PREGEL

A System for Large-Scale Graph Processing

The Problem

- Large Graphs are often part of computations required in modern systems (Social networks and Web graphs etc.)
- There are many graph computing problems like shortest path, clustering, page rank, minimum cut, connected components etc. but there exists no scalable general purpose system for implementing them.

Characteristics of the algorithms

- They often exhibit poor locality of memory access.
- Very little computation work required per vertex.
- Changing degree of parallelism over the course of execution.

Refer [1, 2]

Possible solutions

- Crafting a custom distributed framework for every new algorithm.
- Existing distributed computing platforms like MapReduce.
  - These are sometimes used to mine large graphs[3, 4], but often give sub-optimal performance and have usability issues.
- Single-computer graph algorithm libraries
  - Limiting the scale of the graph is necessary
  - BGL, LEDA, NetworkX, JDSL, Standford GraphBase or FGL
- Existing parallel graph systems which do not handle fault tolerance and other issues
  - The Parallel BGL[5] and CGMgraph[6]

Pregel

Google, to overcome, these challenges came up with Pregel.

- Provides scalability
- Fault-tolerance
- Flexibility to express arbitrary algorithms

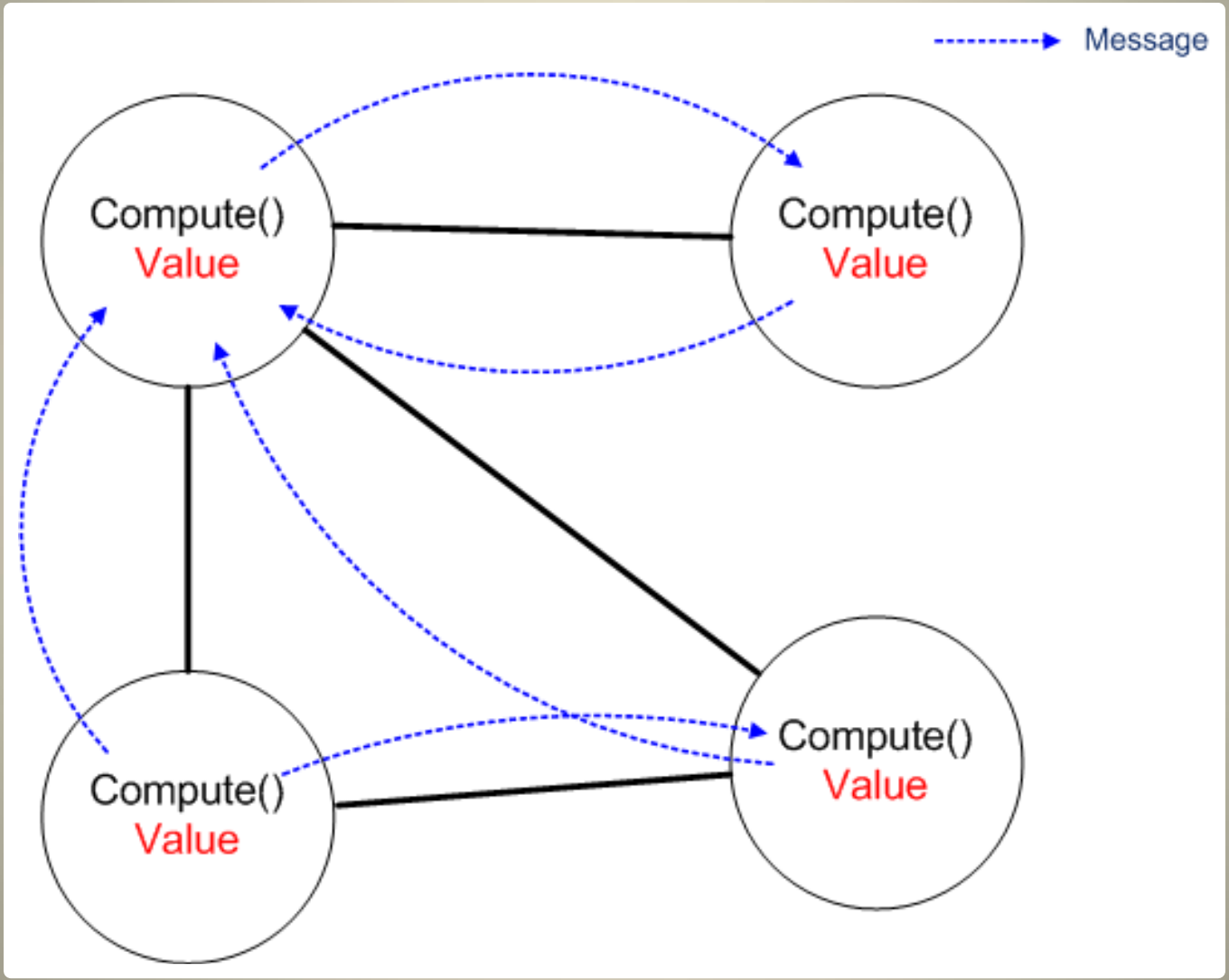
The high level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel model[7].

Message passing model

A pure message passing model has been used, omitting remote reads and ways to emulate shared memory because:

1. Message passing model was found sufficient for all graph algorithms
2. Message passing model performs better than reading remote values because latency can be amortized by delivering larges batches of messages asynchronously.

