CPT-S 415

Big Data

Yinghui Wu

EME B45

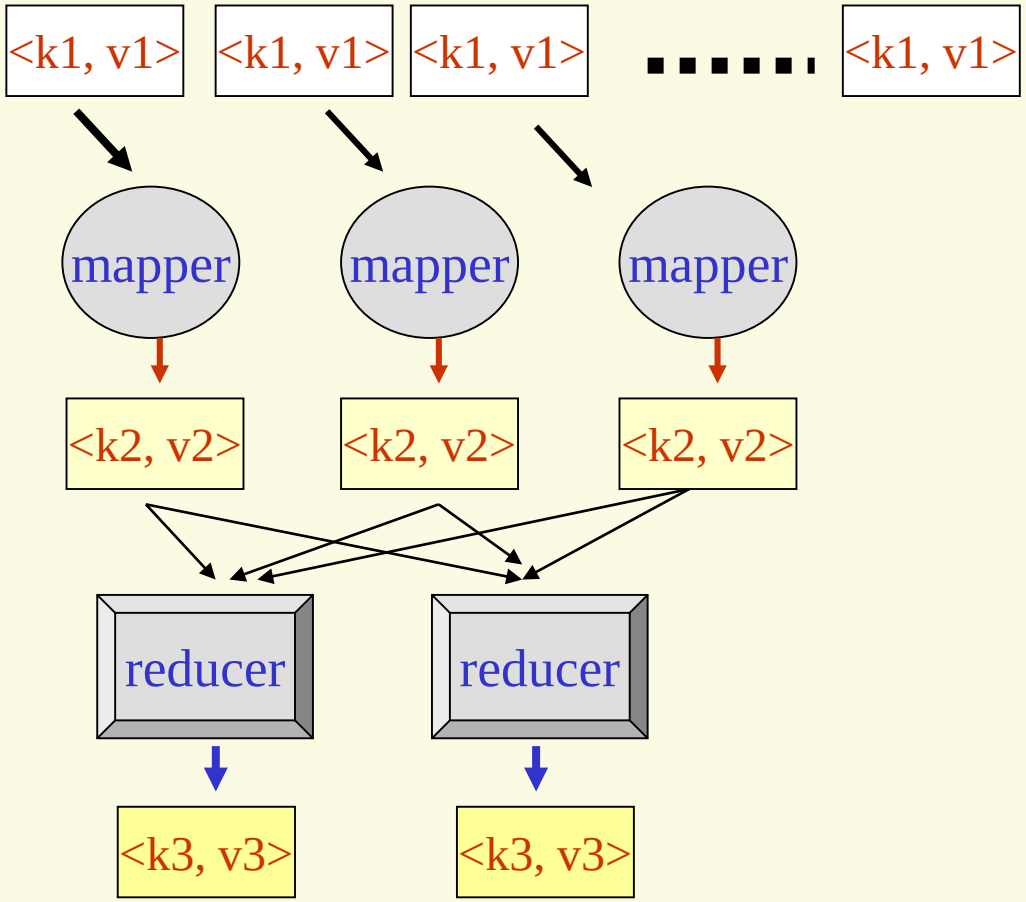
CPT-S 415
Big Data

Parallel models

beyond MapReduce

- ✓ Vertex-centric models
  - Pregel (BSP)
  - GraphLab
- ✓ GRAPE

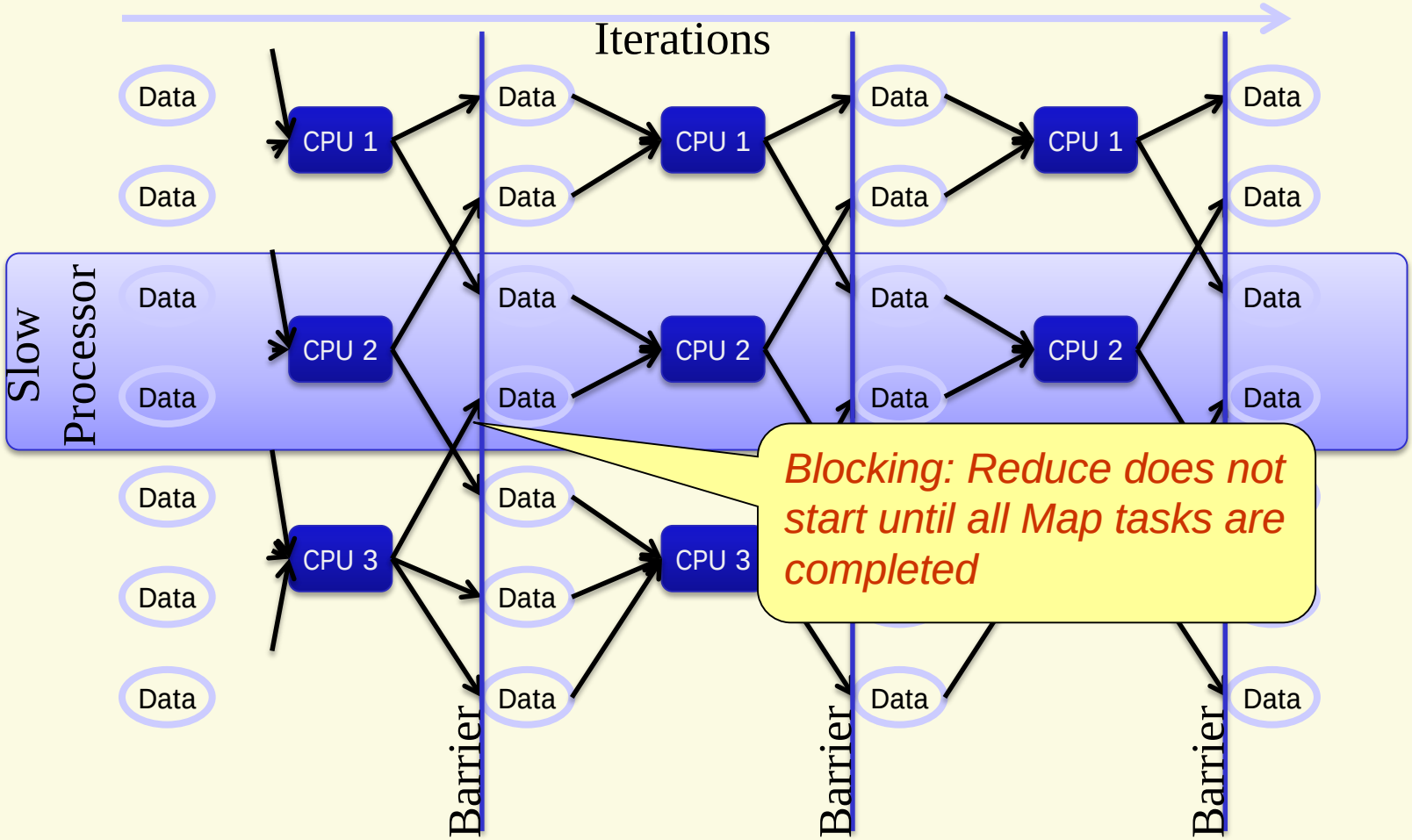
Inefficiency of MapReduce



reasons?

