Asynchronous Graph Processing

CompSci 590.03
Instructor: Ashwin Machanavajjhala

(slides adapted from Graphlab talks at [UAI'10](#) & [VLDB '12](#)
and Gouzhang Wang's talk at CIDR 2013)

Duke UNIVERSITY

Recap: Pregel

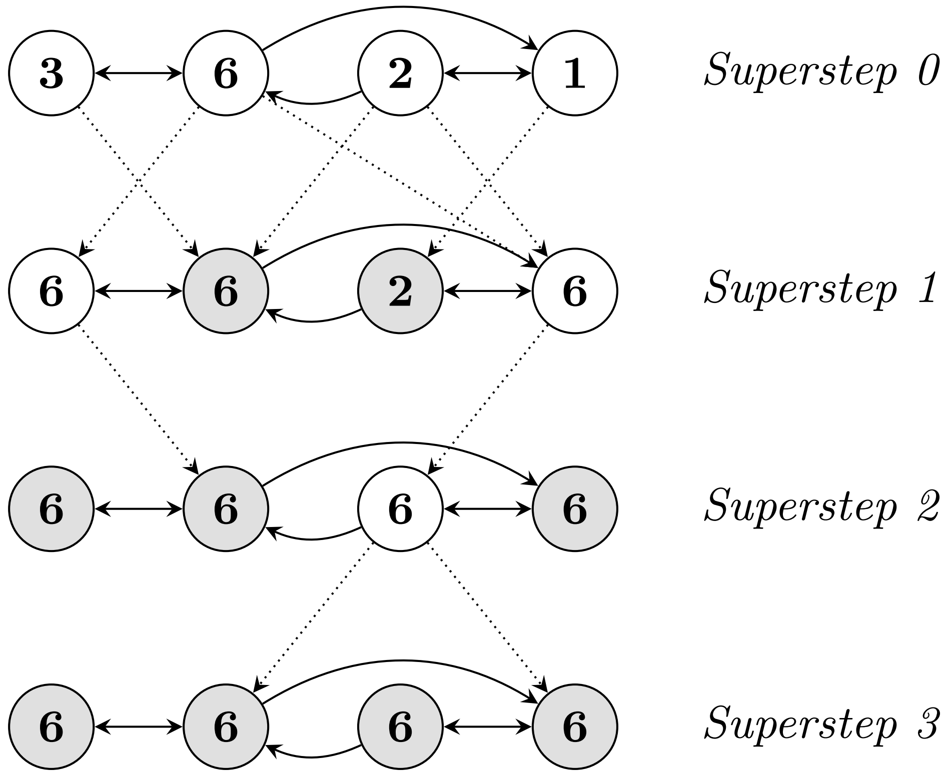
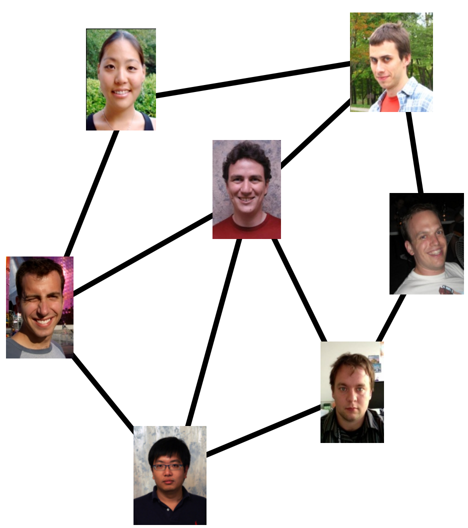


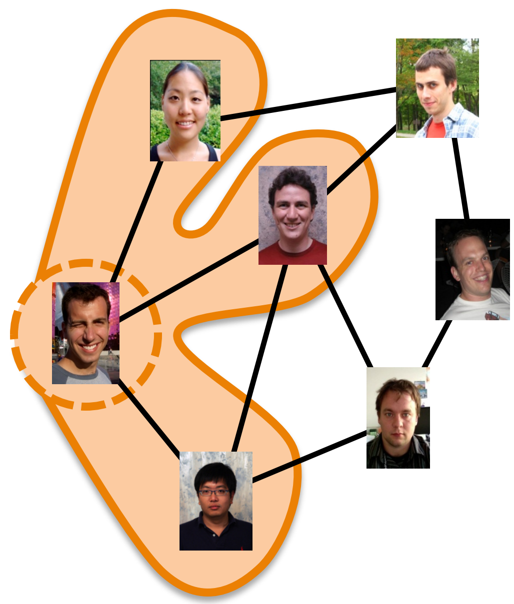
Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

