Graph-Based Parallel Computing

William Cohen
Many ML algorithms tend to have

- Sparse data dependencies
- Local computations
- Iterative updates

Examples:
- SGD for logistic regression/MF
- Alternating least squares
- PageRank/Personalized PageRank
- Graph-based SSL (Wed)
- Gibbs sampling (next week)
- Graph analytics: shortest path, triangle counting, ....
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
Pregel (Google, Sigmod 2010)

- Primary data structure is a graph
- Computations are sequence of supersteps, in each of which
  - user-defined function 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
GRAPH ABSTRACTIONS: PREGEL (SIGMOD 2010*)

*Used internally at least 1-2 years before
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!
everyone computes in parallel

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();
};

Figure 3: The Vertex API foundations.
Streaming PageRank: with some long rows

• Repeat until converged:
  – Let $v^{t+1} = c u + (1-c)Wv^t$

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

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

note we need to scan through the graph each time

recap from 3/17
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();
    }
  }
};
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();
    }
};

Another task: single source shortest path
Figure 7: SSSP—1 billion vertex binary tree: varying number of worker tasks scheduled on 300 multicore machines
GRAPH ABSTRACTIONS: SIGNAL/COLLECT (SEMANTIC WEB CONFERENCE, 2010)

Stutz, Strebel, Bernstein, Univ Zurich
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
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

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

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

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

signals are made available in a list and a map
relax “num_iterations” soon
Signal/collection examples

Single-source shortest path

<table>
<thead>
<tr>
<th>initialState</th>
<th>if (isSource) 0 else infinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>return ( \min(\text{oldState}, \min(\text{signals})) )</td>
</tr>
<tr>
<td>signal()</td>
<td>return ( \text{source.state} + \text{edge.weight} )</td>
</tr>
</tbody>
</table>

```
initialState = if (isSource) 0 else infinity
collect() = return \( \min(\text{oldState}, \min(\text{signals})) \)
signal() = return \( \text{source.state} + \text{edge.weight} \)
```
### Signal/collect examples

#### Life

<table>
<thead>
<tr>
<th>initialState</th>
<th>if (isInitiallyAlive) 1 else 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>switch (sum(signals))</td>
</tr>
<tr>
<td></td>
<td>case 0: return 0 // dies of loneliness</td>
</tr>
<tr>
<td></td>
<td>case 1: return 0 // dies of loneliness</td>
</tr>
<tr>
<td></td>
<td>case 2: return oldState // same as before</td>
</tr>
<tr>
<td></td>
<td>case 3: return 1 // becomes alive if dead</td>
</tr>
<tr>
<td></td>
<td>other: return 0 // dies of overcrowding</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state</td>
</tr>
</tbody>
</table>

### PageRank

<table>
<thead>
<tr>
<th>initialState</th>
<th>baseRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>return baseRank + dampingFactor * sum(signals)</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state * edge.weight / sum(edgeWeights(source))</td>
</tr>
<tr>
<td>Function</td>
<td>Code</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>initialState</td>
<td><code>if (isTrainingData) trainingData else avgProbDist</code></td>
</tr>
<tr>
<td>collect()</td>
<td><code>if (isTrainingData) return oldState</code></td>
</tr>
<tr>
<td></td>
<td><code>else return signals.sum.normalise</code></td>
</tr>
<tr>
<td>signal()</td>
<td><code>return source.state</code></td>
</tr>
<tr>
<td>initialState</td>
<td>Set(id)</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>collect()</td>
<td><code>return union(oldState, union(signals))</code></td>
</tr>
<tr>
<td>signal()</td>
<td><code>return source.state</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>initialState</th>
<th>randomColour</th>
</tr>
</thead>
</table>
| collect()     | `if (contains(signals, oldState))`  
|               |     `return randomColorExcept(oldState)`  
|               | `else`  
|               | `return oldState`  |
| signal()      | `return source.state` |

Fig. 9. Vertex colouring (data-graph).
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 \(\rightarrow\) 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
Algorithm 3 Score-guided asynchronous execution

\[\text{ops} := 0\]

\[\text{while } \text{ops} < \text{max_ops and } \exists v \in V(\text{v.scoreSignal()} > s_{\text{threshold}} \text{ or v.scoreCollect()} > c_{\text{threshold}}) \text{ do}\]

\[\text{S := choose subset of } V\]

\[\text{for all } v \in S \text{ parallel do}\]

\[\text{Randomly call either v.doSignal() or v.doCollect() iff respective threshold is reached; increment ops if an operation was executed.}\]

\[\text{end for}\]

\[\text{end while}\]
GRAPH ABSTRACTIONS: GRAPHLAB (UAI, 2010)

Guestrin, Gonzalez, Bikel, etc.