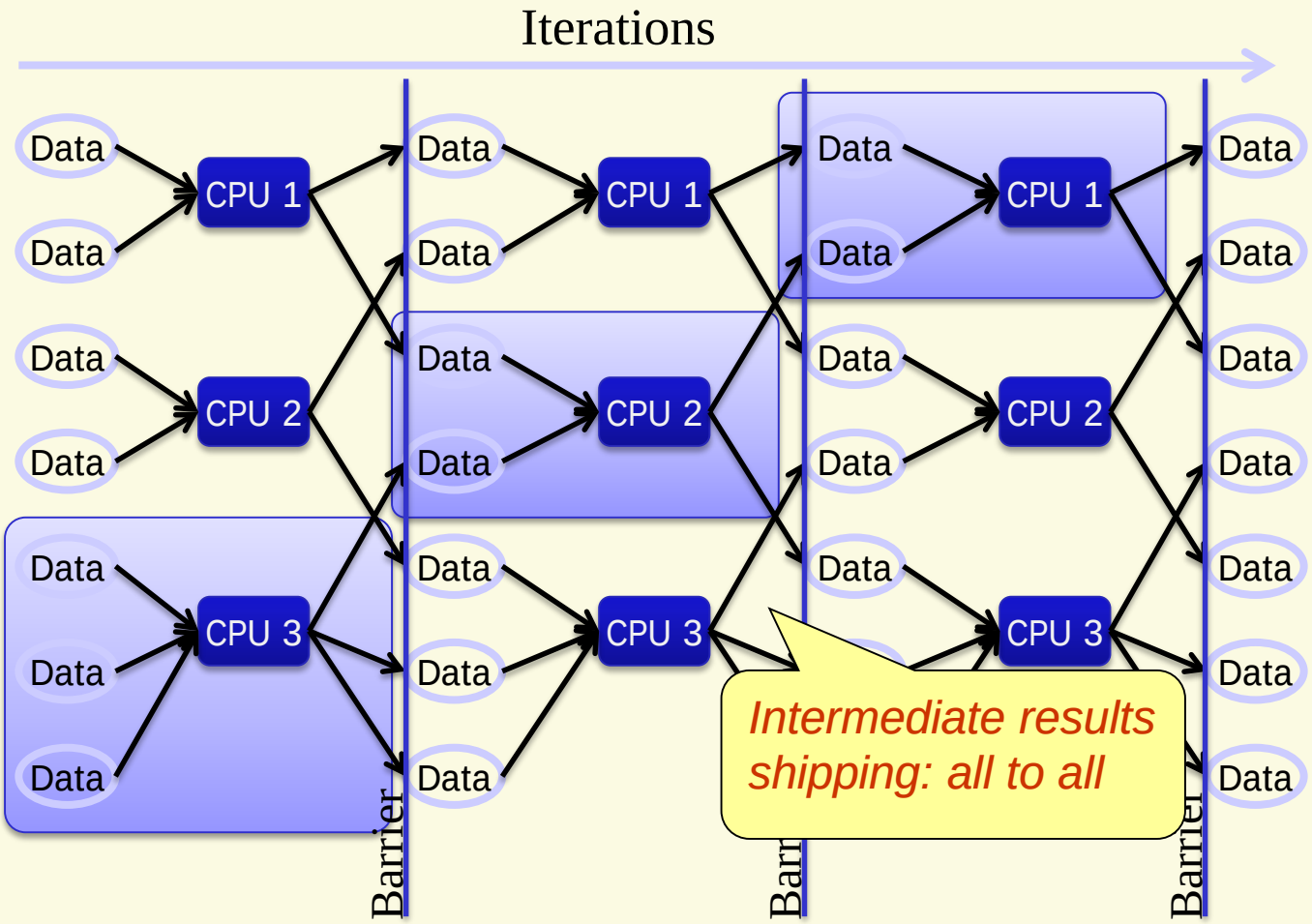
Iterative Algorithms

- ✓ Map-Reduce not efficiently express iterative algorithms:



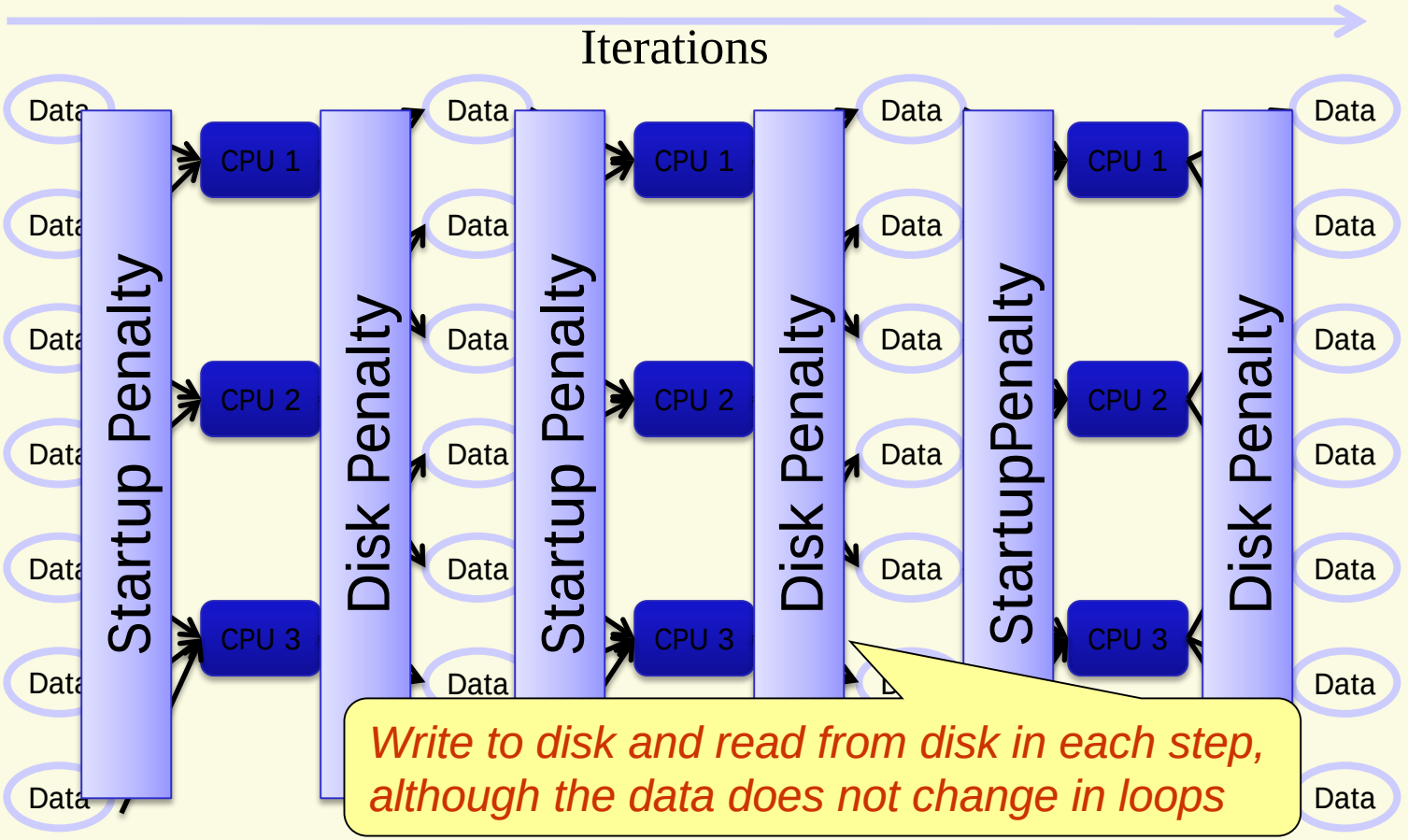
Iterative MapReduce

- Only a subset of data needs computation:



Iterative MapReduce

- ✓ System is not optimized for iteration:



The need for parallel models beyond MapReduce

- ✓ MapReduce:
  - Inefficiency: blocking, intermediate result shipping (all to all); write to disk and read from disk in each step, even for invariant data in a loop
  - Does not support iterative graph computations:
    - External driver
    - No mechanism to support global data structures that can be accessed and updated by all mappers and reducers
  - Support for incremental computation?
  - Have to re-cast algorithms in MapReduce, hard to reuse existing (incremental) algorithms
  - General model, not limited to graphs

Can we do better for graph algorithms?

Map-Reduce for Data-Parallelism

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Graph parallel models

Lasso

Label Propagation

Kernel Methods

Belief Propagation

Tensor Factorization

PageRank

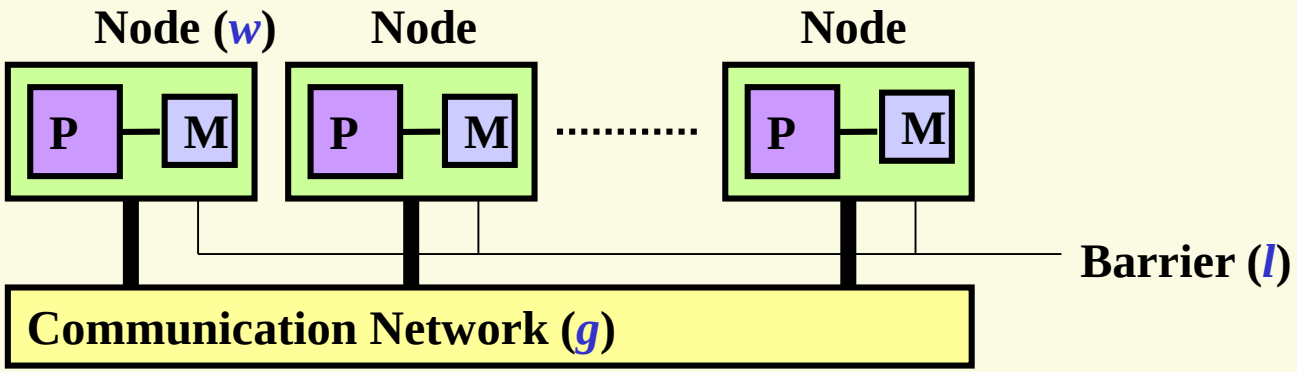
Deep Belief Networks

Neural Networks

BSP model

Bulk Synchronous Parallelism

- ✓ BSP is based on the Synchronizer Automata (Leslie Valiant, Harvard University)
- ✓ The model consists of:
  - A set of processor-memory pairs.
  - A communications network that delivers messages in a point-to-point manner.
  - A mechanism for the efficient barrier synchronization for all or a subset of the processes.