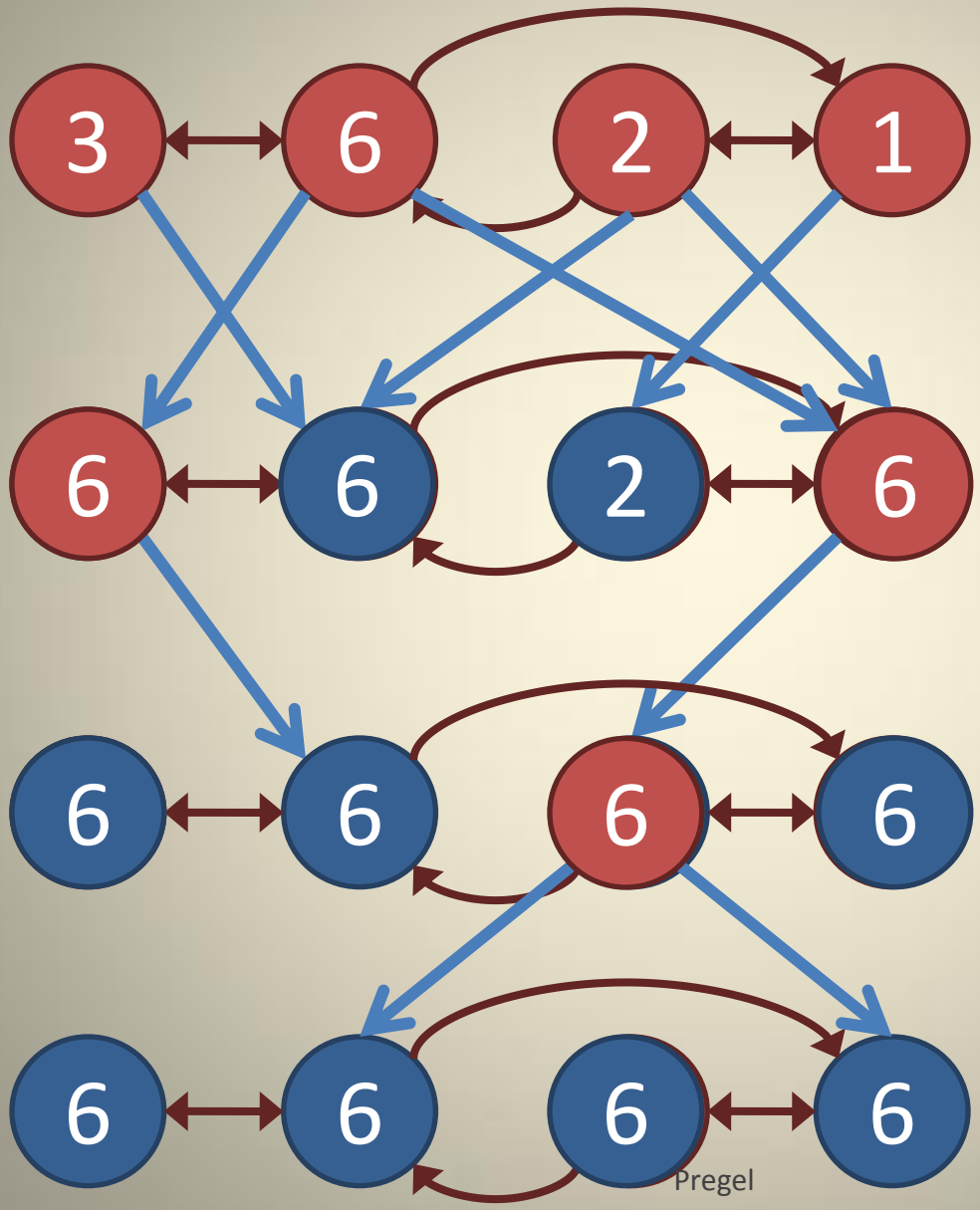
Message passing model



Example

Find the largest value of a vertex in a strongly connected graph

Finding the largest value in a graph



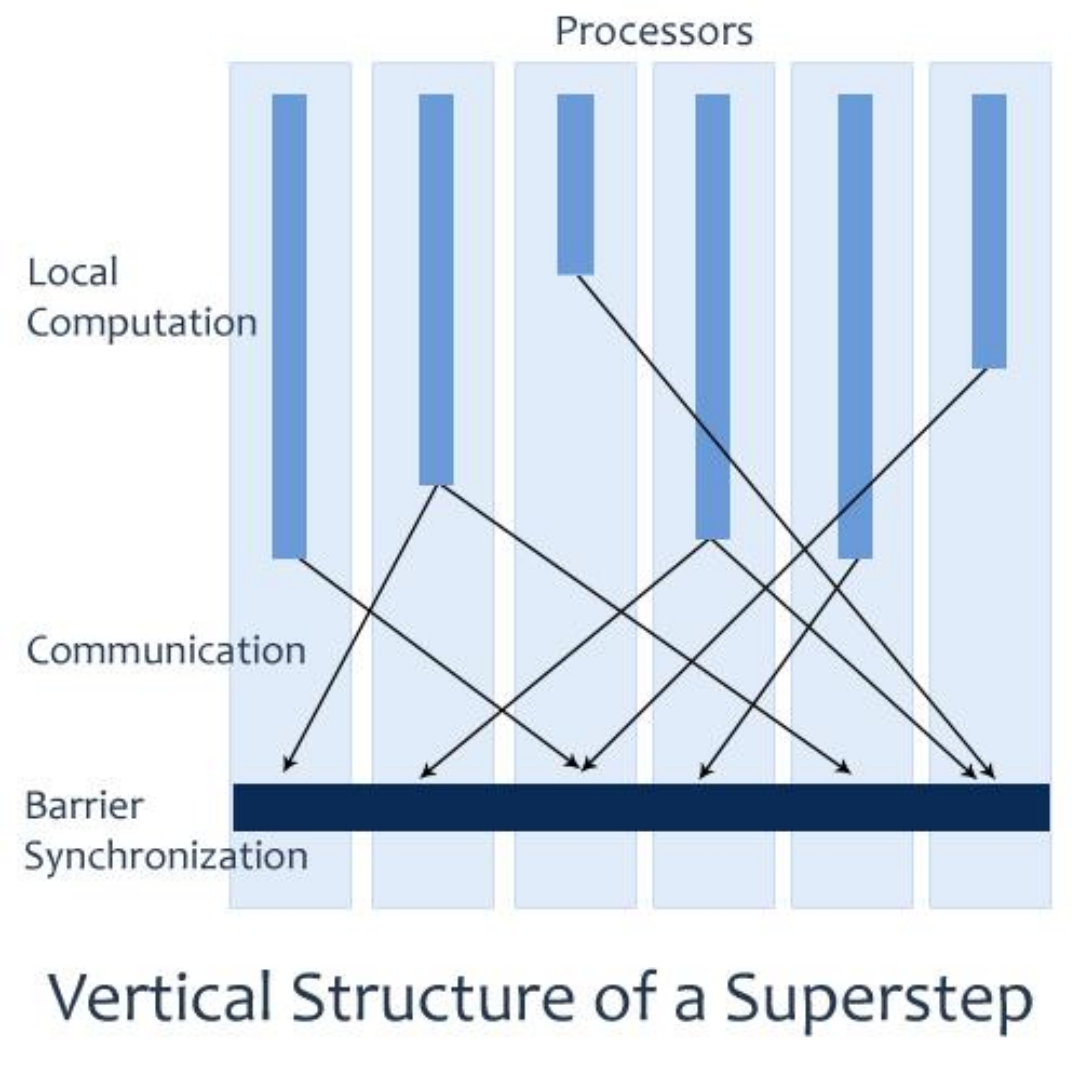
Blue Arrows are messages

Blue vertices have voted to halt

Basic Organization

- Computations consist of a sequence of iterations called **supersteps.**
- During a superstep, the framework invokes a **user defined function for each vertex** which specifies the behavior at a single vertex V and a single Superstep S. The function can:
  - Read messages sent to V in superstep S-1
  - Send messages to other vertices that will be received in superstep S+1
  - Modify the state of V and of the outgoing edges
  - Make topology changes (Introduce/Delete/Modify edges/vertices)

Basic Organization - Superstep



Model Of Computation: Entities

VERTEX

- Identified by a unique identifier.
- Has a modifiable, user defined value.

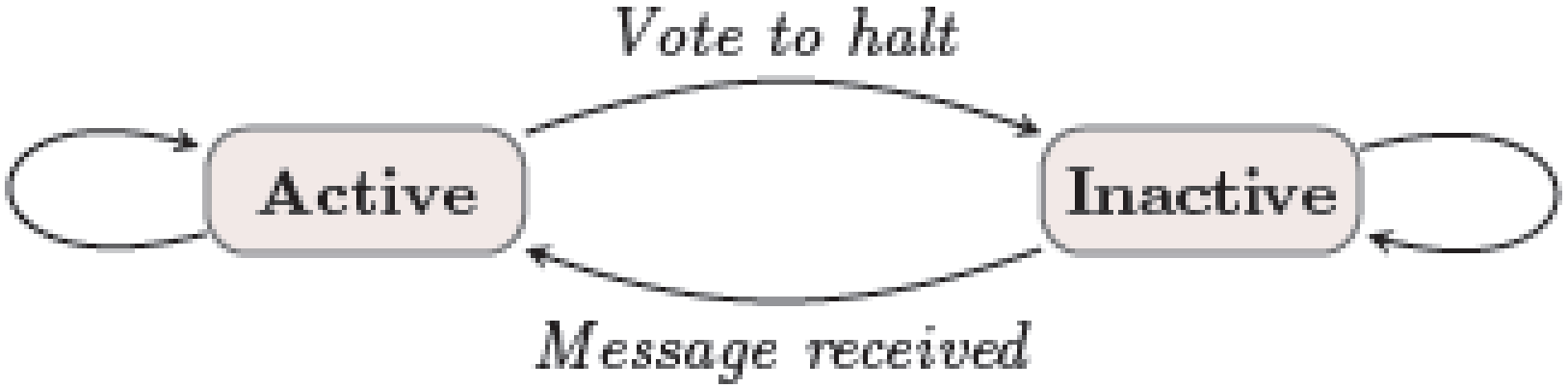
EDGE

- Source vertex and Target vertex identifiers.
- Has a modifiable, user defined value.

Model Of Computation: Progress

- In superstep 0, all vertices are active.
- Only active vertices participate in a superstep.
  - They can go inactive by voting for halt.
  - They can be reactivated by an external message from another vertex.
- The algorithm terminates when all vertices have voted for halt and there are no messages in transit.

Model Of Computation: Vertex



State machine for a vertex

Comparison with MapReduce

Graph algorithms can be implemented as a series of MapReduce invocations but it requires passing of entire state of graph from one stage to the next, which is not the case with Pregel.

Also Pregel framework simplifies the programming complexity by using supersteps.

The C++ API

Creating a Pregel program typically involves subclassing the predefined **Vertex** class.

- The user overrides the virtual **Compute()** method. This method is the function that is computed for every active vertex in supersteps.
- **Compute()** can get the vertex's associated value by **GetValue()** or modify it using **MutableValue()**
- Values of edges can be inspected and modified using the out-edge iterator.

The C++ API – Message Passing

Each message consists of a value and the name of the destination vertex.

- The type of value is specified in the template parameter of the Vertex class.

Any number of messages can be sent in a superstep.

- The framework guarantees delivery and non-duplication but not in-order delivery.

