right).$$

- Apply: Update the value of the central vertex

$$D_u^{\text{new}} \leftarrow a\left(D_u, \Sigma\right).$$

- Scatter
  - Update the data on adjacent edges.
  - Signal neighbors for future computation (if needed)

$$\forall v \in \mathbf{Nbr}[u]: \quad \left(D_{(u,v)}\right) \leftarrow s\left(D_u^{\text{new}}, D_{(u,v)}, D_v\right).$$

PowerGraph Abstraction

- From GraphLab
  - Borrow the shared-memory view
- From Pregel
  - Borrow the commutative, associative gather concept
- GASVertexProgram interface

interface *GASVertexProgram*(u) {

  // Run on gather_nbrs(u)

  gather$(D_u, D_{(u,v)}, D_v) \to Accum$

  sum$(Accum$ left, $Accum$ right$) \to Accum$

  apply$(D_u, Accum) \to D_u^{\text{new}}$

  // Run on scatter_nbrs(u)

  scatter$(D_u^{\text{new}}, D_{(u,v)}, D_v) \to (D_{(u,v)}^{\text{new}}, Accum)$

}

PowerGraph Abstraction (cont'd)

- Support both parallel bulk synchronous and asynchronous model
- Delta Caching
  - Avoid unnecessary gather computation
  - A cache of the accumulator **au**
  - Abelian group
    - If accumulator type has commutative and associative sum (+) and Inverse (-)

$$\Delta a = g(D_u, D_{(u,v)}^{\text{new}}, D_v^{\text{new}}) - g(D_u, D_{(u,v)}, D_v).$$

Algorithm 1: Vertex-Program Execution Semantics

Input: Center vertex u
if *cached accumulator* a_u *is empty* **then**
  foreach *neighbor* v *in gather_nbrs*(u) **do**
    $a_u \leftarrow \text{sum}(a_u, \text{gather}(D_u, D_{(u,v)}, D_v))$
  end
end
$D_u \leftarrow \text{apply}(D_u, a_u)$
foreach *neighbor* v *scatter_nbrs*(u) **do**
  $(D_{(u,v)}, \Delta a) \leftarrow \text{scatter}(D_u, D_{(u,v)}, D_v)$
  if a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \text{sum}(a_v, \Delta a)$
  else $a_v \leftarrow \text{Empty}$
end

PowerGraph Abstraction (cont'd)