Bulk Synchronous Parallel Model (BSP)

- ✓ Leslie G. Valiant: A Bridging Model for Parallel Computation. Commun. ACM 33 (8): 1[illegible]

analogous to MapReduce rounds

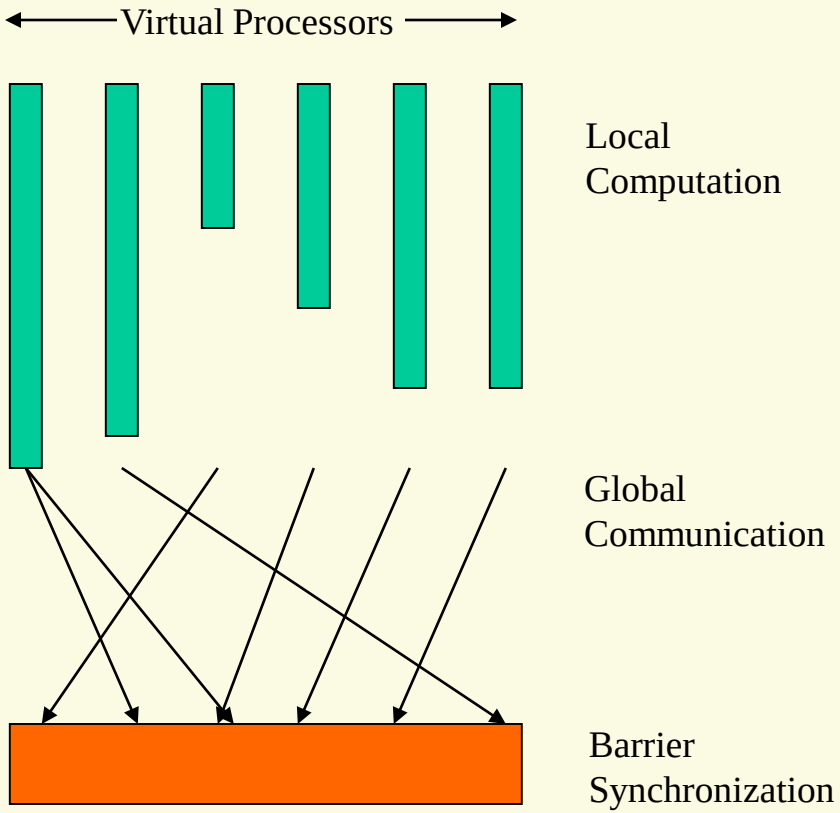
- ✓ Processing: a series of supersteps
- ✓ Vertex: computation is defined to run on each vertex
- ✓ Superstep S: *all vertices compute in parallel; each vertex v may*
  - receive messages sent to v from superstep S – 1;
  - perform some computation: modify its states and the states of its outgoing edges
  - Send messages to other vertices (to be received in the next superstep)

Message passing

Vertex-centric, message passing

BSP Programming Style

- Vertical Structure
  - Sequential composition of "supersteps".
    - Local computation
    - Process Communication
    - Barrier Synchronization
- Horizontal Structure
  - Concurrency among a fixed number of virtual processors.
  - Processes do not have a particular order.
  - Locality plays no role in the placement of processes on processors.
  - p = number of processors.



Barrier Synchronization

- "Often expensive and should be used as sparingly as possible."
- Developers of BSP claim that barriers are not as expensive as they are believed to be in high performance computing folklore.
- The cost of a barrier synchronization has two parts.
  - The cost caused by the variation in the completion time of the computation steps that participate.
  - The cost of reaching a globally-consistent state in all processors.