Many slides below pilfered from Carlos or Joey....
Exclusive: Apple acquires Turi in major exit for Seattle-based machine learning and AI startup

- 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 → Apple
CoEM (Rosie Jones, 2005)

Named Entity Recognition Task

Is “Dog” an animal?
Is “Catalina” a place?

<table>
<thead>
<tr>
<th></th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>0.2M</td>
<td>20M</td>
</tr>
<tr>
<td>Large</td>
<td>2M</td>
<td>200M</td>
</tr>
</tbody>
</table>

the dog <X> ran quickly
Australia travelled to <X>
Catalina Island <X> is pleasant

Hadoop 95 Cores 7.5 hrs
CoEM (Rosie Jones, 2005)

<table>
<thead>
<tr>
<th></th>
<th>Number of CPUs</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>95 Cores</td>
<td>7.5 hrs</td>
</tr>
<tr>
<td>GraphLab</td>
<td>16 Cores</td>
<td>30 min</td>
</tr>
</tbody>
</table>

- **6x fewer CPUs!**
- **15x Faster!**
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 $\rightarrow$ Power Law

Top 1% of vertices is adjacent to 53% of the edges!

Altavista Web Graph: 1.4B Vertices, 6.7B Edges

GraphLab group/Aapo
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)
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
Signal/collection examples

**Single-source shortest path**

<table>
<thead>
<tr>
<th>initialState</th>
<th>if (isSource) 0 else infinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>return min(oldState, min(signals))</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state + edge.weight</td>
</tr>
</tbody>
</table>

---

**Graphs:**

- **initial**
  - State 0 connected to state ∞ (infinity)
  - State ∞ connected to state ∞ (infinity)

- **step 1**
  - State 0 connected to state 1
  - State 1 connected to state ∞ (infinity)

- **step 2**
  - State 1 connected to state 2
### Life

<table>
<thead>
<tr>
<th>initialState</th>
<th>if (isInitiallyAlive) 1 else 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>switch (sum(signals))</td>
</tr>
<tr>
<td></td>
<td>case 0: return 0 // dies of loneliness</td>
</tr>
<tr>
<td></td>
<td>case 1: return 0 // dies of loneliness</td>
</tr>
<tr>
<td></td>
<td>case 2: return oldState // same as before</td>
</tr>
<tr>
<td></td>
<td>case 3: return 1 // becomes alive if dead</td>
</tr>
<tr>
<td></td>
<td>other: return 0 // dies of overcrowding</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state</td>
</tr>
</tbody>
</table>

### PageRank

<table>
<thead>
<tr>
<th>initialState</th>
<th>baseRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>return baseRank + dampingFactor * sum(signals)</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state * edge.weight / sum(edgeWeights(source))</td>
</tr>
</tbody>
</table>
### Signal/collect examples

**Co-EM/wvRN/Harmonic fields**

<table>
<thead>
<tr>
<th>Method</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>initialState</code></td>
<td><code>if (isTrainingData) trainingData else avgProbDist</code></td>
</tr>
</tbody>
</table>
| `collect()` | `if (isTrainingData)`  
  `return oldState`
  `else`
  `return signals.sum.normalise` |
| `signal()` | `return source.state` |
PageRank in PowerGraph

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

gather/sum like a group by … reduce or collect

PageRankProgram(i)

- \textbf{Gather}(j \rightarrow i) : \text{return } w_{ji} \ast R[j]
- \textbf{sum}(a, b) : \text{return } a + b;
- \textbf{Apply}(i, \Sigma) : R[i] = \beta + (1 - \beta) \ast \Sigma
- \textbf{Scatter}(i \rightarrow j) :
  - if (R[i] changes) then \textit{activate}(j)

scatter is like a \textit{signal}
Distributed Execution of a PowerGraph Vertex-Program

Gather
Apply
Scatter

Machine 1

Machine 2

Machine 3

Machine 4

\[ \Sigma_1 + \Sigma + \Sigma' \]
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 matters...