PageRank

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v.rank / #outNbrs(v)
sum(a, b): return a + b
apply(D_u, acc):
  rnew = 0.15 + 0.85 * acc
  D_u.delta = (rnew - D_u.rank)/
              #outNbrs(u)
  D_u.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(D_u,D_(u,v),D_v):
  if(|D_u.delta|>ε) Activate(v)
  return delta
```

Greedy Graph Coloring

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

The PageRank support delta caching in the gather phase. But Greedy Graph Coloring doesn't

Distributed Graph Placement

- Common Approach
  - Place a graph via p-way **edge-cut**, which performs poorly on power-law graphs.
  - Each cut edge leads to storage and network overheads.
    - Network - update   information
    - Storage - maintain edge information and copies of neighbors, **ghost**

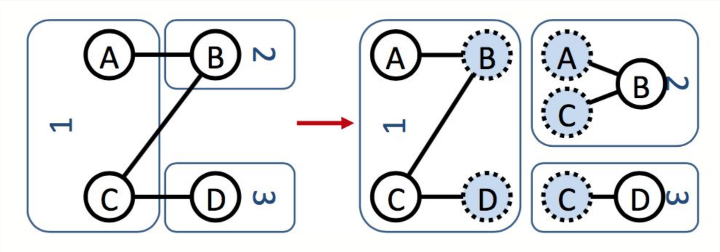


Fig. 4 (a): An example of three-way edge-cut, where the amount of ghost is even larger than vertices

Distributed Graph Placement (cont'd)

- Balanced p-way **Vertex-Cut**
  - The GAS abstraction allows a vertex to be spread in machines, called a **mirror**.
  - An edge lies only in one machine.

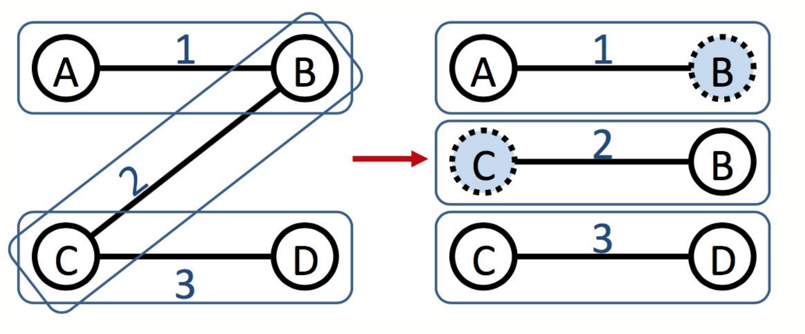


Fig. 4 (b): An example of three-way vertex-cut.

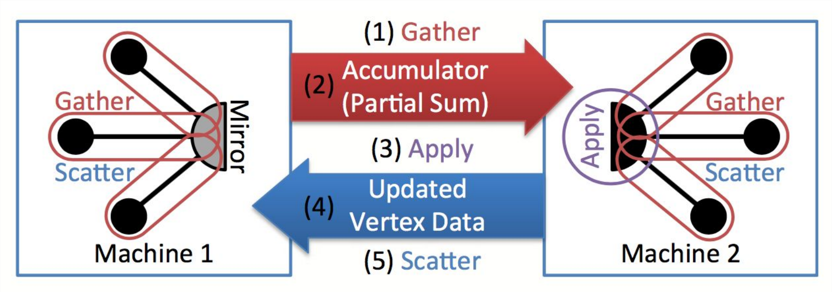


Fig. 5: The communication pattern between two replications.

Distributed Graph Placement (cont'd)

- To minimize the vertex-cut **replication factor**, we would like to achieve the following objective function.
- **Replication factor:** copies per vertex

$$\min_{A} \frac{1}{|V|} \sum_{v \in V} |A(v)|$$

$$\text{s.t.} \quad \max_{m} |\{e \in E \mid A(e) = m\}|, < \lambda \frac{|E|}{p}$$

Eq. 1: The objective function of a nice vertex-cut,
where lambda is a parameter slightly larger than 1

Distributed Graph Placement (cont’d)

- Method1: Random Vertex-Cuts
  - Expected **replication factor** is better than that of the edge-cut
  - **Same partition is always better than edge-cut**
  - Theoretically better

$$\mathbb{E}\left[\frac{|Edges\ Cut|}{|V|}\right] = \left(1-\frac{1}{p}\right)\mathbb{E}[\mathbf{D}[v]] = \left(1-\frac{1}{p}\right)\frac{\mathbf{h}_{|V|}(\alpha-1)}{\mathbf{h}_{|V|}(\alpha)}, \quad (5.2)$$

$$\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$$