Graph Processing

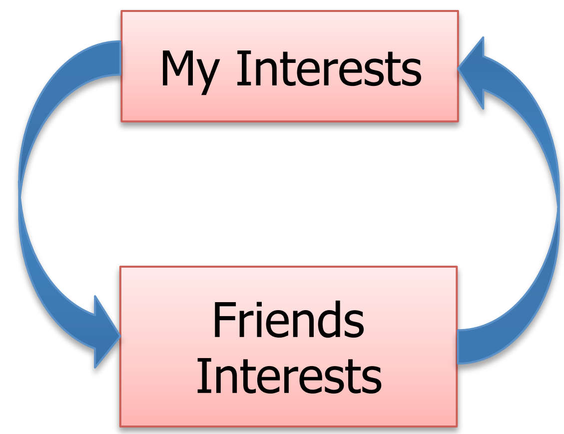
Dependency Graph



Local Updates



Iterative Computation



Duke UNIVERSITY

This Class

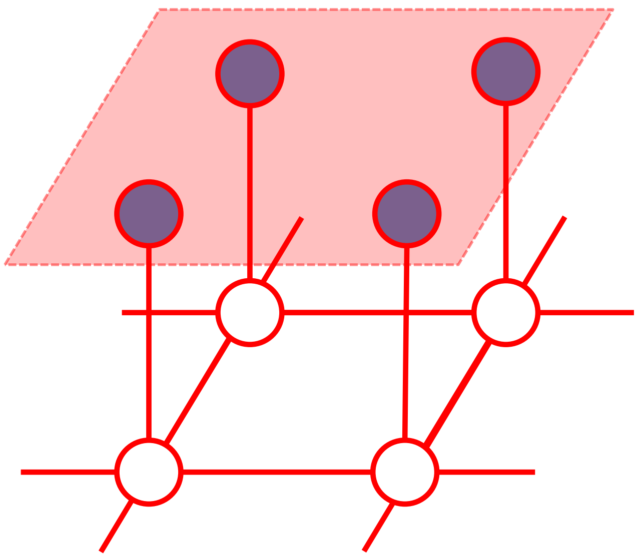
- Asynchronous Graph Processing

Duke UNIVERSITY

Example: Belief Propagation

$$p(x_1, x_2, \cdots, x_n) \propto \prod_{u \in V} \phi_u(x_u) \cdot \prod_{(u,v) \in E} \phi_{u,v}(x_u, x_v)$$

- Want to compute marginal distribution at each node.