A message can be sent to any vertex if it's identifier is known.

The C++ API – Pregel Code

Pregel Code for finding the max value

```
Class MaxFindVertex
        : public Vertex<double, void, double> {
  public:
        virtual void Compute(MessageIterator* msgs) {
                int currMax = GetValue();
                SendMessageToAllNeighbors(currMax);
                for ( ; !msgs->Done(); msgs->Next()) {
                        if (msgs->Value() > currMax)
                                currMax = msgs->Value();
                }
                if (currMax > GetValue())
                        *MutableValue() = currMax;
                else VoteToHalt();
        }
};
```

The C++ API – Combiners

Sending a message to another vertex that exists on a different machine has some overhead. However if the algorithm doesn't require each message explicitly but a function of it (example sum) then combiners can be used.

This can be done by overriding the **Combine()** method.

-It can be used only for associative and commutative operations.

The C++ API – Combiners

Example:

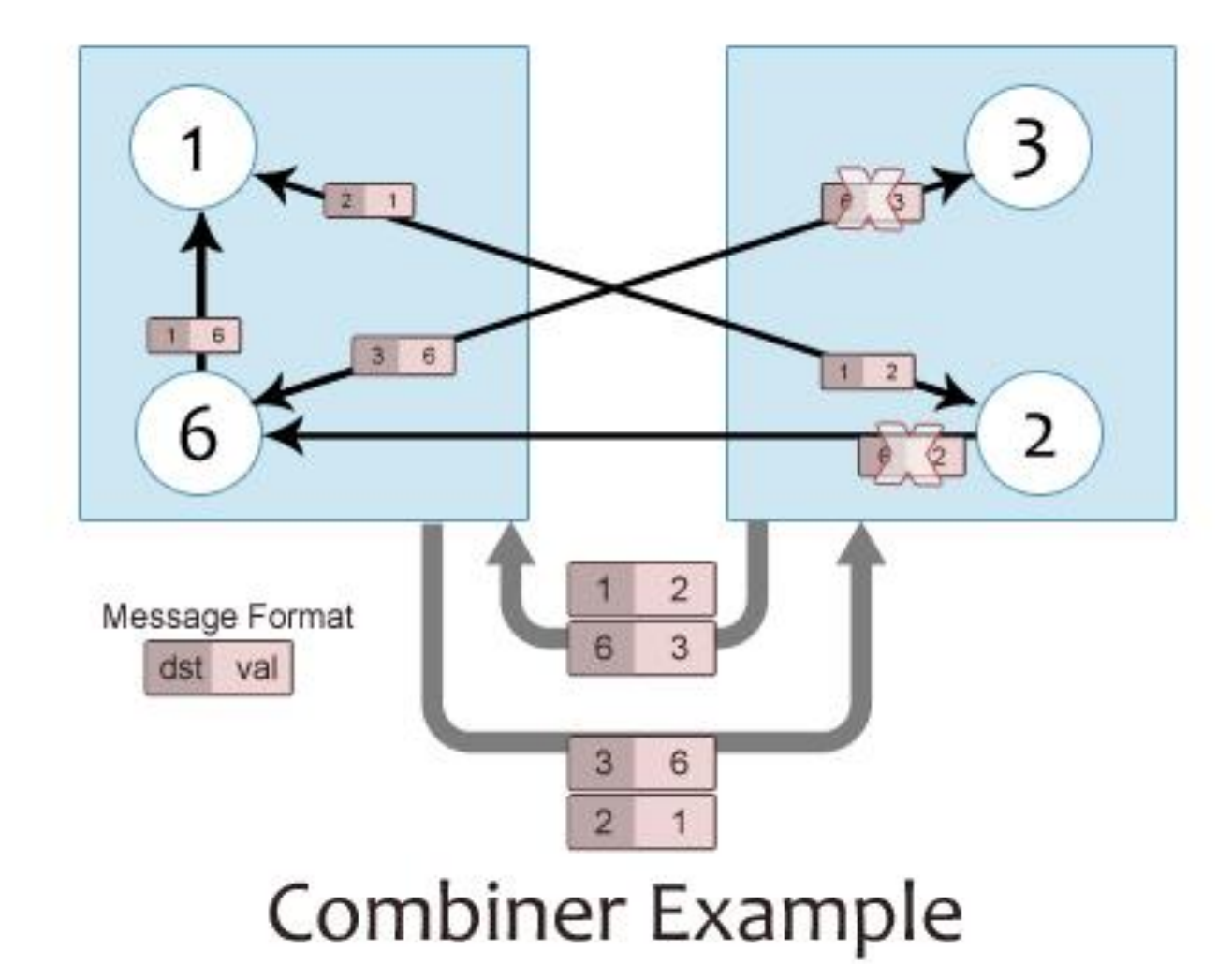
Say we want to count the number of incoming links to all the pages in a set of interconnected pages.

In the first iteration, for each link from a vertex(page) we will send a message to the destination page.

Here, **count** function over the incoming messages can be used a combiner to optimize performance.

In the MaxValue Example, a **Max** combiner would reduce the communication load.

The C++ API – Combiners



The C++ API – Aggregators

They are used for Global communication, monitoring and data.

- Each vertex can produce a value in a superstep S for the Aggregator to use. The Aggregated value is available to all the vertices in superstep S+1.

Aggregators can be used for statistics and for global communication.

Can be implemented by subclassing the **Aggregator** Class

- Commutativity and Assosiativity required

The C++ API – Aggregators

Example:

Sum operator applied to out-edge count of each vertex can be used to generate the total number of edges in the graph and communicate it to all the vertices.

- More complex reduction operators can even generate histograms.

In the MaxValue example, we can finish the entire program in a single superstep by using a **Max** aggregator.

The C++ API – Topology Mutations

The **Compute()** function can also be used to modify the structure of the graph.

Example: Hierarchical Clustering

Mutations take effect in the superstep after the requests were issued.

Ordering of mutations, with

- deletions taking place before additions,
- deletion of edges before vertices and
- addition of vertices before edges

resolves most of the conflicts. Rest are handled by user-defined handlers.

Implementation

Pregel is designed for the Google cluster architecture.

The architecture schedules jobs to optimize resource allocation, involving killing instances or moving them to different locations.

Persistent data is stored as files on a distributed storage system like GFS[8] or BigTable.

Basic Architecture

The Pregel library divides a graph into partitions, based on the vertex ID, each consisting of a set of vertices and all of those vertices' out-going edges.

The default function is *hash(ID)* mod *N*, where *N* is the number of partitions.

The next few slides describe the several stages of the execution of a Pregel program.