Theorem 1: the expected replication factor of edge-cut

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}\left(\frac{p-1}{p}\right)^{d} d^{-\alpha}, \quad (5$$

$$= \frac{p}{|V|}\sum_{v\in V}\left(1-\left(1-\frac{1}{p}\right)^{\mathbf{D}[v]}\right).$$

Theorem 2: the expected replication factor of vertex-cut

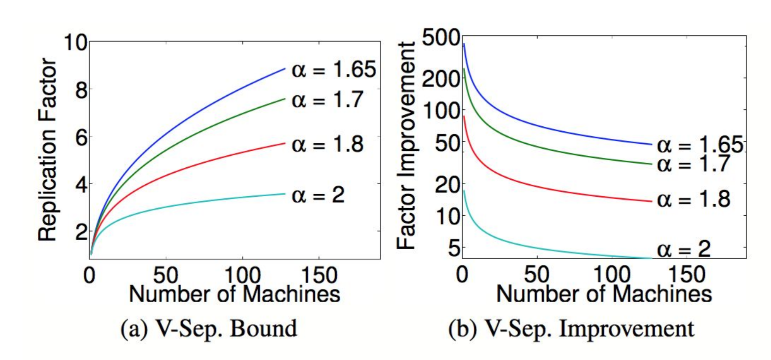


Fig. 6: (a) Replication factor over machines, (b) Theoretical improvement adopting vertex-cut

Distributed Graph Placement (cont'd)

- Theoretically, it can be proved that vertex-cut is better
- Same partition is always better than edge-cut

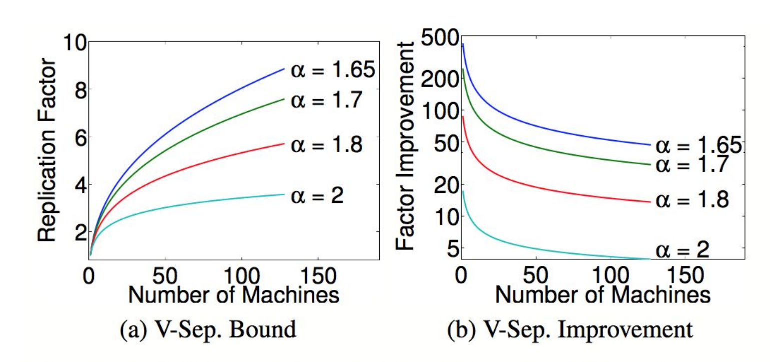


Fig. 6: (a) Replication factor over machines, (b) Theoretical improvement adopting vertex-cut

Distributed Graph Placement (cont'd)

- Method 2: Greedy Vertex-Cuts
  - The method proposed an objective function with conditional expectation and the greedy policy accordingly.
  - Two distributed implementations are proposed and compared
    - Coordinated: w distributed table
    - Oblivious: w/o distributed table

$$\arg\min_{k} \mathbb{E}\left[\sum_{v\in V} |A(v)| \,\middle|\, A_i, A(e_{i+1}) = k\right]$$

Eq. 2: The objective function of the greedy vertex-cut given previous cuts as conditions.

Case 1: If $A(u)$ and $A(v)$ intersect, then the edge should be assigned to a machine in the intersection.

Case 2: If $A(u)$ and $A(v)$ are not empty and do not intersect, then the edge should be assigned to one of the machines from the vertex with the most unassigned edges.

Case 3: If only one of the two vertices has been assigned, then choose a machine from the assigned vertex.

Case 4: If neither vertex has been assigned, then assign the edge to the least loaded machine.

Table. 1: The greedy policy given eq.2 and theorem.2

Distributed Graph Placement (cont'd)

- Methods comparison

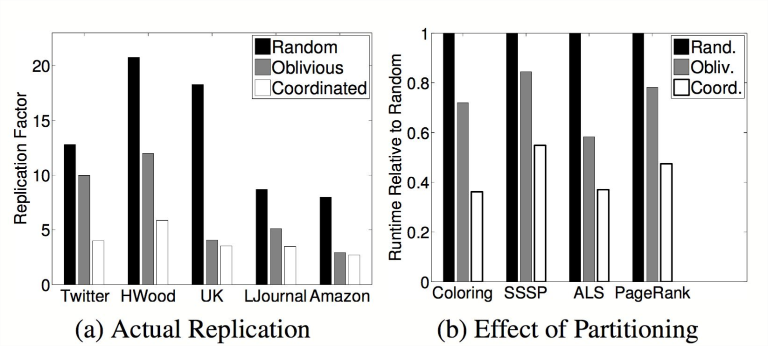


Fig. 7: Replication factors over algorithms

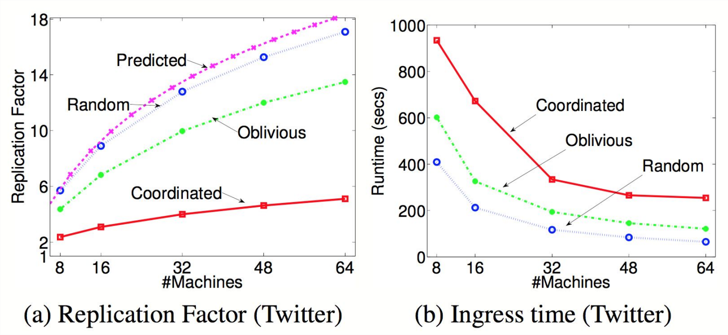


Fig. 8: Replication factors over machine numbers

Abstraction Comparison

- Experiment
  - Pregel, GraphLab, PowerGraph
  - PageRank on five synthetically constructed power-law graph
    - Ten-million vertices
    - **alpha 1.8 - 2.2**
  - Eight-node Linux cluster
  - **Piccolo** is used as a proxy implementation for Pregel due to its memory limitations