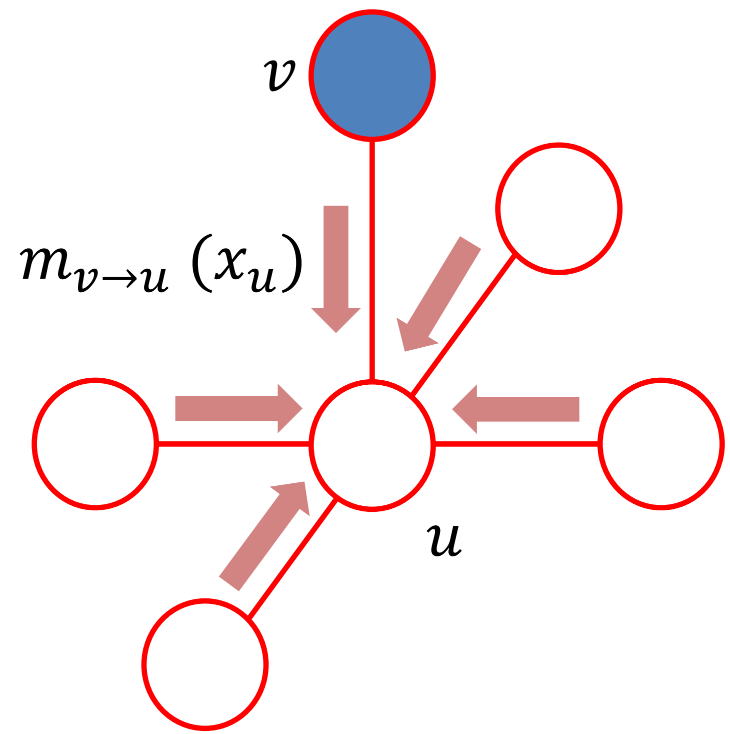


Duke UNIVERSITY

Belief Propagation

- Belief at a vertex depends on messages received from neighboring vertices

$$b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \to u}(x_u)$$



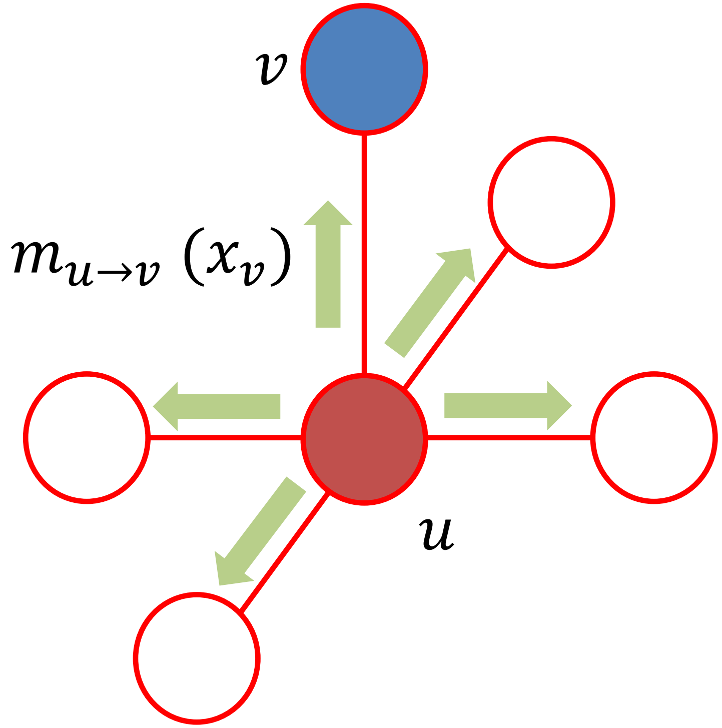
Duke UNIVERSITY

Belief Propagation

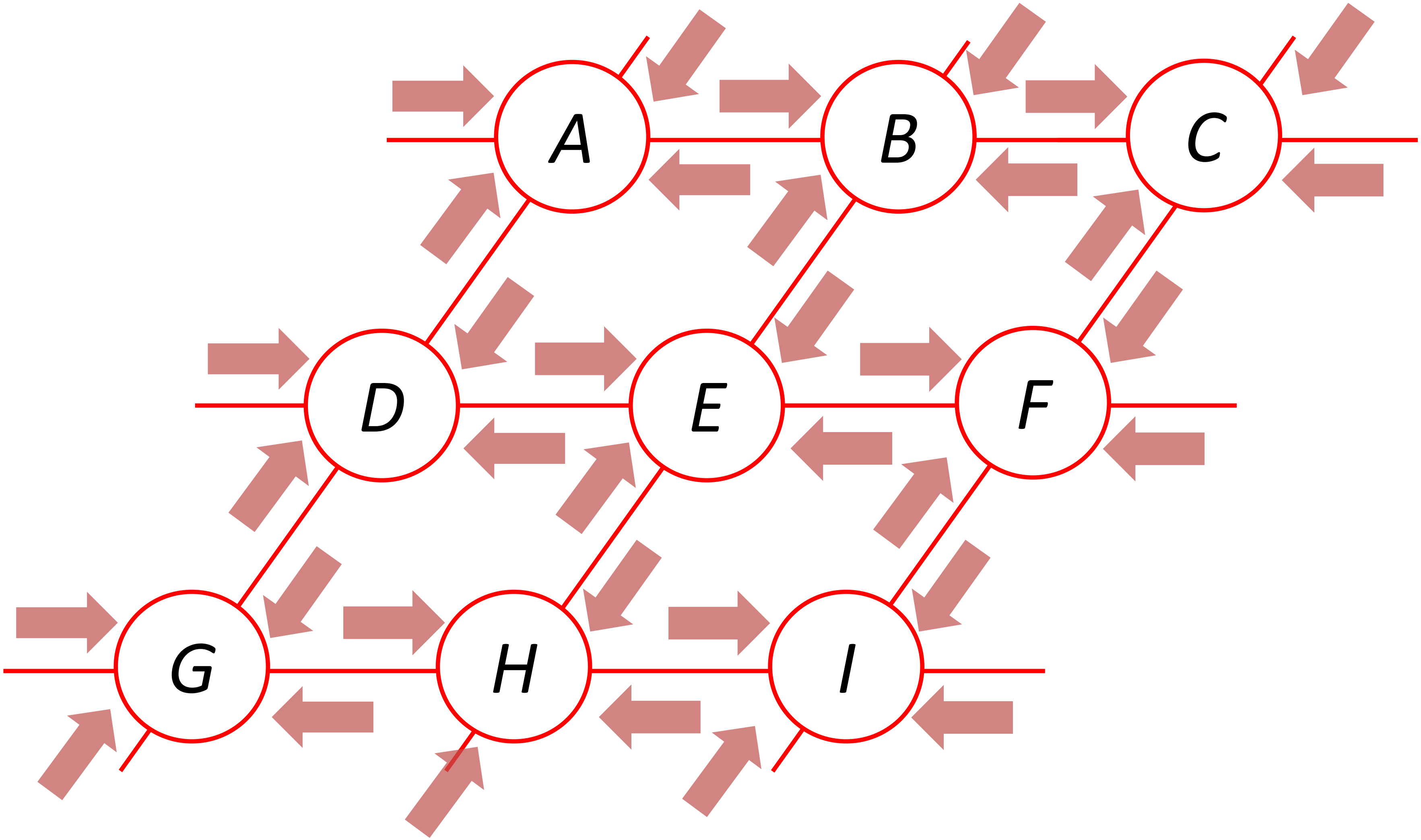
- Belief at a vertex depends on messages received from neighboring vertices

$$b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \to u}(x_u)$$

$$m_{u \to v}(x_v) \propto \sum_{x_u \in \Omega} \phi_{u,v}(x_u, x_v) \cdot \frac{b_u(x_u)}{m_{v \to u}(x_u)}$$