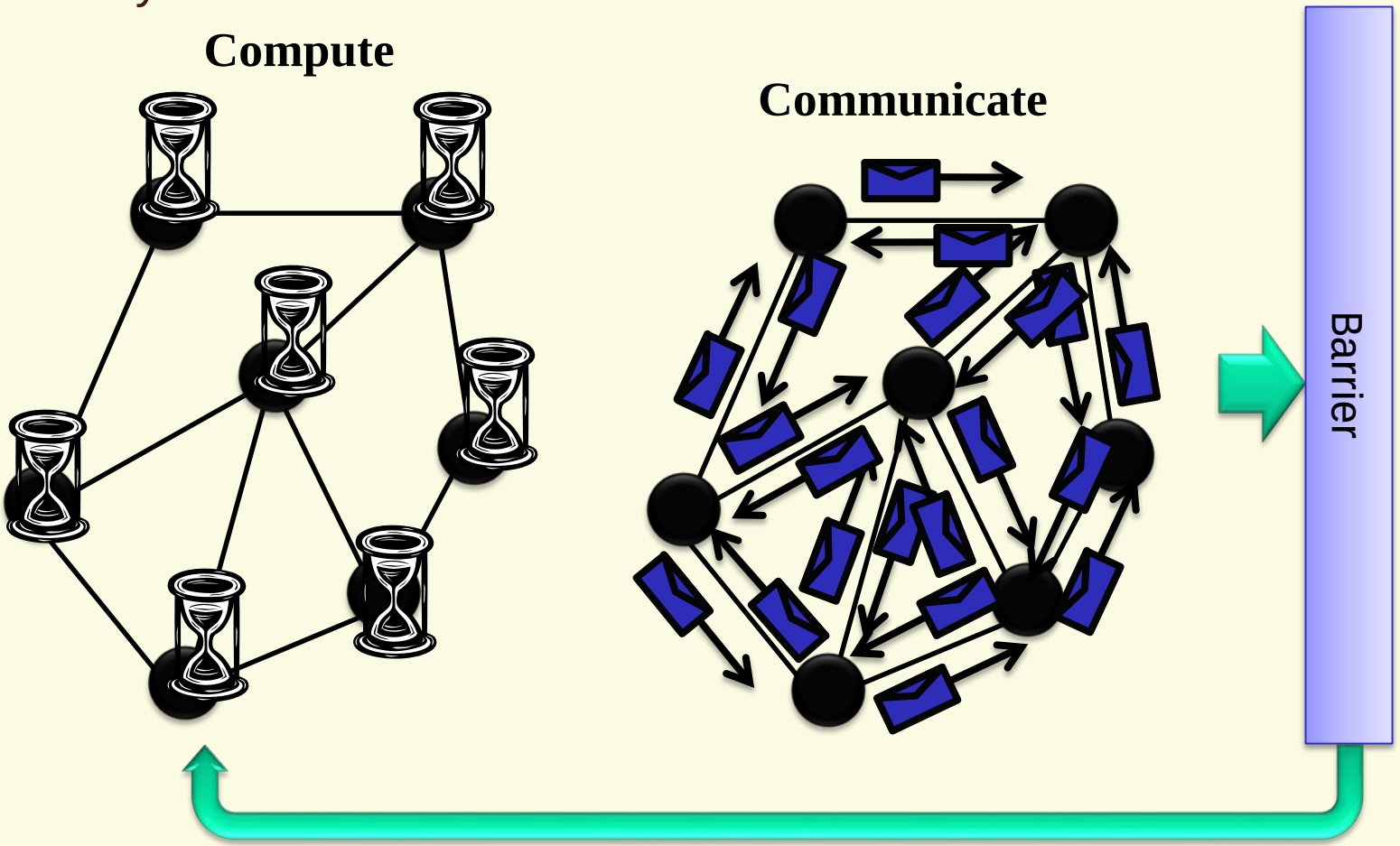
Vertex-centric models

Pregel: think like a vertex

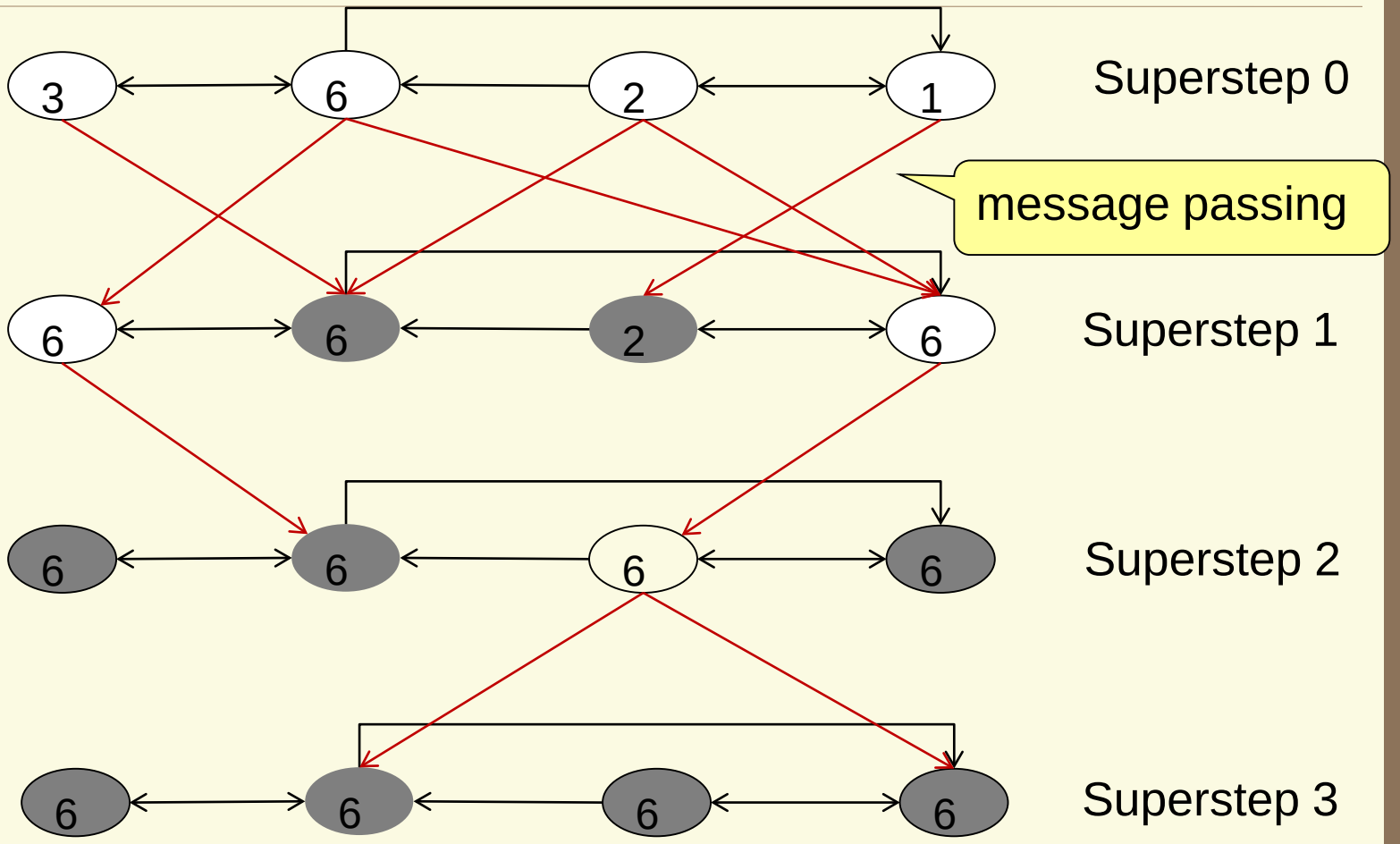
- Input: a directed graph G
  - Each vertex v: a node id, and a value
  - Edges: contain values (associated with vertices)
- Vertex: modify its state/edge state/edge sets (topology)
- Supersteps: within each, *all vertices compute in parallel*
- Termination:
  - Each vertex votes to halt
  - When all vertices are inactive and no messages in transit
- Synchronization: supersteps

Pregel (Giraph)

- Bulk Synchronous Parallel Model:



Example: maximum value



Shaded vertices: voted to halt

Vertex API

```
Template (VertexValue, EdgeValue, MessageValue)
Class Vertex {
   void Compute (MessageIterator: msgs)
    const  vertex_id;
    const   superstep();
    const  VertexValue&  GetValue();
    VertexValue*  MutableValue();
    OutEdgeIterator  GetOutEdgeIterator();
    void  SendMessageTo (dest_vertex, MessageValue& message);
    void  VoteToHalt();
   }
```

User defined

All messages received

Iteration control

Vertex value: mutable

Outgoing edges

Message passing: messages can be sent to any vertex whose id is known

Think like a vertex: local computation

PageRank

The likelihood that page v is visited by a random walk:

$$\alpha\,(1/|V|) \;+\; (1-\alpha)\,\Sigma\_(u \in L(v))\; P(u)/C(u)$$

random jump

following a link from other pages

- ✓ Recursive computation: for each page v in G,
  - compute P(v) by using P(u) for all $u \in L(v)$

until

  - converge: no changes to any P(v)
  - after a fixed number of iterations

A BSP algorithm?

PageRank in Pregel

```
PageRankVertex {
   Compute (MessageIterator: msgs) {
            if (superstep() >= 1)
               then sum := 0;
                     for all messages in msgs do
                      *MutableValue() := α  /NumVertices() + (1- α) sum;
    }
    if (superstep() < 30)
            then  n := GetOutEdgeIterator().size();
                  sendMessageToAllNeighbors(GetValue() / n);
            else   VoteToHalt();
   }
```

iterations

α (1/|V|) + (1 - α) Σ_(u ∈ L(v)) P(u)

Assume 30 iterations

α (1/|V|) + (1 - α) Σ_(u ∈ L(v)) P(u)/C(u)

Pass revised rank to its neighbors

VertexValue: the current rank

Dijkstra's algorithm for distance queries

- ✓ Distance: single-source shortest-path problem
  - Input: A directed weighted graph G, and a node s in G
  - Output: The lengths of shortest paths from s to all nodes in G

- ✓ Dijkstra (G, s, w):
  1. for all nodes v in V do
     a. d[v] ← ∞; ←
  2. d[s] ← 0; Que ← V;
  3. while Que is nonempty do
     a. u ← ExtractMin(Que);
     b. for all nodes v in adj(u) do
        a) if d[v] > d[u] + w(u, v) then d[v] ← d[u] + w(u, v);

Use a priority queue Que; w(u, v): weight of edge (u, v); d(u): the distance from s to u

Extract one with the minimum d(u)

An algorithm in Pregel?

Distance queries in Pregel

```
ShortesPathVertex {
  Compute (MessageIterator: msgs) {
        if  isSource(vertex_id( )
        then minDist := 0  else  minDist  := ∞;
        for all messages  m in msgs do
             minDist := min(minDist, m.Value());
        if midDist < GetValue()
        then    *MutableValue() := minDist;
        for all nodes v linked to from the current node u do
             SendMessageTo(v,  minDist + w(u, v);
        VoteToHalt();
  }
```

Refer to the current node as u

aggregation

Messages: distances to u

MutableValue: the current distance

Pass revised distance to its neighbors

Think like a vertex

Combiner and Aggregation

- ✓ Combine several messages intended for a vertex
  - – Provided that the messages can be aggregated (“reduced”) by using some associative and commutative function
  - – Reduce the number of messages
- ✓ Each vertex can provide a value to an aggregator in any superstep S.
- ✓ System aggregates these values (“reduce”)
- ✓ The aggregated values are made available to all vertices in superstep S + 1.

Global data structures

optimization

Topology mutation

- ✓ Function compute can add or remove vertices
- ✓ Possible conflicts:
  - Vertex 1 adds an edge to vertex 100
  - Vertex 2 deletes vertex 100
  - Vertex 1 creates a vertex 10 with value 10
  - Vertex 2 also creates a vertex 10 with value 12
- ✓ Handling conflicts:
  - Partial order on operations: edge removal < vertex removal < vertex addition < edge addition
- ✓ System: user specified

Extra power, yet increased complication

Pregel implementation

- ✓ Vertices are assigned to machines: hash(vertex.id) mod N
- ✓ Partitions can be user-specified, to co-locate all Web pages form the same site, for instance
- ✓ Cross edges: minimize edges across partitions
  - Sparsest Cut Problem
- ✓ Master: coordinate a set of workers (partitions, assignments)
- ✓ Worker: processes one or more partitions, local computation
  - Know the partition function and partitions assigned to it
  - All vertices in a partition are initially active
  - Worker notifies maste… of a superstep

Giraph, http://www.apache.org/dyn/closer.cgi/giraph

Master, worker

Fault tolerance

- ✓ Checkpoints: mater instructs workers to save state to HDFS
  - Vertex values
  - Edge values
  - Incoming messages
- ✓ Master saves aggregated values to disk
- ✓ Worker failure:
  - detected by regular “ping” messages issued by the master (mark it failed after specified interval)
  - Recovered by creating a new worker, with the state stored from previous checkpoint

recovery

The vertex centric model of GraphLab

No supersteps

- ✓ Vertex: computation is defined to run on each vertex
- ✓ *All vertices compute in parallel*
  - Each vertex reads and writes to data on adjacent nodes or edges

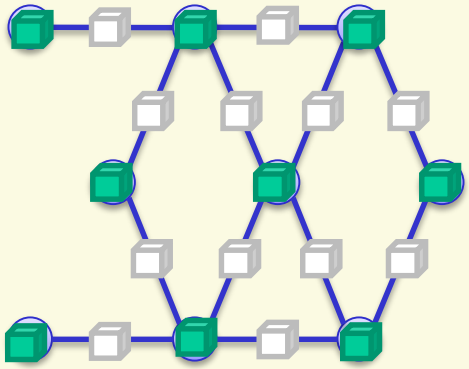
asynchronous

- ✓ Consistency: serialization
  - Full consistency: no overlap for concurrent updates
  - Edge consistency: exclusive read-write to its vertex and adjacent edges; read only to adjacent vertices
  - Vertex consistency: all updates in parallel (sync operations)
- ✓ Asynchronous: all vertices