Pregel Execution

1. Many copies of the user program begin executing on a cluster of machines. One of these copies acts as the master.

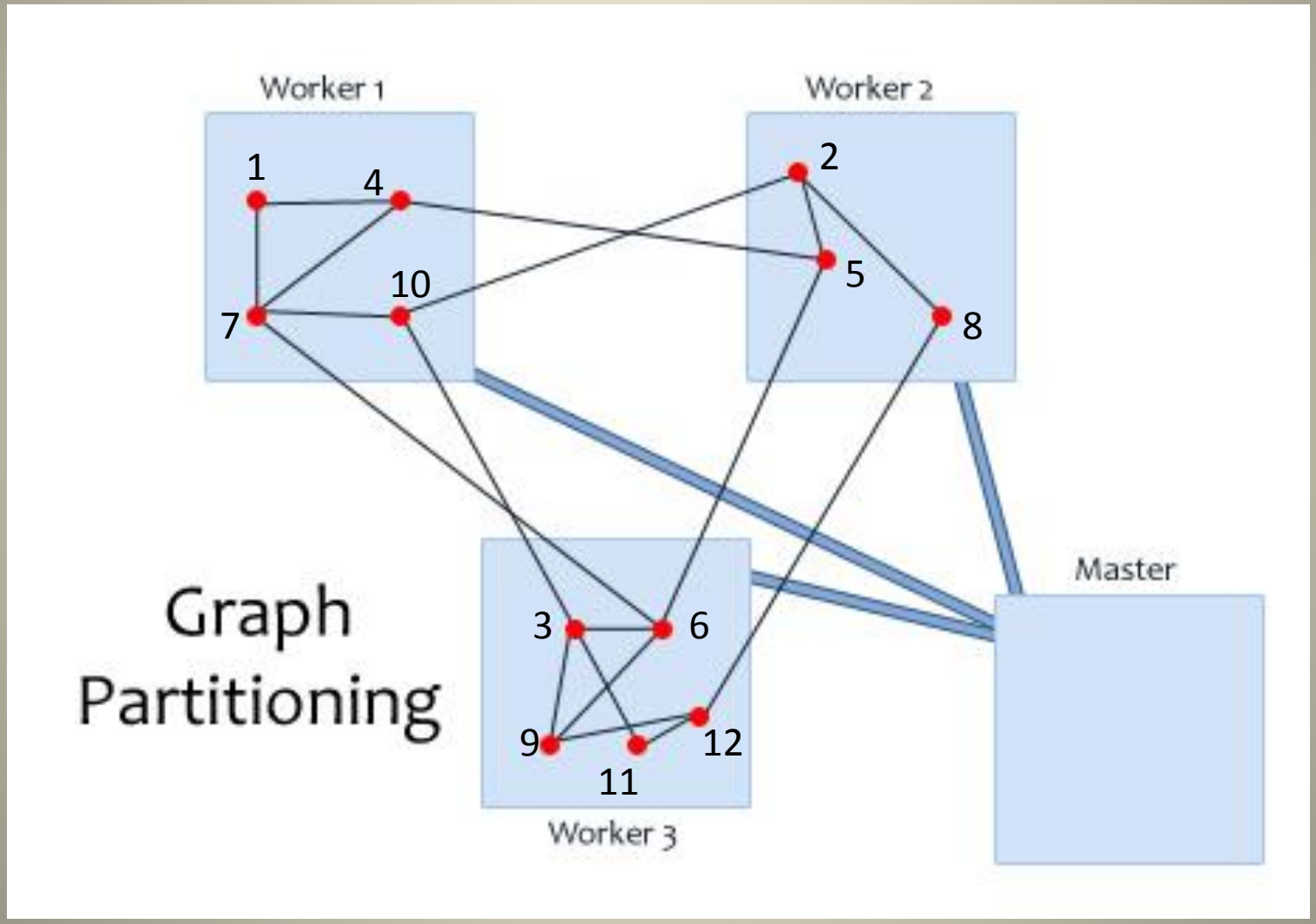
The master is not assigned any portion of the graph, but is responsible for coordinating worker activity.

Pregel Execution

2. The master determines how many partitions the graph will have and assigns one or more partitions to each worker machine.

Each worker is responsible for maintaining the state of its section of the graph, executing the user's **Compute()** method on its vertices, and managing messages to and from other workers.

Pregel Execution



Pregel Execution

3. The master assigns a portion of the user's input to each worker.

The input is treated as a set of records, each of which contains an arbitrary number of vertices and edges.

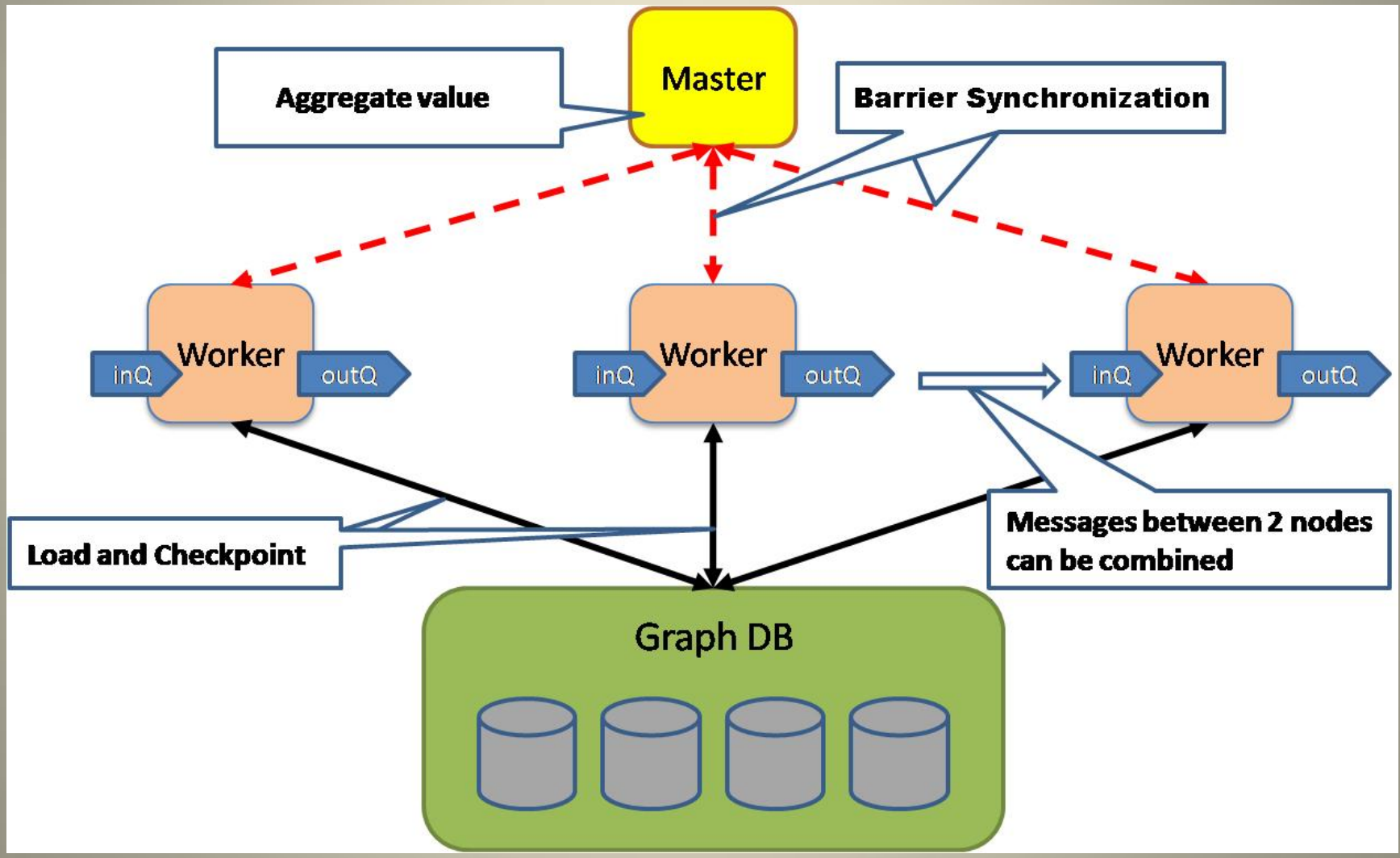
After the input has finished loading, all vertices are marked are active.