Abstraction Comparison (cont'd)

- Computation Imbalance
  - **Standard deviation of worker-per-iteration** as a measure of imbalance
  - GraphLab performs worse in fan-in due to lock on adjacent vertices
  - Pregel performs worse in fan-out due to communications across multiple machines

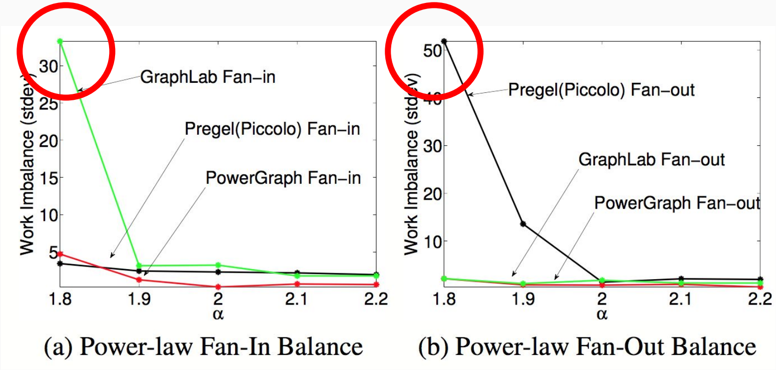


Fig. 9: Work imbalance of fan-in and fan-out

Abstraction Comparison (cont'd)

- Communication Imbalance
  - Pregel communicates more on fan-out due to message sending
  - PowerGraph & GraphLab both "expose" updated vertex values to neighbors, without considering the direction of edges, leading to Comm invariant to alpha.
  - Furthermore, PowerGraph is significantly better because of vertex-cut.

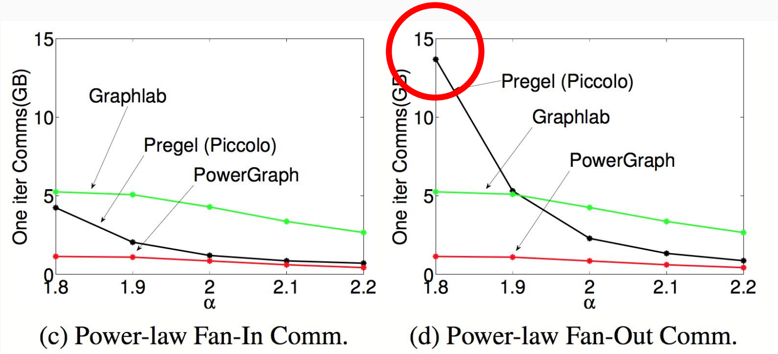


Fig. 9: Communication imbalance of fan-in and fan-out

Abstraction Comparison (cont'd)

- Runtime
  - Runtime is significantly affected by communication.
  - The limited effect from work imbalance derived from the lightweight nature of PageRank. More complex algorithms are expected to bring out the effect.
  - Greedy vertex-cut is better.

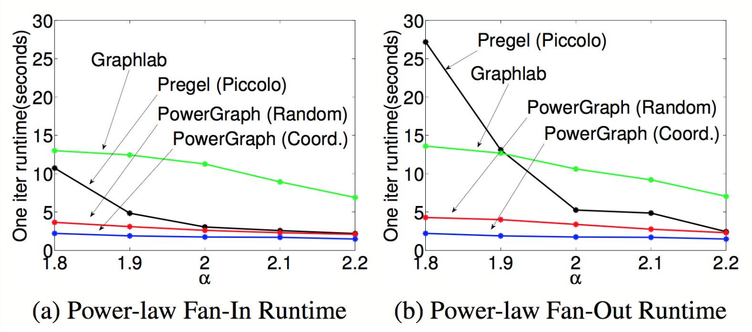


Fig. 10: Runtime of fan-in and fan-out

Implementation & Evaluation

- Synchronous Engine
    - Greedy partitioning increase loading overhead while can still be better if more than 20 iterations are applied.

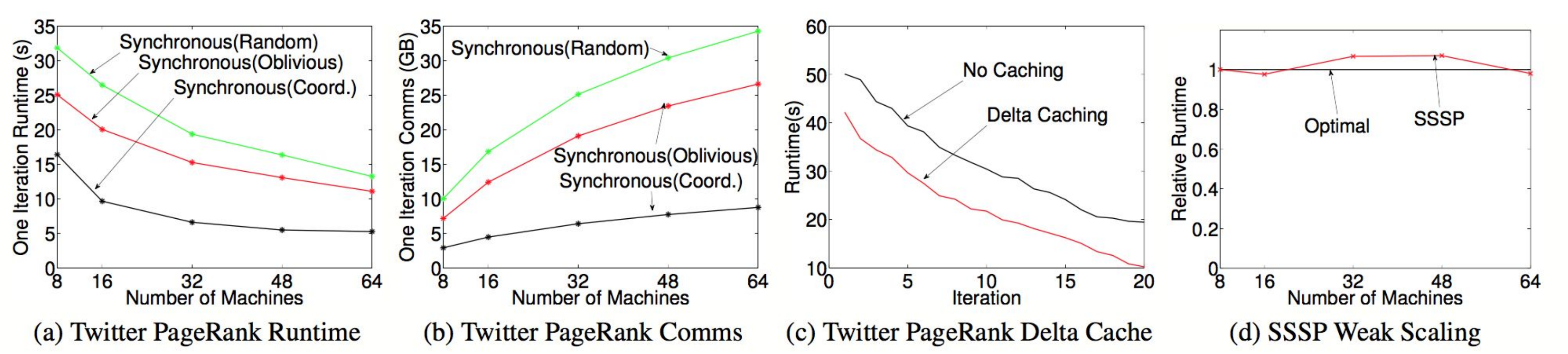


Fig. 11: Experimental results. (a)-(c) evaluate synchronous engine with Twitter PageRank. Iteration here denotes superstep. (d) proves weak scalability with SSSP (ten-million vertices per machine).

Implementation & Evaluation (cont'd)

- Asynchronous Engine
  - 4 States: **INACTIVE, GATHER, APPLY, SCATTER**
- Async. Serializable Engine
  - Chandy-Misra solution

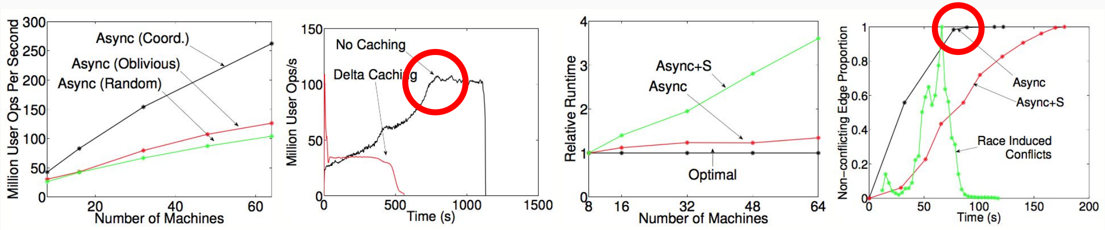


Fig. 12: Experimental results. (a)-(b) performs experiments on Twitter PageRank. (c)-(d) performs experiments on coloring, where five-million vertices per machine is applied.

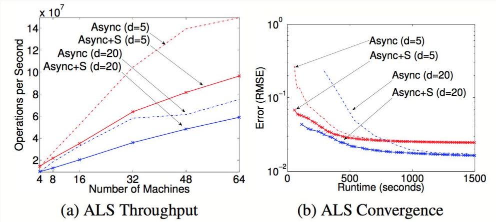


Fig. 13: ALS algorithm, where we can see that Async+S converges faster.

Implementation & Evaluation (cont'd)

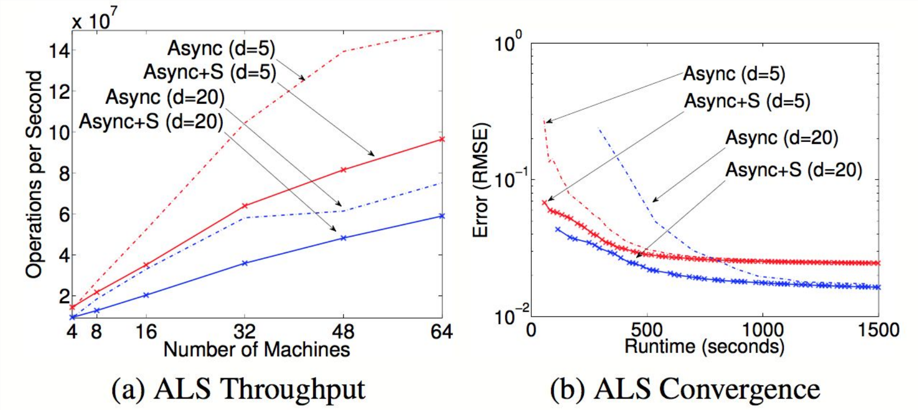