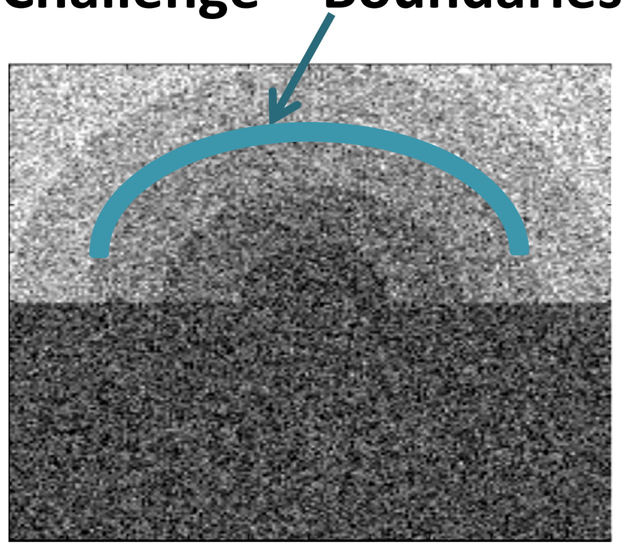
Original BP Algorithm



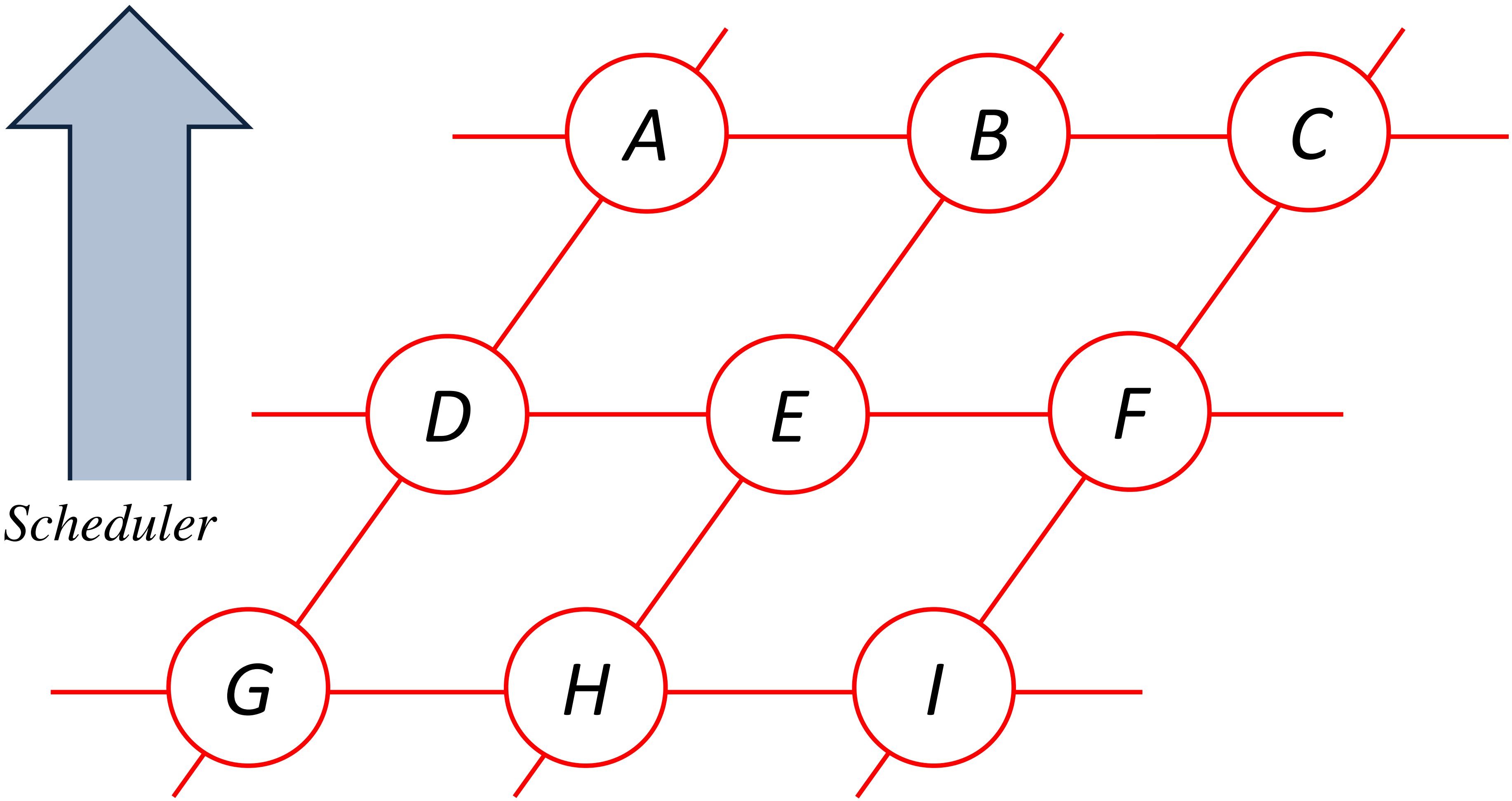
Original BP Algorithm can be inefficient