![Bar chart showing reduction in runtime for different methods and graph algorithms.](image)

- **PageRank**
  - Random
  - Oblivious
  - Greedy

- **Collaborative Filtering**
  - Random
  - Oblivious
  - Greedy

- **Shortest Path**
  - Random
  - Oblivious
  - Greedy
### Partitioning Performance

**Twitter Graph:** 41M vertices, 1.4B edges

**Cost**
- **Oblivious** balances partition quality and partitioning time.

**Construction Time**
- **Oblivious** balances partition quality and partitioning time.

---

**Better**

---

**GraphLab group/Aapo**
GraphLab’s descendents

- GraphLab
- 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:
    - some algorithms on graph are streamable (i.e., PageRank-Nibble)
    - 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
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

```
1. Load
2. Compute
3. Write
```
PSW: Layout

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

Vertices 1..100
Shard 1

Vertices 101..700
Shard 2

Vertices 701..1000
Shard 3

Vertices 1001..10000
Shard 4

Shards small enough to fit in memory; balance size of shards
PSW: Loading Sub-graph

Load subgraph for vertices 1..100

Vertices 1..100
- Shard 1

Vertices 101..700
- Shard 2

Vertices 701..1000
- Shard 3

Vertices 1001..10000
- Shard 4

Load all in-edges in memory

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

1. Load
2. Compute
3. Write
PSW: Loading Sub-graph

Load subgraph for vertices 101..700

1. Load
2. Compute
3. Write

Load all in-edges in memory

Out-edge blocks in memory
PSW Load-Phase

Only $P$ large reads for each interval.

$P^2$ reads on one full pass.

Interval 1

Shard 1
Shard 2
Shard 3
Shard 4
PSW: Execute updates

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

- In write phase, the blocks are written back to disk
  - Next load-phase sees the preceding writes as 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:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>P (shards)</th>
<th>Preprocessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>live-journal</td>
<td>4.8M</td>
<td>69M</td>
<td>3</td>
<td>0.5 min</td>
</tr>
<tr>
<td>netflix</td>
<td>0.5M</td>
<td>99M</td>
<td>20</td>
<td>1 min</td>
</tr>
<tr>
<td>twitter-2010</td>
<td>42M</td>
<td>1.5B</td>
<td>20</td>
<td>2 min</td>
</tr>
<tr>
<td>uk-2007-05</td>
<td>106M</td>
<td>3.7B</td>
<td>40</td>
<td>31 min</td>
</tr>
<tr>
<td>uk-union</td>
<td>133M</td>
<td>5.4B</td>
<td>50</td>
<td>33 min</td>
</tr>
<tr>
<td>yahoo-web</td>
<td>1.4B</td>
<td>6.6B</td>
<td>50</td>
<td>37 min</td>
</tr>
</tbody>
</table>
On a Mac Mini:

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

**WebGraph Belief Propagation**

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**
  - 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, resilient 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

<table>
<thead>
<tr>
<th>Id</th>
<th>Property (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rxin</td>
<td>(Stu., Berk.)</td>
</tr>
<tr>
<td>Jegonzal</td>
<td>(PstDoc, Berk.)</td>
</tr>
<tr>
<td>Franklin</td>
<td>(Prof., Berk)</td>
</tr>
<tr>
<td>Istoica</td>
<td>(Prof., Berk)</td>
</tr>
</tbody>
</table>

### Edge Property Table

<table>
<thead>
<tr>
<th>SrcId</th>
<th>DstId</th>
<th>Property (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rxin</td>
<td>jegonzal</td>
<td>Friend</td>
</tr>
<tr>
<td>franklin</td>
<td>rxin</td>
<td>Advisor</td>
</tr>
<tr>
<td>istoica</td>
<td>franklin</td>
<td>Coworker</td>
</tr>
<tr>
<td>franklin</td>
<td>jegonzal</td>
<td>PI</td>
</tr>
</tbody>
</table>
Operators

- Table (RDD) operators are inherited from Spark:

  - map
  - filter
  - groupBy
  - sort
  - union
  - join
  - leftOuterJoin
  - rightOuterJoin

  - reduce
  - count
  - fold
  - reduceByKey
  - groupByKey
  - cogroup
  - cross
  - zip

  - sample
  - take
  - first
  - reduceByKey
  - partitionBy
  - mapWith
  - pipe
  - save
  - ...
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),
                          (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]
}
The GraphX Stack
(Lines of Code)

PageRank (5)
Connected Comp. (10)
Shortest Path (10)
SVD (40)
ALS (40)
K-core (51)
Triangle Count (45)
LDA (120)

Pregel (28) + GraphLab (50)

GraphX (3575)

Spark
GraphX is roughly 3x slower than GraphLab
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
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