Pregel Execution

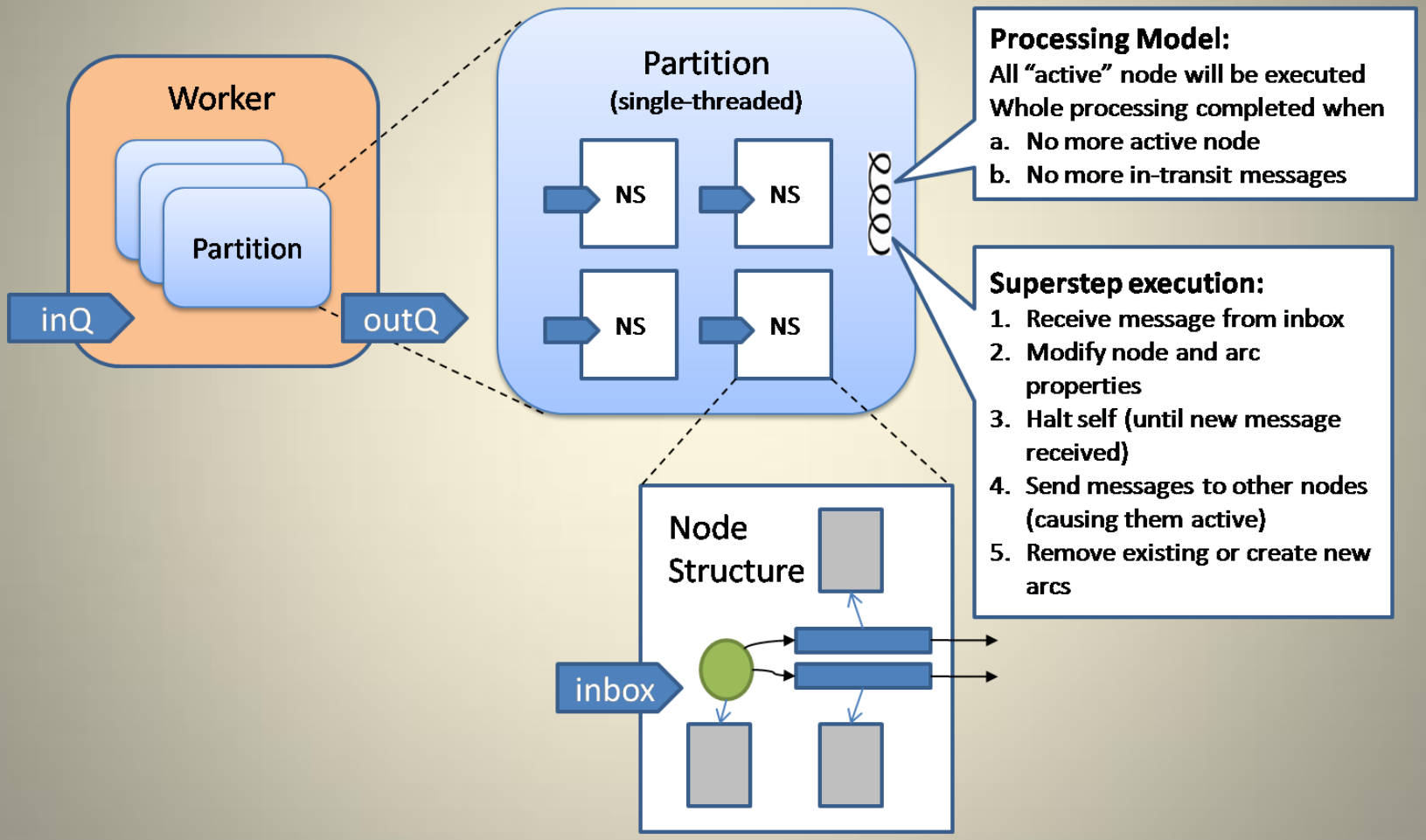
4. The master instructs each worker to perform a superstep. The worker loops through its active vertices, and call **Compute()** for each active vertex. It also delivers messages that were sent in the previous superstep.

When the worker finishes it responds to the master with the number of vertices that will be active in the next superstep.

Pregel Execution



Pregel Execution



Fault Tolerance

- Checkpointing is used to implement fault tolerance.
  - At the start of every superstep the master may instruct the workers to save the state of their partitions in stable storage.
  - This includes vertex values, edge values and incoming messages.
- Master uses “ping“ messages to detect worker failures.

Fault Tolerance

- When one or more workers fail, their associated partitions' current state is lost.
- Master reassigns these partitions to available set of workers.
  - They reload their partition state from the most recent available checkpoint. This can be many steps old.
  - The entire system is restarted from this superstep.
- *Confined recovery* can be used to reduce this load

Applications

PageRank

PageRank

PageRank is a link analysis algorithm that is used to determine the importance of a document based on the number of references to it and the importance of the source documents themselves.

[This was named after Larry Page (and not after rank of a webpage)]

PageRank

A = A given page

T_1 …. T_n = Pages that point to page A (citations)

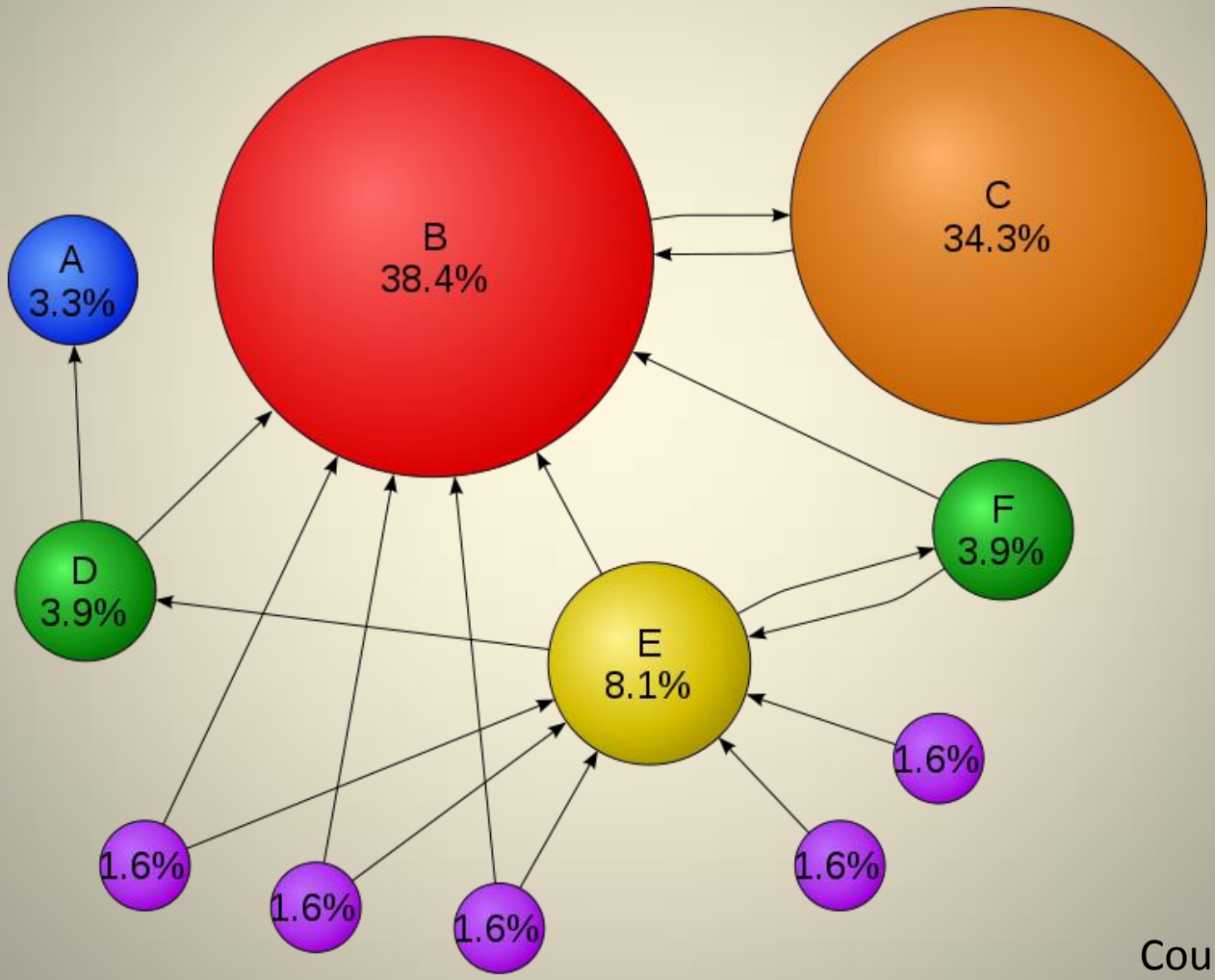
d = Damping factor between 0 and 1 (usually kept as 0.85)

C(T) = number of links going out of T

PR(A) = the PageRank of page A

$$PR(A) = (1-d) + d \bullet (\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_2)}{C(T_2)} + ... + \frac{PR(T_n)}{C(T_n)})$$

PageRank



Courtesy: Wikipedia

PageRank

PageRank can be solved in 2 ways:

- A system of linear equations
- An iterative loop till convergence

We look at the pseudo code of iterative version

```
Initial value of PageRank of all pages = 1.0;
While ( sum of PageRank of all pages – numPages > epsilon) {
        for each Page Pi in list {
                PageRank(Pi) = (1-d);
                for each page Pj linking to page Pi {
                        PageRank(Pi) += d ×
                                (PageRank(Pj)/numOutLinks(Pj));

                }
        }
}
```

PageRank in MapReduce – Phase I

Parsing HTML

- Map task takes (URL, page content) pairs and maps them to (URL, (PR_{init}, list-of-urls))
  - PR_{init} is the "seed" PageRank for URL
  - list-of-urls contains all pages pointed to by URL

- Reduce task is just the identity function

PageRank in MapReduce – Phase 2

PageRank Distribution

- Map task takes (URL, (cur_rank, url_list))
  - For each *u* in url_list, emit (*u*, cur_rank/|url_list|)
  - Emit (URL, url_list) to carry the points-to list along through iterations

- Reduce task gets (URL, url_list) and many (URL, *val*) values
  - Sum *val*s and fix up with *d*
  - Emit (URL, (new_rank, url_list))

PageRank in MapReduce - Finalize

- A non-parallelizable component determines whether convergence has been achieved
- If so, write out the PageRank lists - done
- Otherwise, feed output of Phase 2 into another Phase 2 iteration

PageRank in Pregel

```
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 + 0.85 * sum;
                }
                if (supersteps() < 30) {
                        const int64 n = GetOutEdgeIterator().size();
                        SendMessageToAllNeighbors(GetValue() / n);
                } else {
                        VoteToHalt();
                }}};
```

PageRank in Pregel

The pregel implementation contains the PageRankVertex, which inherits from the Vertex class.