Fig. 13: ALS algorithm is applied to categorize documents, where d denotes the number of topics. We can see that Async+S converges faster.

Implementation & Evaluation (cont'd)

- Fault-Tolerance: Snapshots
  - Sync: between super step
  - Async: suspend execution
- MLDM Applications
  - The state-of-the-art of LDA is heavily optimized

PageRank	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [22]	198s	–	1.1B	50x8
Spark [37]	97.4s	40M	1.5B	50x2
Twister [15]	36s	50M	1.4B	64x4
PowerGraph (Sync)	3.6s	40M	1.5B	64x8

Triangle Count	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [36]	423m	40M	1.4B	1636x?
PowerGraph (Sync)	1.5m	40M	1.4B	64x16

LDA	Tok/sec	Topics	System
Smola et al. [34]	150M	1000	100x8
PowerGraph (Async)	110M	1000	64x16

Tbl. 2: MLDL application experiments against multiple abstractions / systems

Conclusions

- **GAS** model
  - Allows graph factorization, leading to better partitioning
  - Delta-caching leads to better runtime
- **Vertex-cut**
  - Theoretically relates to power-law constant, and better than edge-cut
  - Performs better using Greedy policy
- Implementation
  - **Async+S** has a smaller throughput while converges faster in certain MLDM applications

GraphX

An embedded graph processing framework built on top of Apache Spark

Niche

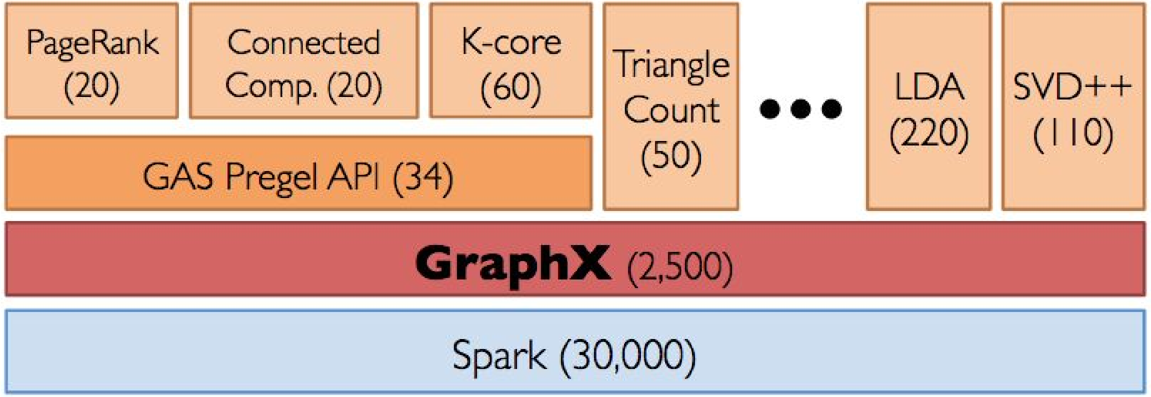


Figure 1: **GraphX** is a thin layer on top of the Spark general-purpose dataflow framework (lines of code).

Contribution

1. Graph as a collection in Spark
2. Better vertex cutting
3. Optimizations.
4. Test and benchmarking

Different processing paradigm

Graph processing abstraction:

- Pregel, GraphLab, PowerGraph, etc
- Difficult to cooperate with unstructured data.
- Favor Snapshot recovery over fault-tolerance.

General-purpose distributed dataflow framework:

- Spark, Dryad.
- Challenging in implementation.
- Does not leverage the common patterns in iterative graph algorithm.

Why spark

1. RDD keeps data in memory
2. RDD permits user-defined partitioning
3. Lineage for recovery

Assumptions

Graph parallel abstraction

- Iteratively apply UDF to vertex
- Only work on static graph(no growth, no shrinking)
- Cannot communicate with unconnected vertex

GAS decomposition

- Gather, Apply, Scatter
- Enables vertex partitioning(easier to cut vertex, less mirror,)

Example