- Spends time updating nodes which have already converged

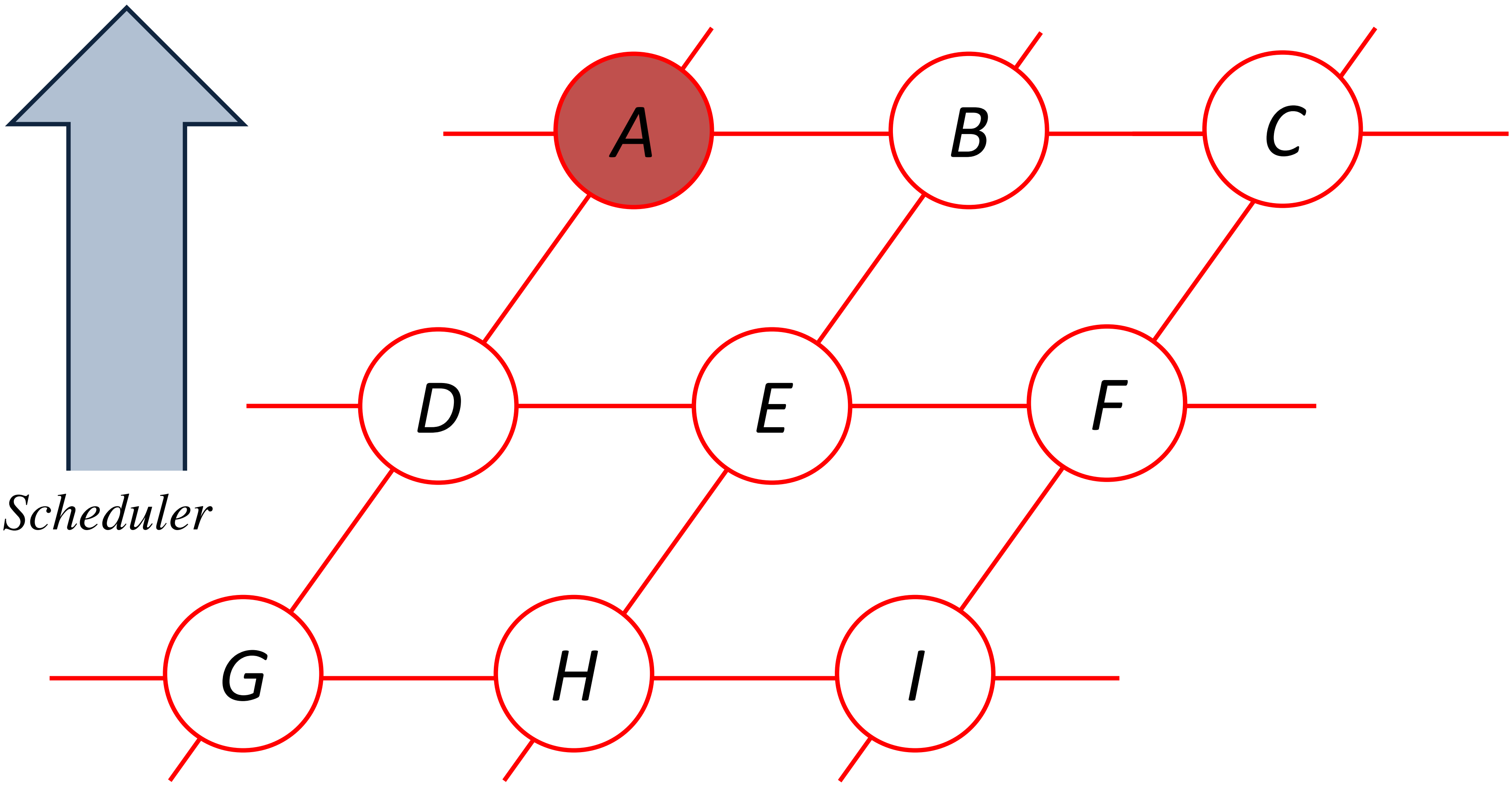
Challenge = Boundaries