The class has the vertex value type **double** to store tentative PageRank and message type **double** to carry PageRank fractions.

The graph is initialized so that in superstep 0, value of each vertex is **1.0** .

PageRank in Pregel

In each superstep, each vertex sends out along each outgoing edge its tentative PageRank divided by the number of outgoing edges.

Also, each vertex sums up the values arriving on messages into *sum* and sets its own tentative PageRank to $0.15 + 0.85 \times sum$

For convergence, either there is a limit on the number of supersteps or *aggregators* are used to detect convergence.

Applications

Shortest Paths

Shortest Path

There are several important variants of shortest paths like *single-source* shortest path, *s-t* shortest path and *all-pairs* shortest path.

We shall focus on *single-source* shortest path problem, which requires finding a shortest path between a single source vertex and every other vertex in the graph.

Shortest Path

```
Class ShortestPathVertex
        : public Vertex<int, int, int> {
  public:
        virtual 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();
        }
};
```

Shortest Path

In this algorithm, we assume the value associated with every vertex to be INF (a constant larger than any feasible distance).

In each superstep, each vertex first receives, as messages from its neighbors, updated potential minimum distances from the source vertex.

If the minimum of these updates is less than the value currently associated with the vertex, then this vertex updates its value and sends out potential updates to its neighbors, consisting of the weight of each outgoing edge added to the newly found minimum distance.

Shortest Path

In the 1st superstep, only the source vertex will update its value (from INF to zero) and send updates to its immediate neighbors.

These neighbors in turn will update their values and send messages, resulting in a wave front of updates through the graph.

The algorithm terminates when no more updates occur, after which the value associated with each vertex denotes the minimum distance from the source vertex to that vertex. The algorithm is guaranteed to terminate if there are no negative edges.

Shortest Path

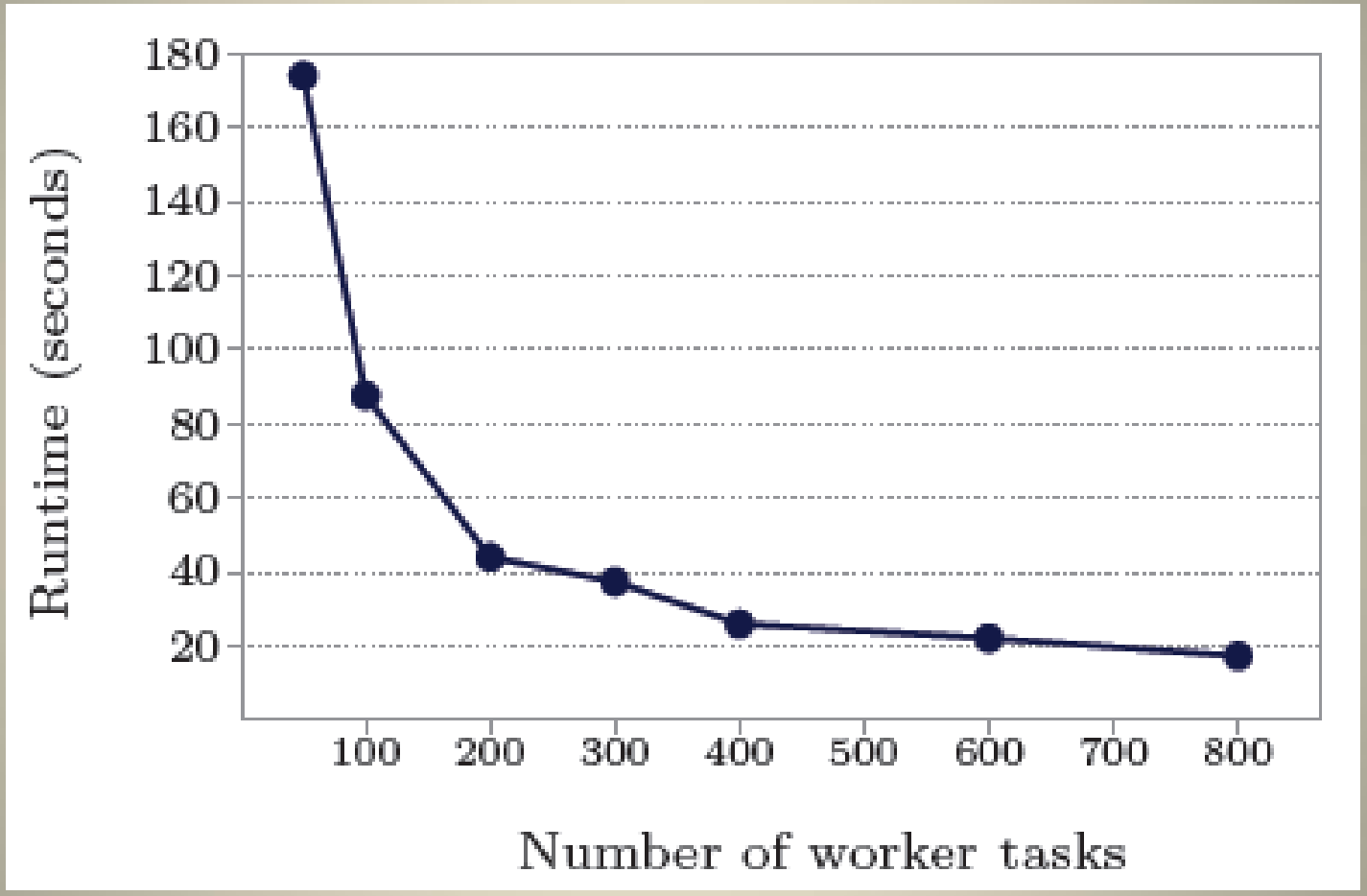
Experiments:

Various experiments were conducted with the single-source shortest paths implementation on a cluster of 300 multicore commodity PCs. Runtimes are reported for

- binary trees (to study scaling properties) and
- lognormal random graphs (to study the performance in a more realistic setting)