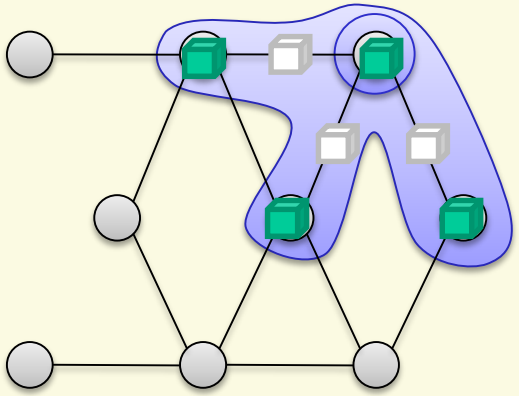
https://dato.com/products/create/open_source.html

The GraphLab Framework

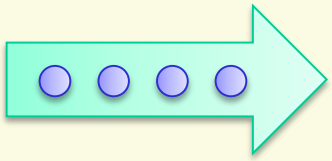
Graph Based
Data Representation



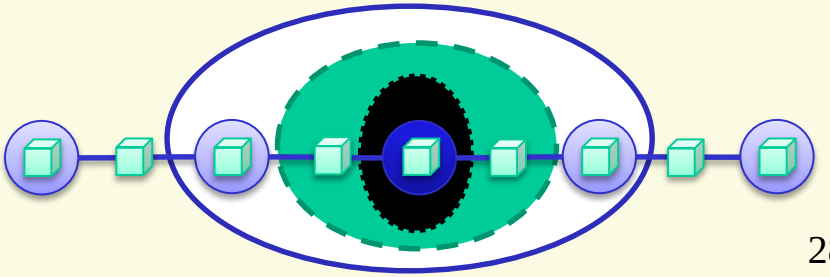
Update Functions
User Computation



Scheduler

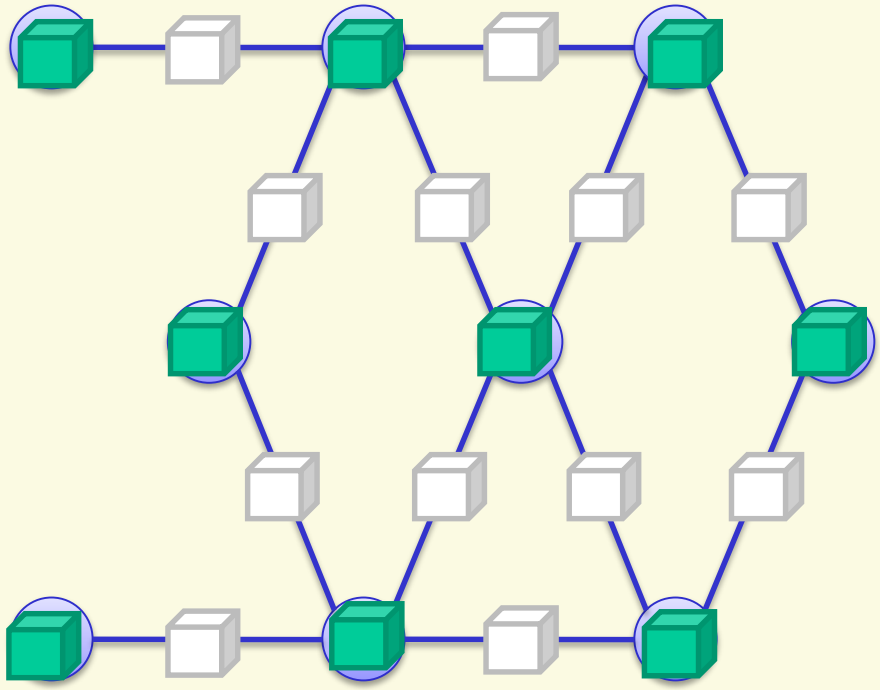


Consistency Model



Data Graph

A **graph** with arbitrary data (C++ Objects) associated with each vertex and edge.



Graph:

• Social Network

Vertex Data:

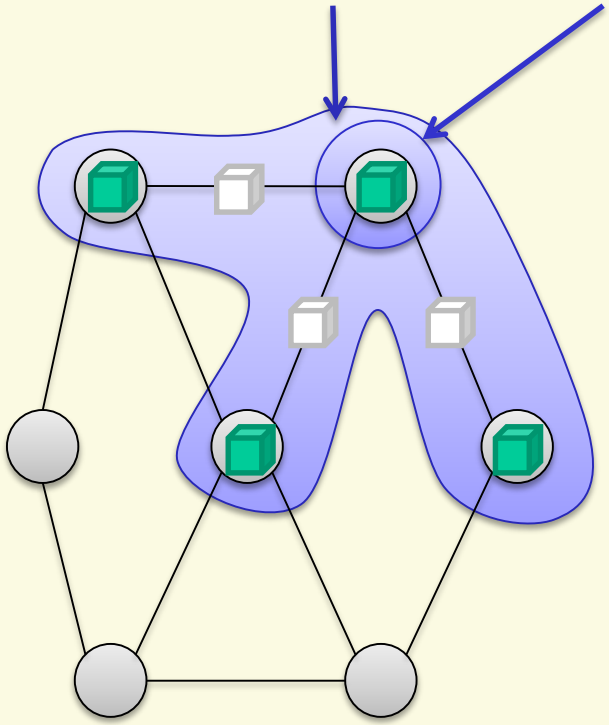
•User profile text
• Current interests estimates

Edge Data:

• Similarity weights

Update Functions

An **update function** is a user defined program which transforms the data in the **scope** of the **vertex** when applied to a vertex



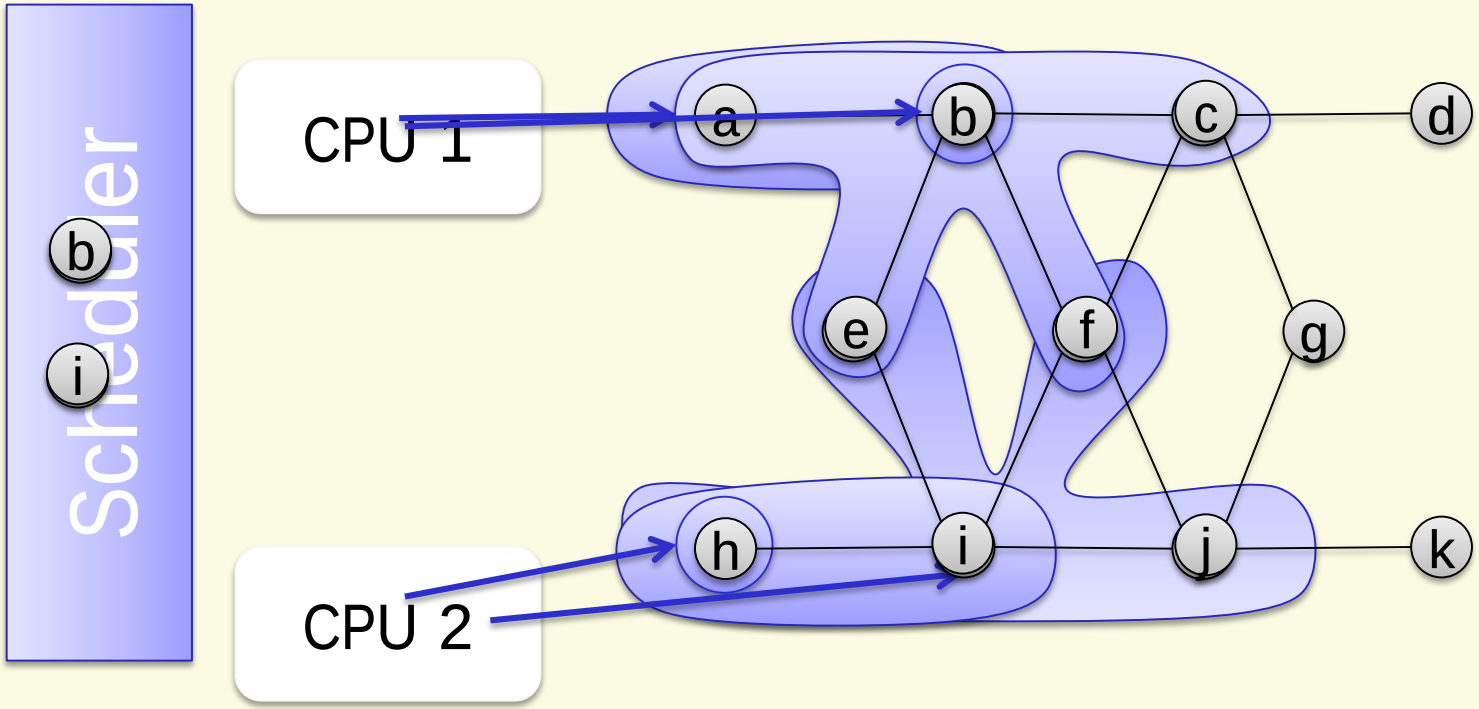
```
label_prop(i, scope){
// Get Neighborhood data
  (Likes[i], W_ij, Likes[j]) ←scope;

// Update the vertex data
```

$$Likes[i] \leftarrow \sum_{j \in Friends[i]} W_{ij} \times Likes[j];$$

```
// Reschedule Neighbors if needed
 if Likes[i] changes then
reschedule_neighbors_of(i);
}
```

The Scheduler

The **scheduler** determines the order that vertices are updated.