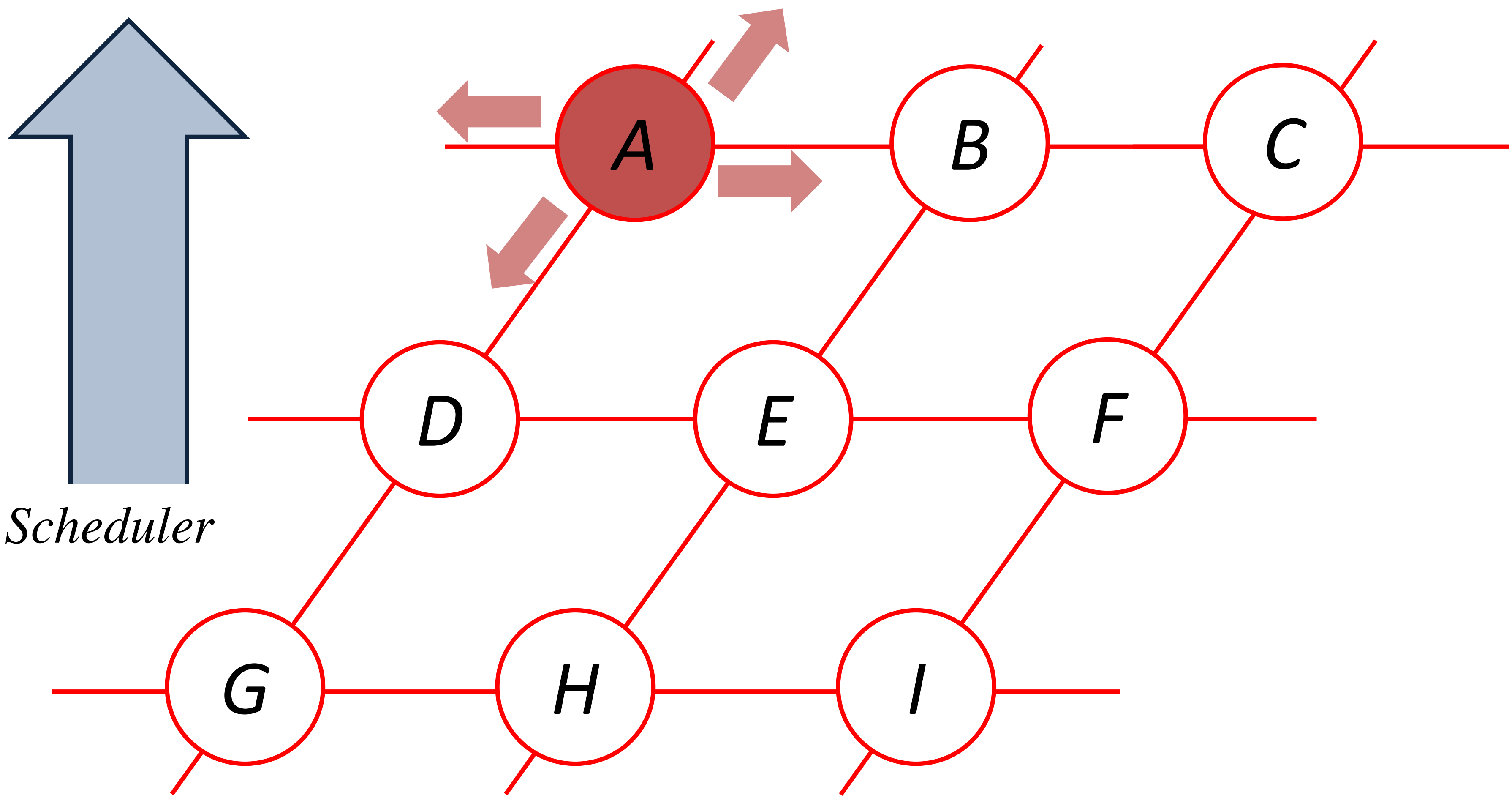
Residual BP Implementation



Residual BP Implementation

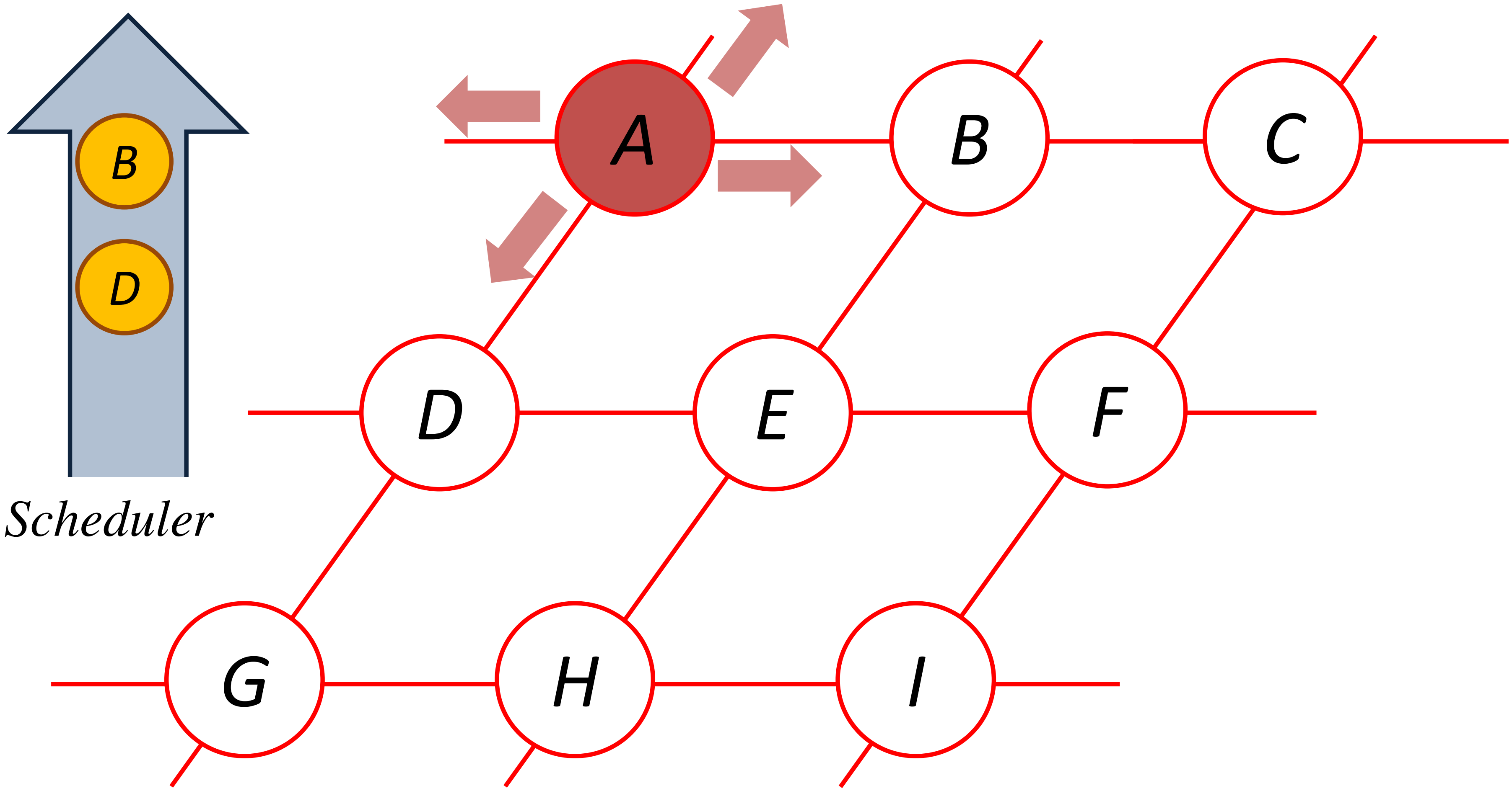