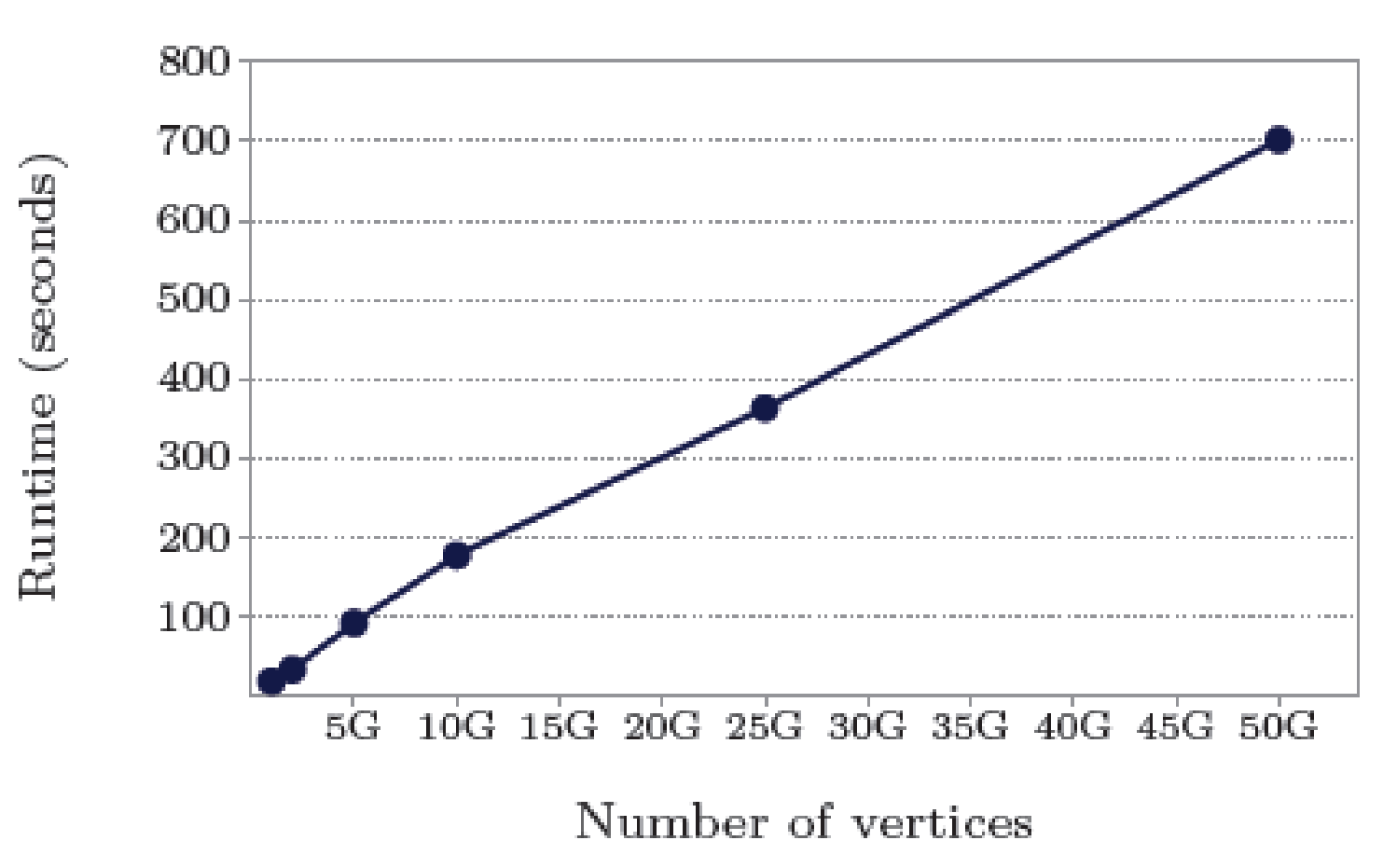
using various graph sizes with the weights of all edges implicitly set to 1.

Shortest Path



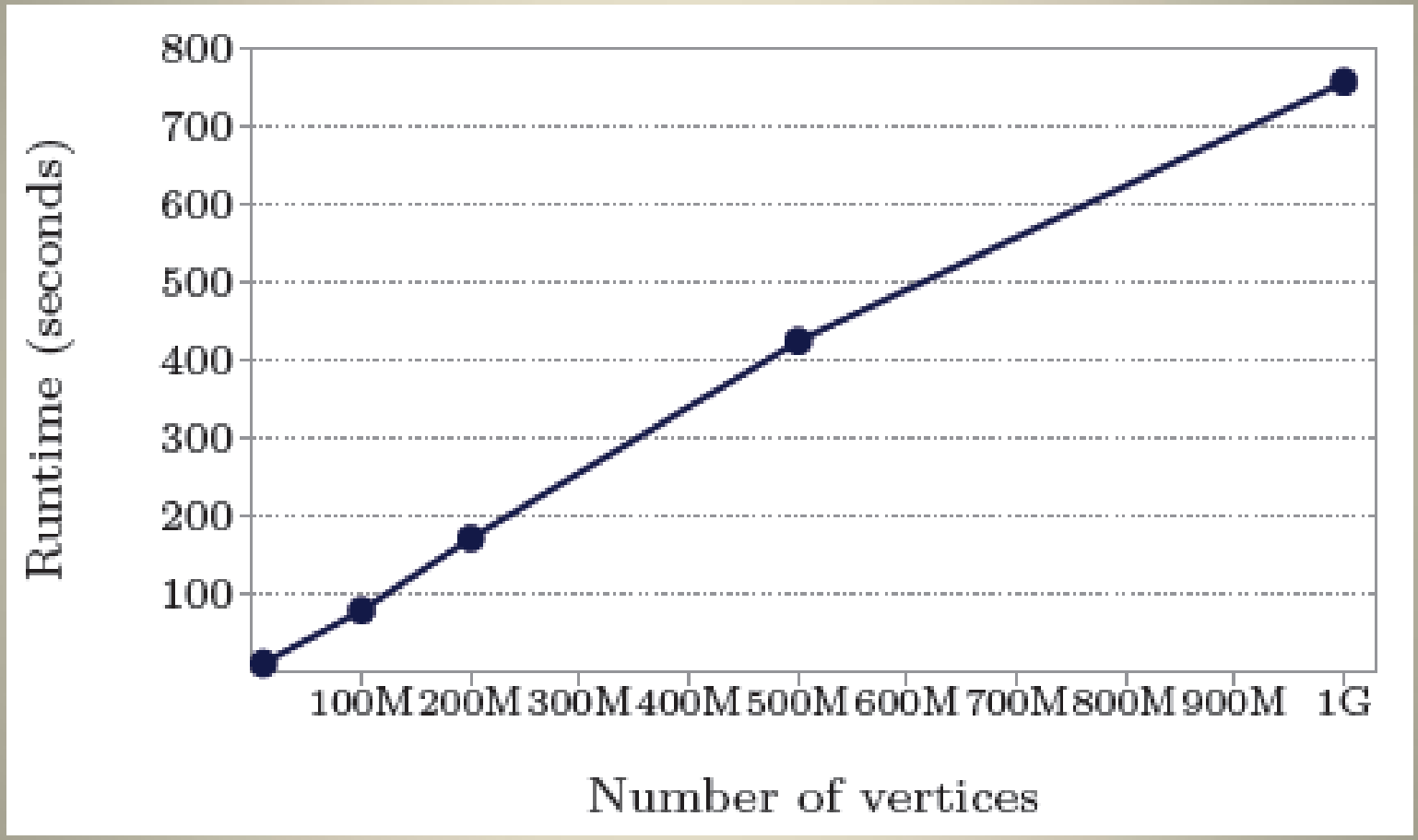
1 billion vertex binary tree: varying number of worker tasks

Shortest Path



binary trees: varying graph sizes on 800 worker tasks

Shortest Path



log-normal random graphs, mean out-degree 127.1 (thus over 127 billion edges in the largest case): varying graph sizes on 800 worker tasks

Applications

Bipartite Matching

Bipartite Matching

The input to a bipartite matching algorithm consists of 2 distinct sets of vertices with edges only between the sets, and the output is a subset of edges with no common endpoints.

In the Pregel implementation, the algorithm is a randomized matching algorithm.

The vertex value is a tuple of 2 values: a flag indicating which set the vertex is in (L or R), and the name of its matched vertex once it is known.

Bipartite Matching

```
Class BipartiteMatchingVertex
    : public Vertex<tuple<position, int>, void, boolean> {
  public:
    virtual void Compute(MessageIterator* msgs) {
      switch (superstep() % 4) {
        case 0: if (GetValue().first == 'L') {
            SendMessageToAllNeighbors(1);
            VoteToHalt();
          }
        case 1: if (GetValue().first == 'R') {
          Rand myRand = new Rand(Time());
          for ( ; !msgs->Done(); msgs->Next()) {
            if (myRand.nextBoolean()) {
              SendMessageTo(msgs->Source, 1);
              break;
            }
          }
          VoteToHalt(); }
```

Bipartite Matching

```
    case 2:
      if (GetValue().first == 'L') {
        Rand myRand = new Rand(Time());
        for ( ; !msgs->Done(); msgs->Next) {
          if (myRand.nextBoolean()) {
            *MutableValue().second = msgs->Source());
            SendMessageTo(msgs->Source(), 1);
            break;
          }
        }
        VoteToHalt(); }
    case 3:
      if (GetValue().first == 'R') {
        msgs->Next();
        *MutableValue().second = msgs->Source();
      }
      VoteToHalt();
  }}};
```

Bipartite Matching

The algorithm proceeds in cycles of 4 phases.

In phase 1, each left vertex not yet matched sends a message to each of its neighbors to request a match, and then unconditionally votes to halt.