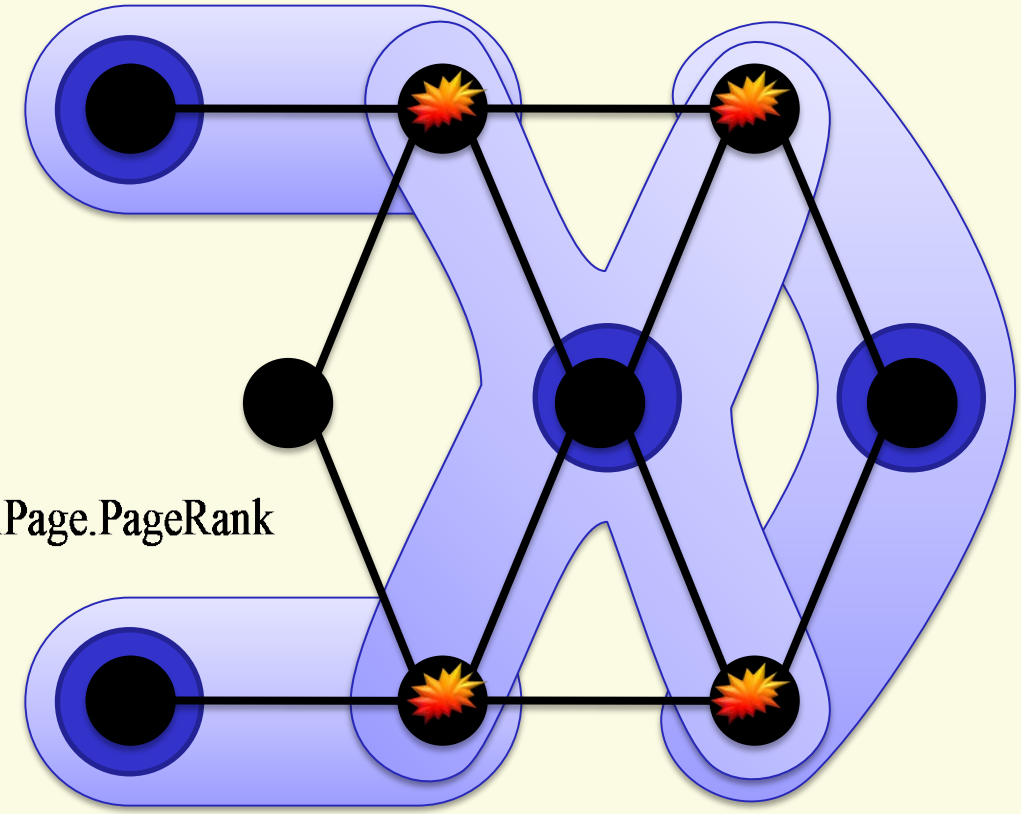


The process repeats until the scheduler is empty.

Ensuring Race-Free Code

- How much can computation **overlap**?

```
Pagerank(scope) {
  vertex.PageRank = α
  ForEach inPage:
     vertex.PageRank += (1 – α) × inPage.PageRank
  vertex.PageRank = tmp  }
```



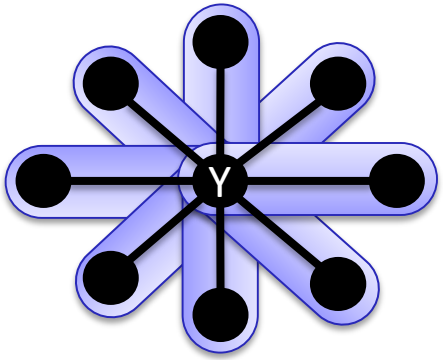
GAS Decomposition

Gather (Reduce)

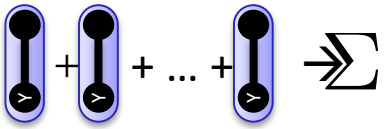
Accumulate information about neighborhood

User Defined:

- Gather(Y → Σ
- $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$



Parallel Sum

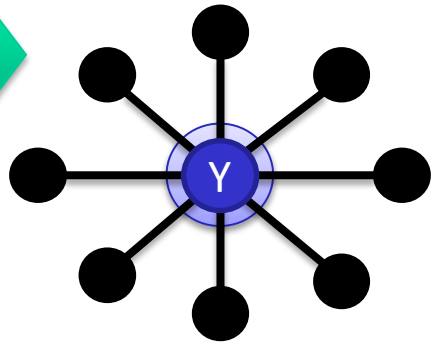


Apply

Apply the accumulated value to center vertex

User Defined:

- Apply(Y Σ) → Y'

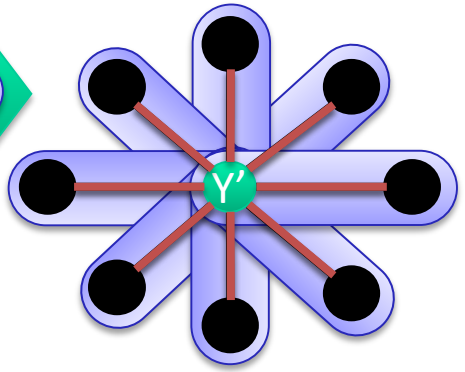


Scatter

Update adjacent edges and vertices.

User Defined:

- Scatter(Y' →)



Update Edge Data & Activate Neighbors

PageRank in PowerGraph

$$R[i] = 0.15 + \sum_{j \in \mathrm{Nbrs}(i)} w_{ji} R[j]$$

PowerGraph_PageRank(i)

Gather(j → i) **:** return $w_{ji} * R[j]$

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

Apply(i, Σ) **:** R[i] = 0.15 + Σ

Scatter(i → j) **:**

if *R[i]* changed then trigger *j* to be **recomputed**

Vertex-centric models vs. MapReduce

- ✓ Vertex centric: think like a vertex; MapReduce: think like a graph
- ✓ Vertex centric: maximize parallelism – asynchronous, minimize data shipment via message passing; support iterations

  MapReduce: inefficiency caused by blocking; distributing intermediate results (all to all), unnecessary write/read; does not provide a mechanism to support iteration
- ✓ Vertex centric: limited to graphs; MapReduce: general
- ✓ Lack of global control: ordering for processing vertices in recursive computation, incremental computation, etc
- ✓ New programming models, have to re-cast algorithms in MapReduce, hard to reuse existing (incremental) algorithms

Can we do better?

Pregel vs GraphLab

- ✓ Distributed system models
  - Asynchronous model and Synchronous model
  - Well known tradeoff between two models
    - Synchronous: concurrency-control/failures easy, poor perf.
    - Asynchronous: concurrency-control/failures hard, good perf.
- ✓ Pregel is a synchronous system
  - No concurrency control, no worry of consistency
  - Fault-tolerance, check point at each barrier
- ✓ GraphLab is asynchronous system
  - Consistency of updates harder (sequential, vertex)
  - Fault-tolerance harder (need a snapshot with consistency)

From Vertex-centric models to Graph-centric models

Querying distributed graphs

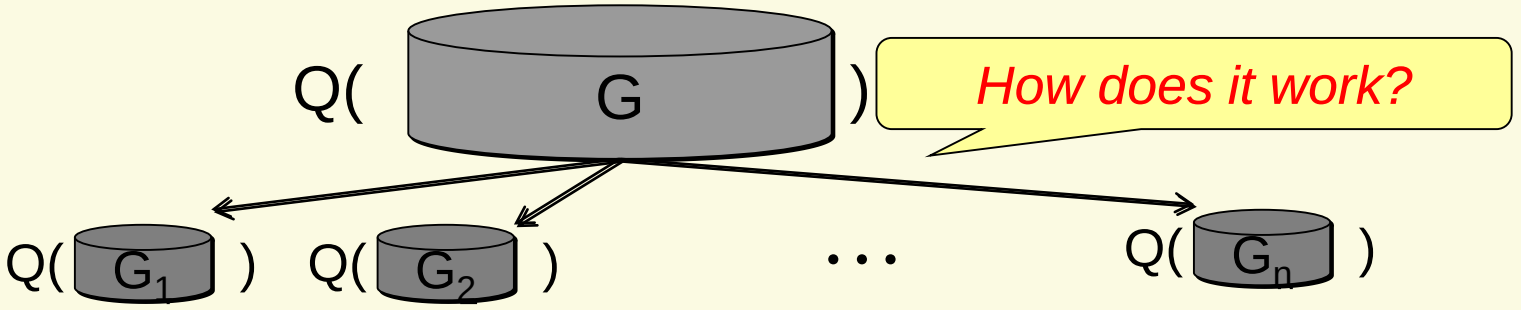
Given a big graph G, and n processors S1, …, Sn