Residual BP Implementation

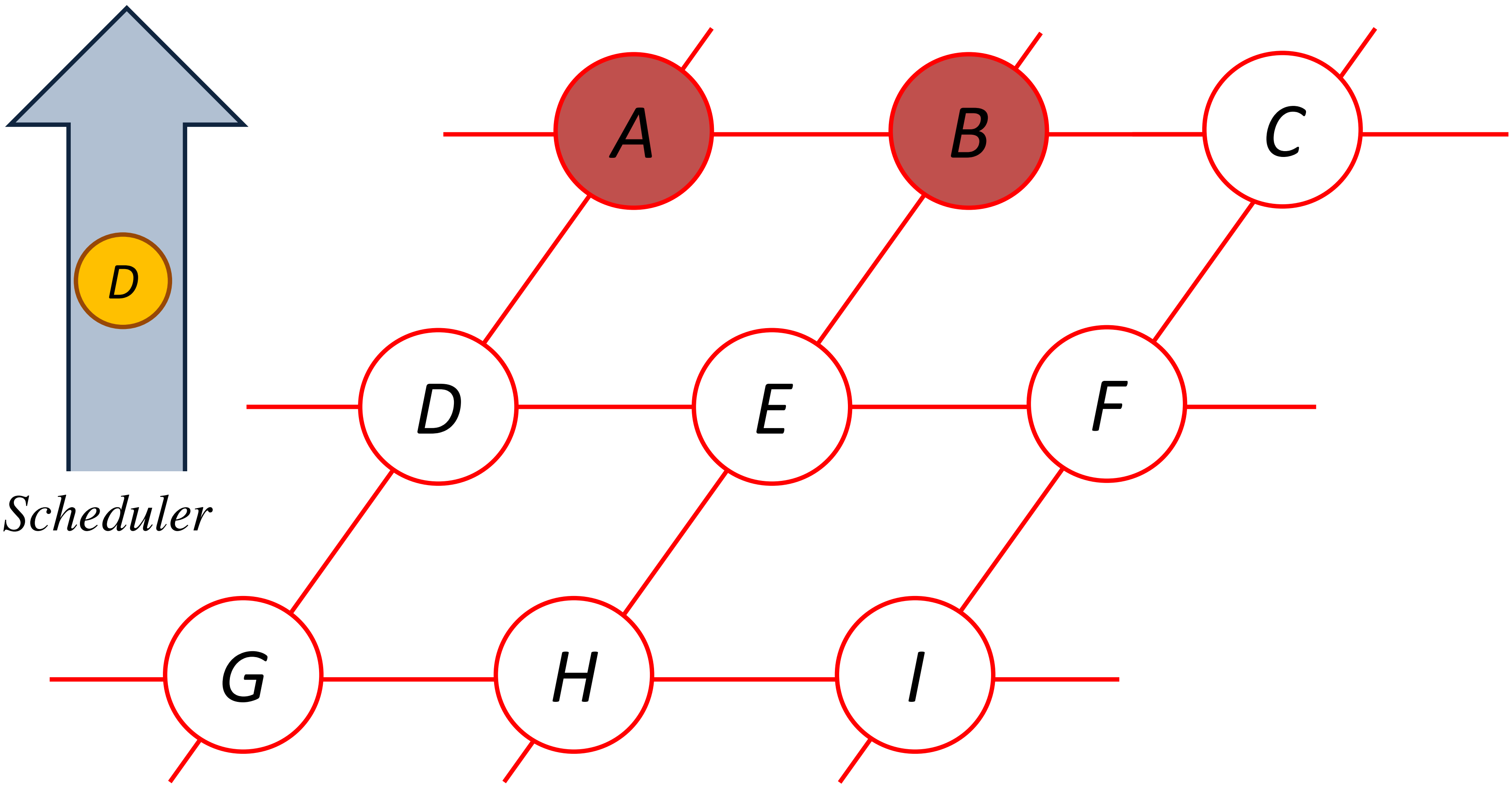


Residual BP Implementation