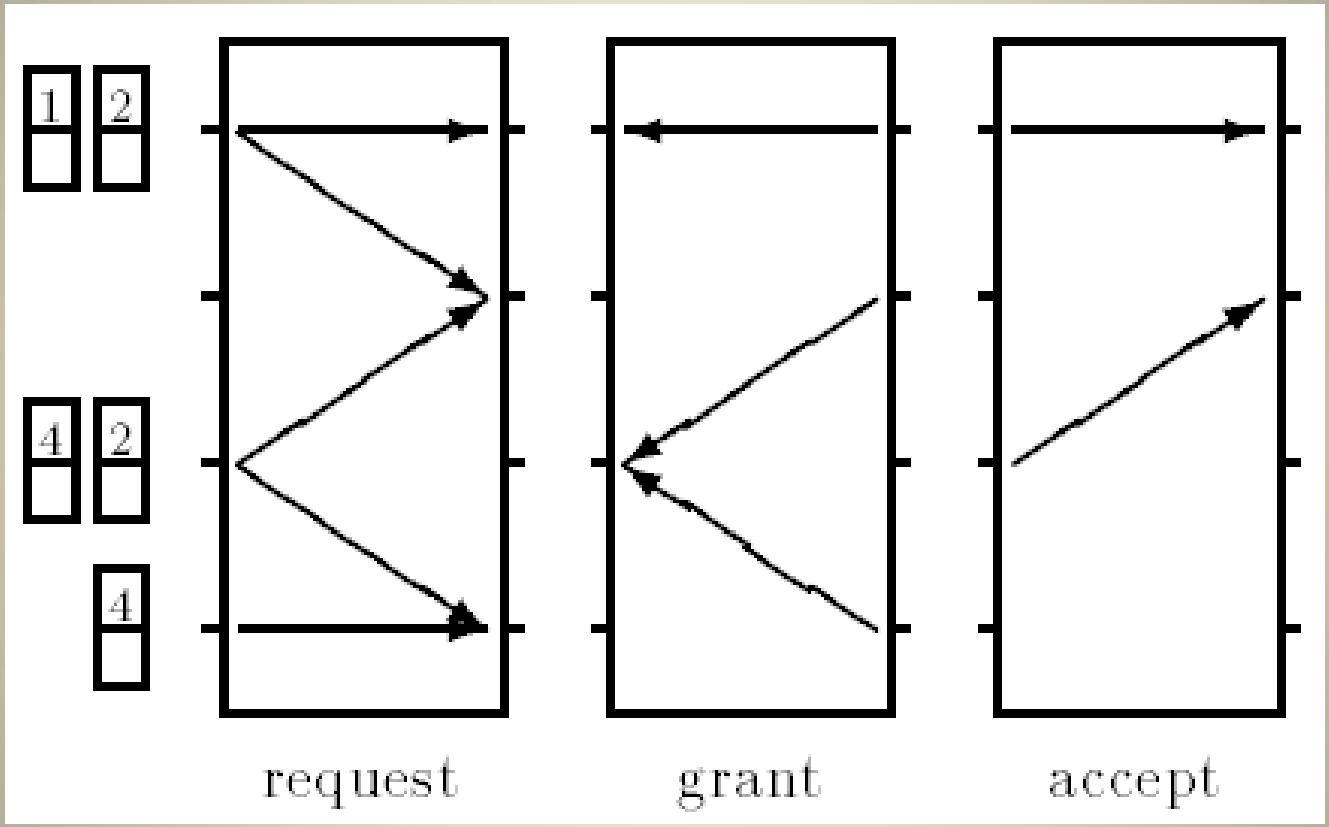
In phase 2, each right vertex not yet matched randomly chooses one of the messages it receives, sends a message granting that request and sends messages to other requestors denying it. Then it unconditionally votes to halt.

Bipartite Matching

In phase 3, each left vertex not yet matched chooses one of the grants it receives and sends an acceptance message. Left vertices that are already will never execute this phase since they will not have sent any messages in phase 0.

In phase 4, an unmatched right vertex receives at most one acceptance message. It notes the matches node and unconditionally votes to halt.

Bipartite Matching



Execution of a cycle (A cycle consists of 4 supersteps)

THANK YOU

ANY QUESTIONS?

References

[1] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry, *Challenges in Parallel Graph Processing*. Parallel Processing Letters 17, 2007, 5-20.

[2] Kameshwar Munagala and Abhiram Ranade, *I/O-complexity of graph algorithms.* in Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms, 1999, 687-694.

[3] Joseph R. Crobak, Jonathan W. Berry, Kamesh Madduri, and David A. Bader, *Advanced Shortest Paths Algorithms on a Massively-Multithreaded Architecture.* in Proc. First Workshop on Multithreaded Architectures and Applications, 2007, 1-8.

[4] U Kung, Charalampos E. Tsourakakis, and Christos Faloutsos*, Pegasus: A Peta-Scale Graph Mining System - Implementation and Observations.* Proc. Intl. Conf. Data Mining, 2009, 229-238.

References

[5] Douglas Gregor and Andrew Lumsdaine, *The Parallel BGL: A Generic Library for Distributed Graph Computations.* Proc. of Parallel Object-Oriented Scientic Computing (POOSC), July 2005.

[6] Albert Chan and Frank Dehne, *CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines.* Intl. J. of High Performance Computing Applications 19(1), 2005, 81-97.

[7] Leslie G. Valiant, *A Bridging Model for Parallel Computation.* Comm. ACM 33(8), 1990, 103-111.

[8] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung, *The Google File System*. In Proc. 19th ACM Symp. On Operating Syst. Principles, 2003, 29-43.

Extra Slides

Applications

Semi-clustering

Semi-clustering

Vertices in a social graph typically represent people, and edges represent connections between them.

A semi-cluster in a social graph is a group of people who interact frequently with each other and less frequently with others. It is different from ordinary clustering in the sense that a vertex may belong to more than one semi-cluster.

Semi-clustering

The algorithm used is a greedy algorithm.

Its input is a weighted, undirected graph and its output is at most C_{max} semi-clusters, each containing at most V_{max} vertices, which are both user-defined parameters.

Semi-clustering

A semi-cluster is assigned a score,

$$S_c = \frac{I_c - f_B B_c}{V_c(V_c - 1)/2}$$

Where,

I_c is the sum of the weights of all internal edges,
B_c is the sum of all boundary edges and
f_B is the boundary score factor, which is a user-defined parameter, between 0 and 1.

The score is normalized by dividing with the number of vertices in the clique, so that large clusters do not receive artificially high scores.

Semi-clustering

Each vertex V maintains a list containing at most C_{max} semi-clusters, sorted by score.

In superstep 0, V enters itself in that list as a semi-cluster of size 1 and score 1, and publishes itself to all of its neighbors.

In subsequent supersteps, V circulates over the semi-clusters sent to it in the previous superstep. If a semi-cluster *c* does not already contain V, V is added to *c* to form *c'*.

Semi-clustering

The semi-clusters are sorted by their scores and the best ones are sent to V's neighbors.

Vertex V updates its list of semi-clusters with the semi-clusters that contain V.

The algorithm terminates either when the semi-clusters stop changing or the user may provide a limit. At that point, the list of best semi-cluster candidates for each vertex may be aggregated into a global list of best semi-clusters.

Model Of Computation: Output

- Output of the vertices is a set of values explicitly output by the vertices
- It can form a directed graph isomorphic to the input or different from it.
  - This can be because edges and vertices can be added/deleted while computation

PageRank in MapReduce

Computing pagerank

$$R'(u) = d \bullet \sum \frac{R'(v)}{N(v)} + (1-d)$$

R’(u) – New page rank of page u

R’(v) – New page rank of page v, where v links to u

N(v) – Number of outlinks from page v

The input that ‘reduce’ gets for each document v linking to u: [Key: “u”, Value: “v <PageRank of v> <number of outlinks from v>”.]

So we just sum over all the values passed to the reducer to compute the new PageRank.

PageRank in MapReduce

Parsing MapReduce

Map Phase

Input: For a document **index.html**

```
<html><body>Blah blah blah... <a href="2.html"> A
link</a>....</html>
```

Output: For each link

```
[Key: "index.html", value: "2.html"]
```

Reduce Phase

Default PageRank

Input:
```
[Key: "index.html",
value: ["2.html", "3.html", "4.html"…]]
```

Output:
```
[key: "index.html", Value: "1.0 2.html 3.html
..." ]
```

PageRank in MapReduce

Computation Iterations

Map Phase

Input: [Key: “index.html”,
Value: “1.0, 1.html, 2.html, 3.html ...”]

Output: For each outlink

[Key: “1.html”, value: “index.html <pagerank> <number of outlinks>” ...]

Reduce Phase

New PageRank

Input: [Key: “1.html”, value: “index.html <pagerank> <number of outlinks>” ...]

Output: [key: “1.html”, Value: “<new pagerank> index.html 3.html” ...]

Iterate till convergence!