```
def PageRank(v: Id, msgs: List[Double]) {
  // Compute the message sum
  var msgSum = 0
  for (m <- msgs) { msgSum += m }
  // Update the PageRank
  PR(v) = 0.15 + 0.85 * msgSum
  // Broadcast messages with new PR
  for (j <- OutNbrs(v)) {
    msg = PR(v) / NumLinks(v)
    send_msg(to=j, msg)
  }
  // Check for termination
  if (converged(PR(v))) voteToHalt(v)
}
```

Graphs as collections

- Graph -> vertices and edges
- Reusability
- Triplet(vertex, edge, values)
- Gather -> emulated via group-by
- Apply -> emulated via map
- Scatter -> emulated via join

Triplet(for joining)

```
CREATE VIEW triplets AS
SELECT s.Id, d.Id, s.P, e.P, d.P
FROM edges AS e
JOIN vertices AS s JOIN vertices AS d
ON e.srcId = s.Id AND e.dstId = d.Id
```

Listing 3: **Constructing Triplets in SQL**: The column `P` represents the properties in the vertex and edge property collections.

Example: Pregel on GraphX

```
class Graph[V, E] {
  // Constructor
  def Graph(v: Collection[(Id, V)],
            e: Collection[(Id, Id, E)])
  // Collection views
  def vertices: Collection[(Id, V)]
  def edges: Collection[(Id, Id, E)]
  def triplets: Collection[Triplet]
  // Graph-parallel computation
  def mrTriplets(f: (Triplet) => M,
      sum: (M, M) => M): Collection[(Id, M)]
  // Convenience functions
  def mapV(f: (Id, V) => V): Graph[V, E]
  def mapE(f: (Id, Id, E) => E): Graph[V, E]
  def leftJoinV(v: Collection[(Id, V)],
      f: (Id, V, V) => V): Graph[V, E]
  def leftJoinE(e: Collection[(Id, Id, E)],
      f: (Id, Id, E, E) => E): Graph[V, E]
  def subgraph(vPred: (Id, V) => Boolean,
      ePred: (Triplet) => Boolean)
    : Graph[V, E]
  def reverse: Graph[V, E]
}
```

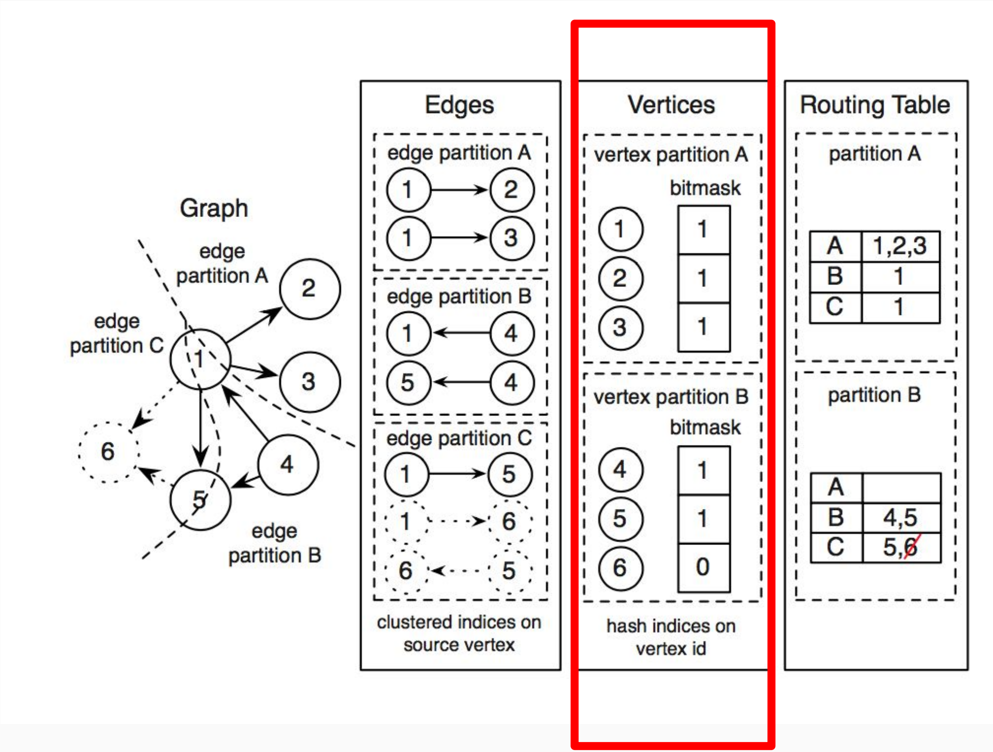
Listing 4: **Graph Operators:** transform vertex and edge collections.