- ✓ G is partitioned into fragments (G1, …, Gn)
- ✓ G is distributed to n processors: Gi is stored at Si

Parallel query answering

- ✓ Input: G = (G1, …, Gn), distributed to (S1, …, Sn), and a query Q
- ✓ Output: Q(G), the answer to Q in G

Each processor Si processes its local fragment Gi in parallel



Dividing a big G into small fragments of manageable size

GRAPE (GRAPh Engine)

manageable sizes

Divide and conquer

- ✓ partition G into fragments (G1, …, Gn), distributed to various sites

evaluate Q on smaller Gi

- ✓ upon receiving a query Q,
  - evaluate Q(Gi) *in parallel*
  - collect partial answers at a coordinator site, and assemble them to find the answer Q(G) in the entire G

Each machine (site) Si

- ✓ processes the same query Q,
- ✓ uses only data stored in its local fragment Gi

data-partitioned parallelism

Partial evaluation

compute **f**(x) ⇒ **f**(**s**, **d**)

the part of known input

yet unavailable input

- ✓ conduct the part of com[illegible] only on **s**
- ✓ generate a partial answer

a residual function

at each site, Gi as the known input

- ✓ Partial evaluation in distribut[illegible]query processing
  - evaluate Q(Gi) *in parallel*
  - collect partial [illegible]tches at a coordinator site, and assemble them to find the [illegible]tire G

Gj as the yet unavailable input

as residual functions

The connection between partial evaluation and parallel processing

Coordinator

Each machine (site) Si is either

- ✓ a coordinator
- ✓ a worker: conduct local computation and produce partial answers

Coordinator: receive/post queries, control termination, and assemble answers

- ✓ Upon receiving a query Q
  - post Q to all workers
  - Initialize a status flag for each worker, mutable by the worker
- ✓ Terminate the computation when all flags are true
  - Assemble partial answers from workers, and produce the final answer Q(G)

Termination, partial answer assembling

Workers

Worker: conduct local computation and produce partial answers

- ✓ upon receiving a query Q,
  - evaluate Q(Gi) *in parallel*
  - send messages to request data for "border nodes"

use local data Gi only

With edges to other fragments

- ✓ Incremental comp… w messages M
  - evaluate Q(Gi + M) *in parallel*
- ✓ set its flag true if no more changes to partial results, and send the partial answer to the coordinator

Incremental computation

This step repeats until the partial answer at site Si is ready

Local computation, partial evaluation, recursion, partial answers

Reachability and regular path queries

- ✓ Reachability
  - Input: A directed graph G, and a pair of nodes s and t in G
  - Question: Does there exist a path from s to t in G?

O(|V| + |E|) time

- ✓ Regular path
  - Input: A node-labelled directed graph G, a pair of nodes s and t in G, and a regular expression R
  - Question: Does there exist a path p from s to t such that the labels of adjacent nodes on p form a string in R?

O(|G| |R|) time

Parallel algorithms?

Costly when G is big

Reachability queries

Worker: conduct local computation and produce partial answers

- ✓ upon receiving a query Q,
  - evaluate Q(Gi) *in parallel*
  - send messages to request data for "border nodes"

With edges to other fragments

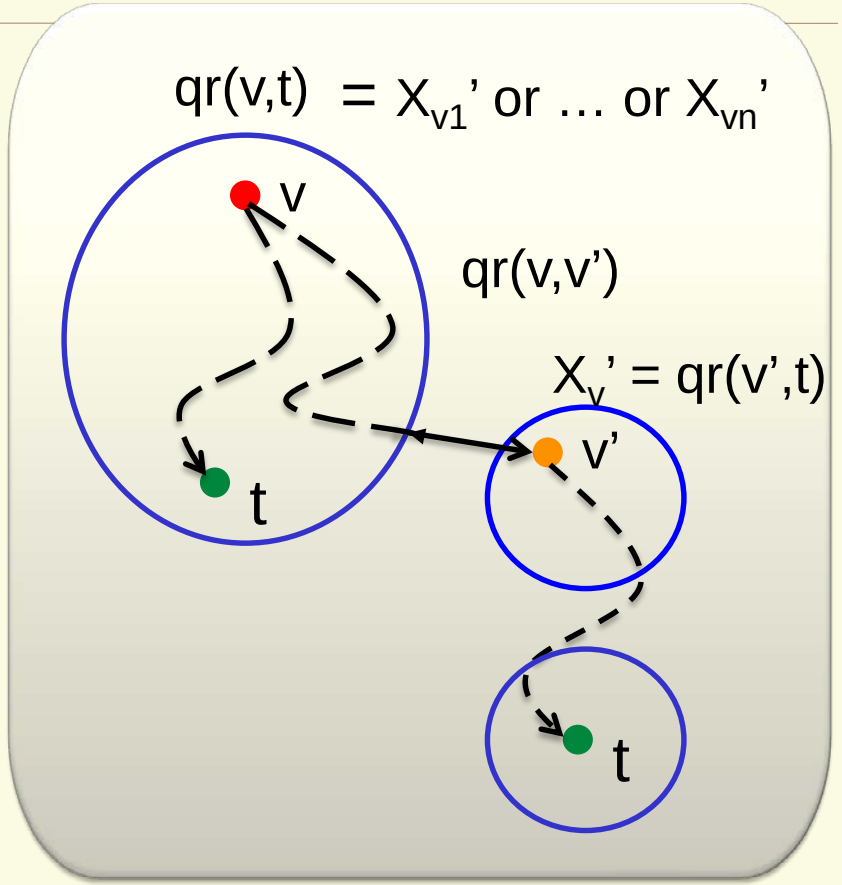
- ✓ Boolean formulas as partial answers
  - For each node v in Gi, a Boolean variable X_v, indicating whether v reaches destination t
  - The truth value of X_v can be expressed as a Boolean formula over X_{vb}

Border nodes in Gi

Local computation: computing the value of X_v in Gi

Boolean variables

- ✓ Locally evaluate each qr(v,t) in Gi in parallel:
- ✓ for each in-node v' in Fi, decides whether v' reaches t; introduce a Boolean variable to each v'
- ✓ Partial answer to qr(v,t): a set of Boolean formula, disjunction of variables of v' to which v can reach



Partial evaluation by introducing Boolean variables

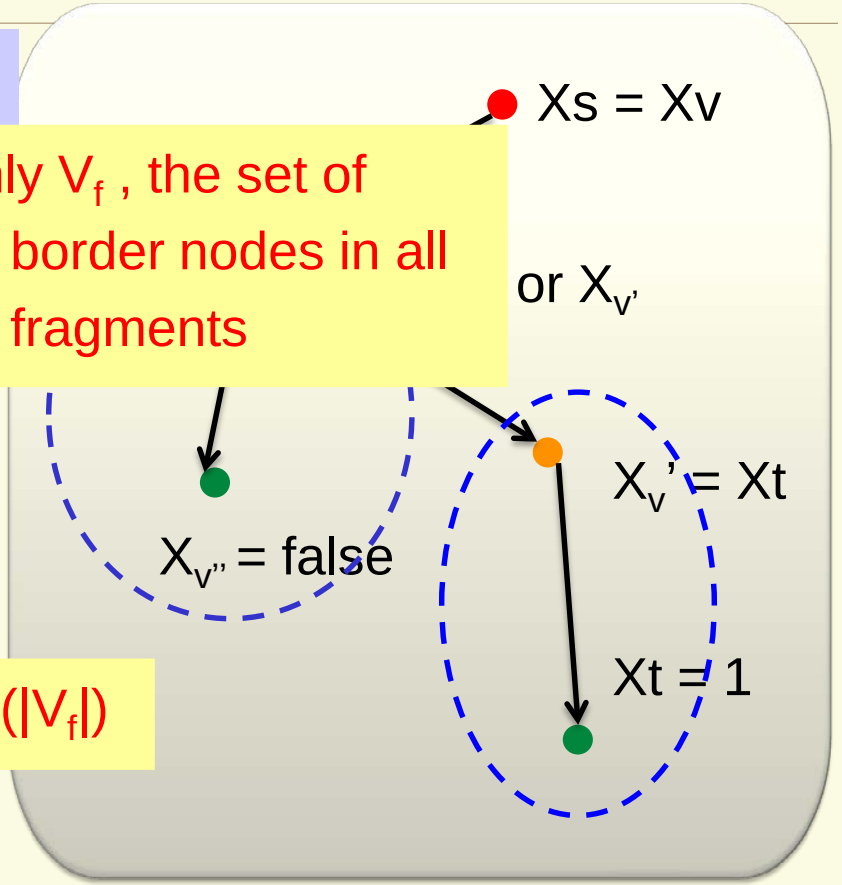
Distributed reachability: assembling

Coordinator: Assemble partial answers from workers, and produce the final answer Q(G