```
def Pregel(g: Graph[V, E],
        vprog: (Id, V, M) => V,
        sendMsg: (Triplet) => M,
        gather: (M, M) => M): Collection[V] = {
  // Set all vertices as active
  g = g.mapV((id, v) => (v, halt=false))
  // Loop until convergence
  while (g.vertices.exists(v => !v.halt)) {
    // Compute the messages
    val msgs: Collection[(Id, M)] =
      // Restrict to edges with active source
      g.subgraph(ePred=(s,d,sP,eP,dP)=>!sP.halt)
      // Compute messages
       .mrTriplets(sendMsg, gather)
    // Receive messages and run vertex program
    g = g.leftJoinV(msgs).mapV(vprog)
  }
  return g.vertices
}
```

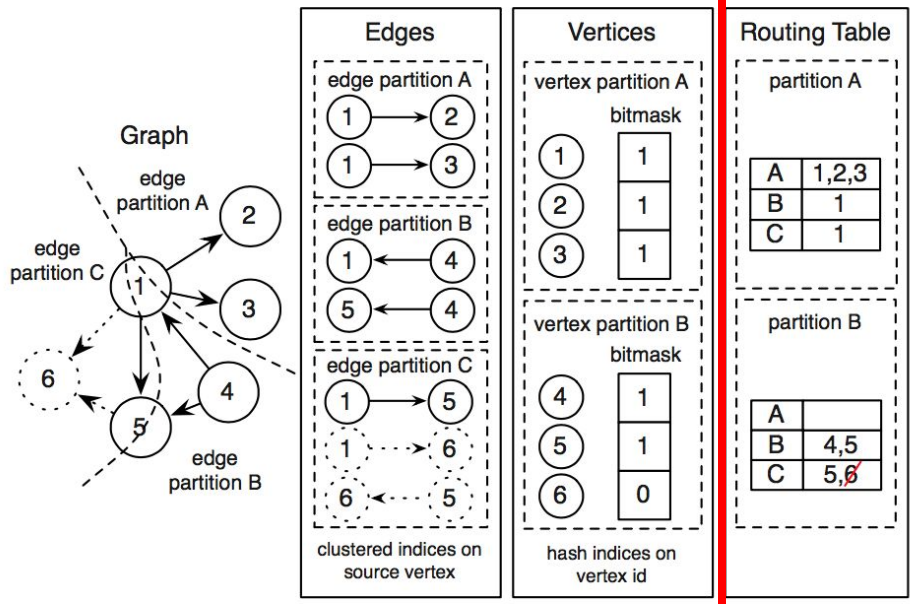
Optimization

- Index reuse
- Multicast join
- Incremental view maintenance
- Filtered index scanning
- Automatic join elimination

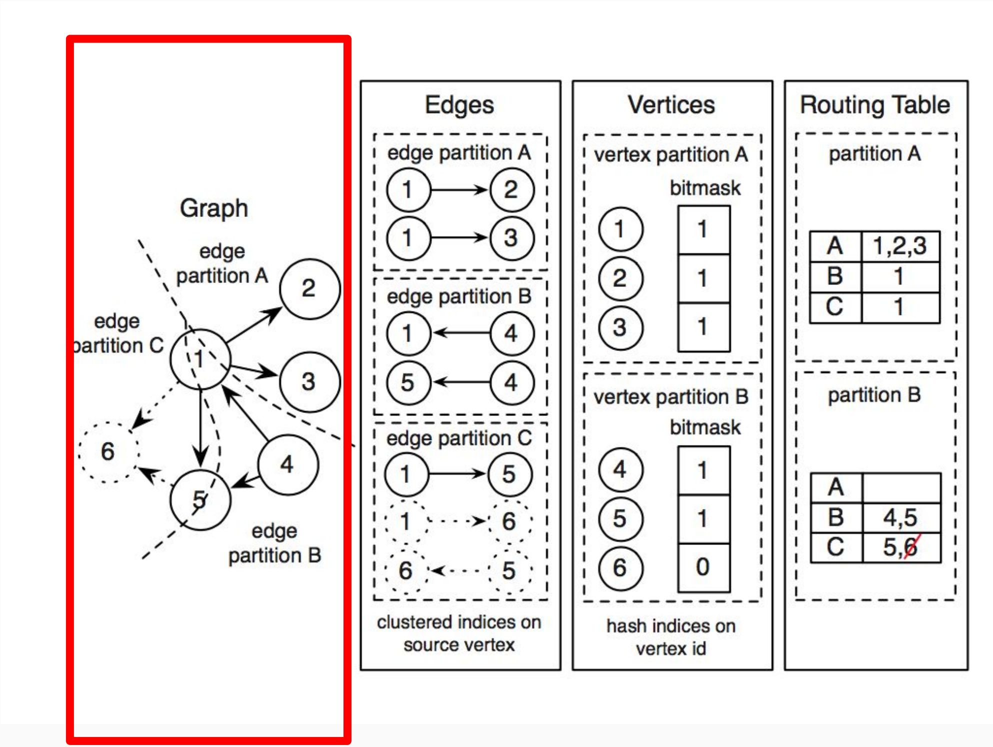
Indices reuse



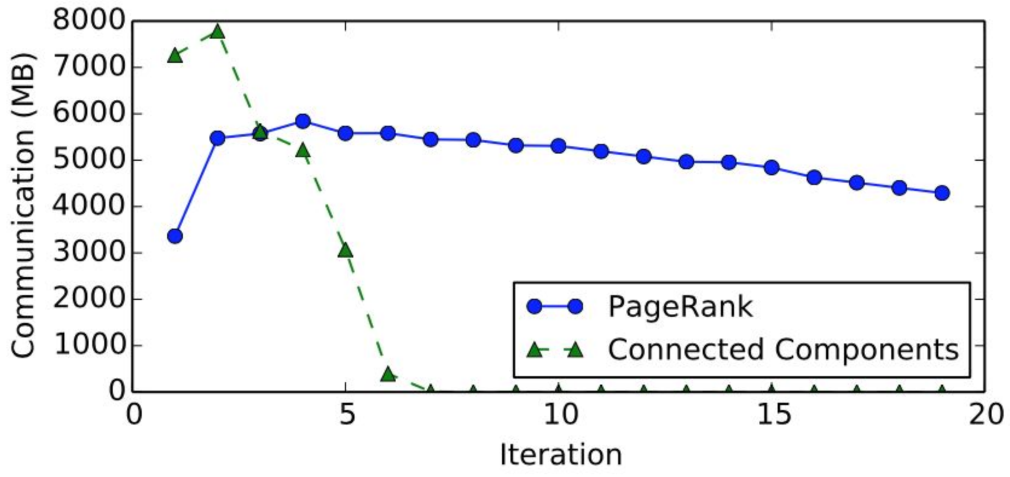
Multicast join



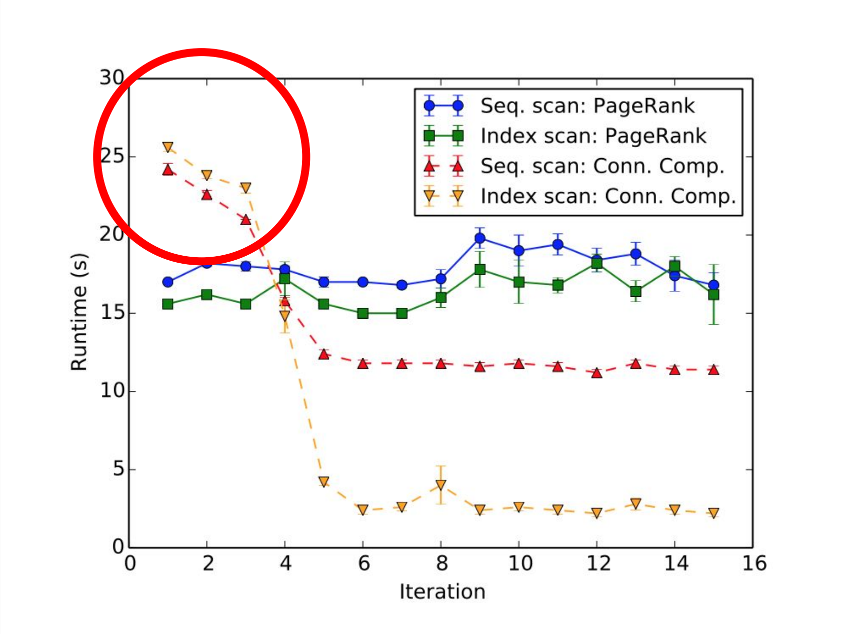
Distributed representation



Incremental maintain view

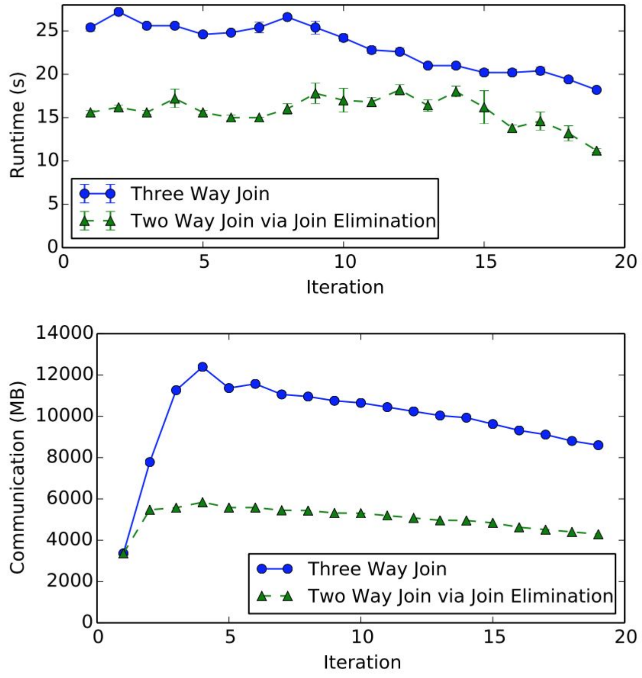


Filtered index scanning

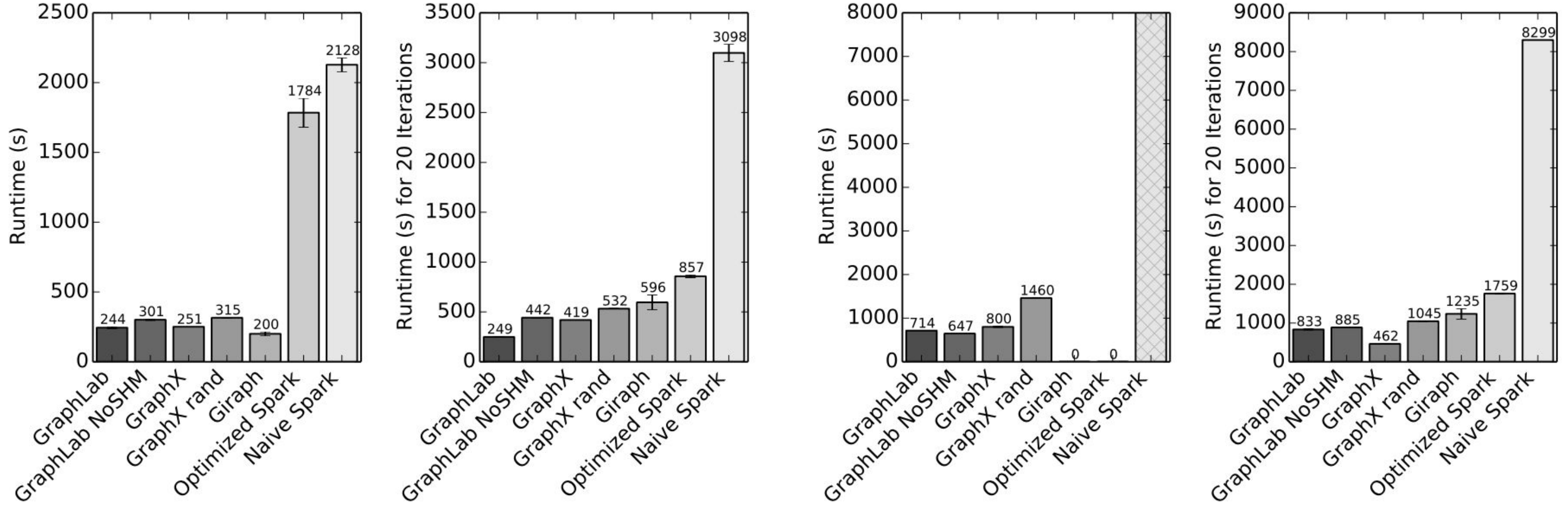


Automatic join elimination

- E.g. When vertex value is not required



Comparison



(a) Conn. Comp. Twitter (b) PageRank Twitter (c) Conn. Comp. uk-2007-05* (d) PageRank uk-2007-05