- ✓ Collect the Boolean equations at coordinator
- ✓ solve a system of linear Boolean equation by using a dependency graph
- ✓ qr(s,t) is true if and only if X_s = true in the equation system

Only V_f , the set of border nodes in all fragments

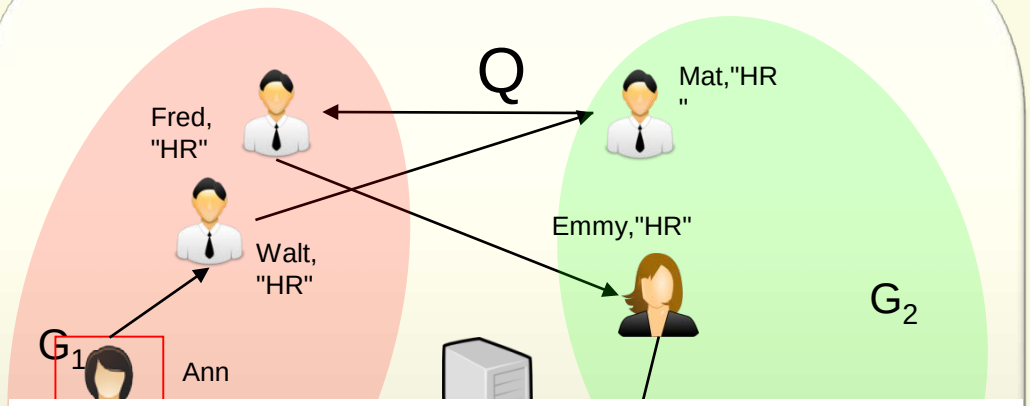
$O(|V_f|)$



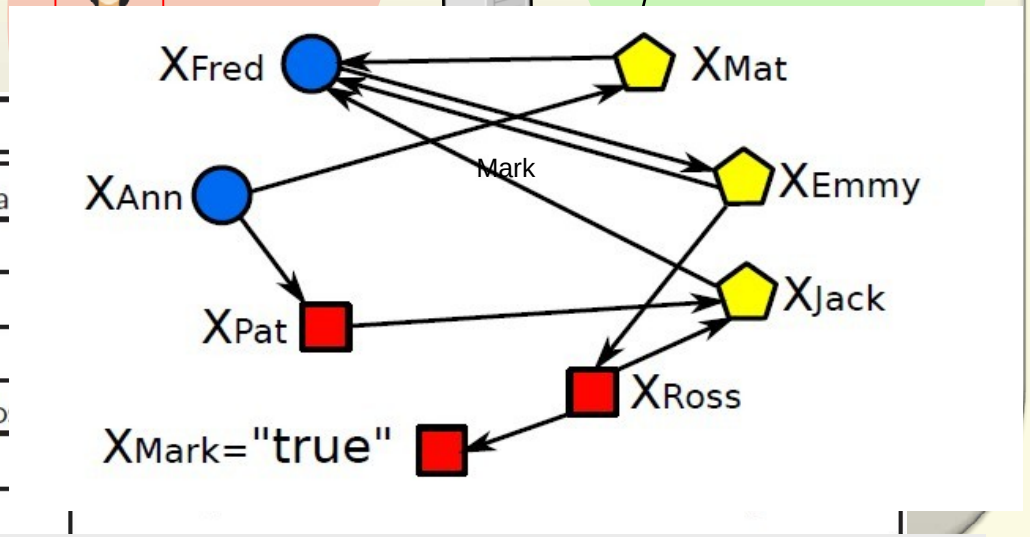
Assembling partial answers

Example

1. Dispatch Q to fragments (at Sc)
2. Partial evaluation: generating Boolean



F_i	$F_i.I$	rf
F_1	Ann	$x_{Pat} \vee x_{Ma}$
	Fred	x_{Emmy}
F_2	Mat	x_{Fred}
	Jack	x_{Fred}
	Emmy	$x_{Fred} \vee x_{Ro}$
F_3	Ross	true
	Pat	x_{Jack}



No messages between different fragments

Reachability queries in GRAPE

- ✓ upon receiving a query Q,
  - evaluate Q(Gi) *in parallel*
  - collect partial answers at a coordinator site, and assemble them to find the answer Q(G) in the entire G

G_m: the largest fragment

Complexity analysis

- ✓ Parallel computation: in O(|V_f||G_m|) time — speedup? $|G_m| = |G|/n$
- ✓ One r
- ✓ Data s … ordinator; no m

Approximation algorithm

Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. Technical report, Swedish Institute of Computer Science, 2013

Complication: minimizing V_f? An NP-complete problem.

Think like a graph

Graph pattern matching by graph simulation

- ✓ Input: A directed graph G, and a graph pattern Q
- ✓ Output: the maximum simulation relation R

- ✓ Maximum simulation relation: always exists and is unique
  - If a match relation exists, then there exists a maximum one
  - Otherwise, it is the empty set – still maximum

- ✓ Complexity: $O((|V| + |V_Q|)(|E| + |E_Q|)$

- ✓ The output is a unique relation, possibly of size $|Q||V|$

A parallel algorithm in GRAPE?

Coordinator

Given a big graph G, and n processors S1, …, Sn

- ✓ G is partitioned into fragments (G1, …, Gn)
- ✓ G is distributed to n processors: Gi is stored at Si

Coordinator: Upon receiving a query Q

- ✓ post Q to all workers
- ✓ Initialize a status flag for each worker, mutable by the worker

- ✓ Boolean formulas as partial answers
  - For each node v in Gi and each pattern node u in Q, a Boolean variable X(u, v), indicating whether matches u
  - The truth value of X(u, v) can be expressed as a Boolean formula over X(u', v'), for border nodes v' in V_f

Again, Boolean formulas

Worker: initial evaluation

Worker: conduct local computation and produce partial answers

- ✓ upon receiving a query Q,
  - evaluate Q(Gi) *in parallel*
  - send messages to request data for “border nodes”

use local data Gi only

- ✓ Local evaluation
  - Invoke an existing algorithm to compute Q(Gi)
  - Minor revision: incorporating Boolean variables

With edges from other fragments

- ✓ Messages:
  - For each node to which there is an edge from another fragment Gj, send the truth value of its Boolean variable to Gj

Partial evaluation: using an existing algorithm

Worker: incremental evaluation

Recursive computation

- ✓ Repeat until the truth values of all Boolean variables in Gi are determined

Use an existing incremental algorithm

  - evaluate Q(Gi + M) *in parallel*

Messages from other fragments

- ✓ set its flag true and send partial answer Q(Gi) to the coordinator
- ✓ Termination:

Partial answer assembling

Coordinator:

- ✓ Terminate the computation when all flags are true
- ✓ The union of partial answers from all the workers is the final answer Q(G)

Incremental computation, recursion, termination

Graph simulation in GRAPE

- ✓ Input: G = (G1, …, Gn), a pattern query Q
- ✓ Output: the unique maximum match of Q in G

- ✓ Performance guarantees
  - Response time: $O((|V_Q| + |V_m|)(|E_Q| + |E_m|)|V_Q||V_f|)$
  - the total amount of data shipped is in $O(|V_f||V_Q|)$

where

- $Q = (V_Q, E_Q)$ — small
- $G_m = (V_m, E_m)$: the largest fragment in G — |G|/n

with 20 machines, 55 times faster than first collecting data and then using a centralized algorithm

parallel query processing with performance guarantees

GRAPE vs. other parallel models

- ✓ Reduce unnecessary computation and data shipment
  - Message passing only between fragments, vs all-to-all (MapReduce) and messages between vertices
  - Incremental computation: on the entire fragment;
- ✓ Think like a graph, via minor revisions of existing algorithms;
  - no need to to re-cast algorithms in MapReduce or BSP
  - Iterative computations: inherited from existing ones
- ✓ Flexibility: MapReduce and vertex-centric models as special cases
  - MapReduce: a single Map (partitioning), multiple Reduce steps by capitalizing on incremental computation
  - Vertex-centric: local computation can be implemented this way

Implement a GRAPE platform?

Summing up

Summary and review

- ✓ What is the MapReduce framework? Pros? Pitfalls?
- ✓ Develop algorithms in MapReduce
- ✓ What are vertex-centric models for querying graphs? Why do we need them?
- ✓ What is GRAPE? Why does it need incremental computation? How to terminate computation in GRAPE?
- ✓ Develop algorithms in vertex-centric models and in GRAPE.
- ✓ Compare the four parallel models: MapReduce, PBS, vertex-centric, and GRAPE