Other observed phenomenons

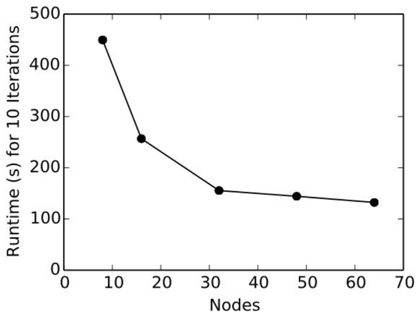


Figure 8: **Strong scaling for PageRank on Twitter (10 Iterations)**

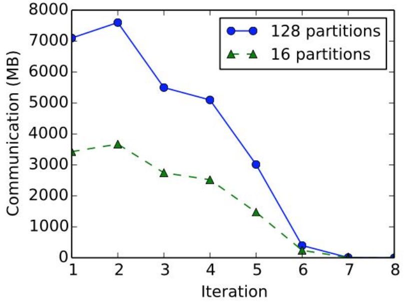
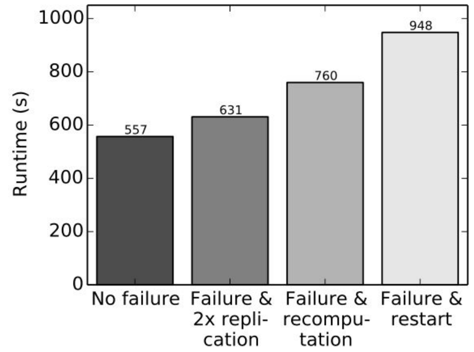


Figure 9: **Effect of partitioning on communication**



Communication is bottleneck

Graph cutting

Fault tolerance

Worth mentioning

- Scalability! But at what COST?
  - Might spend too much effort on scalability, which causes communication / storage overheads.
  - The comparison uses only 20 iterations

Twenty pagerank iterations

System	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Single thread	1	300s	651s

Label propagation to fixed-point (graph connectivity)

System	cores	twitter_rv	uk_2007_05
Spark	128	1784s	8000s+
Giraph	128	200s	8000s+
GraphLab	128	242s	714s
GraphX	128	251s	800s
Single thread	1	153s	417s

Top-notch graph systems v.s. the author's laptop

Twenty pagerank iterations

System	cores	twitter_rv	uk_2007_05
Single thread (simple)	1	300s	651s
Single thread (smarter)	1	110s	256s

Label propagation to fixed-point (graph connectivity)

System	cores	twitter_rv	uk_2007_05
Single thread (simple)	1	153s	417s
Single thread (smarter)	1	15s	30s

And it can be even better