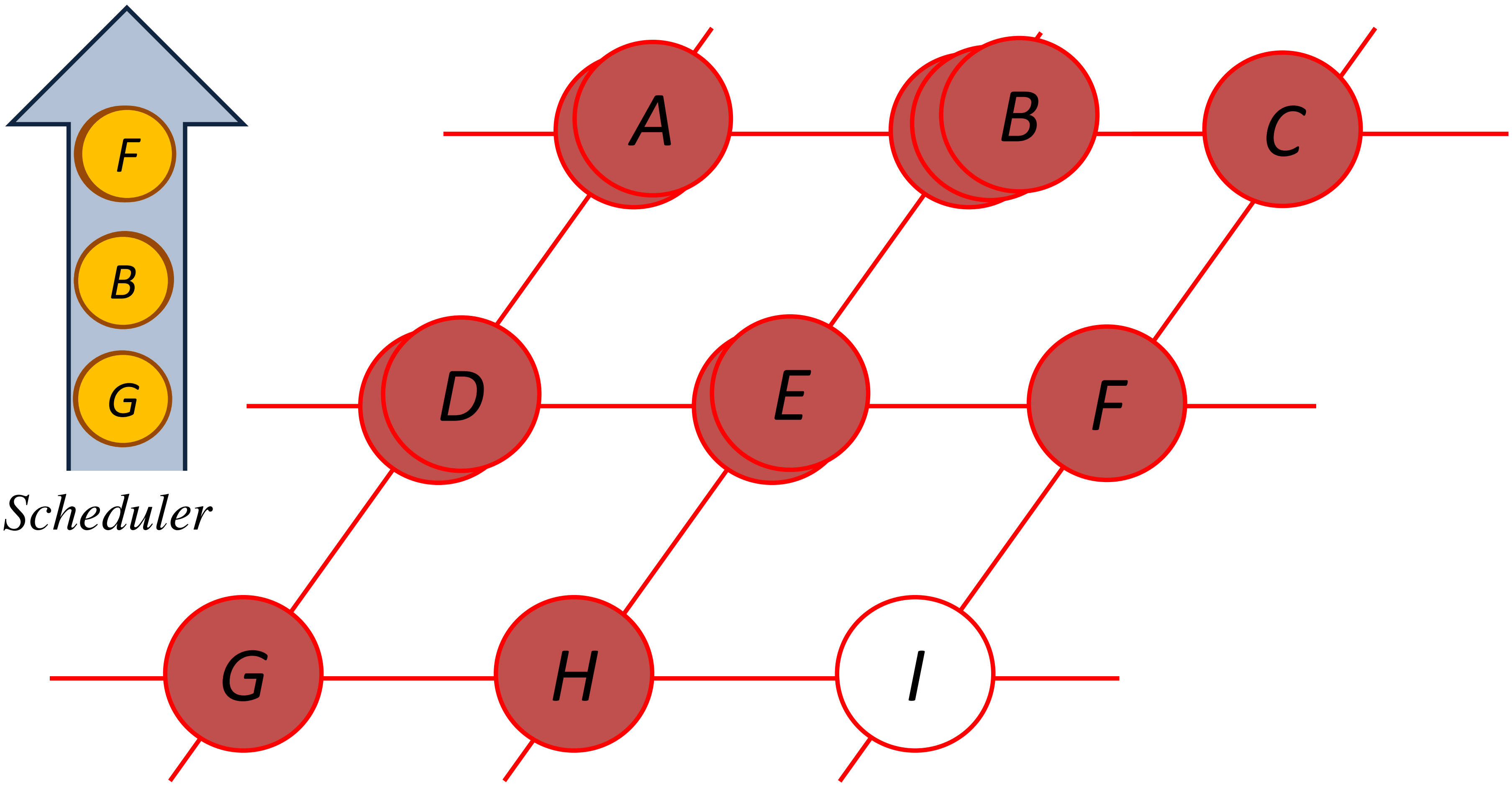
Ordering based on residual (max change in message value)



Residual BP Implementation



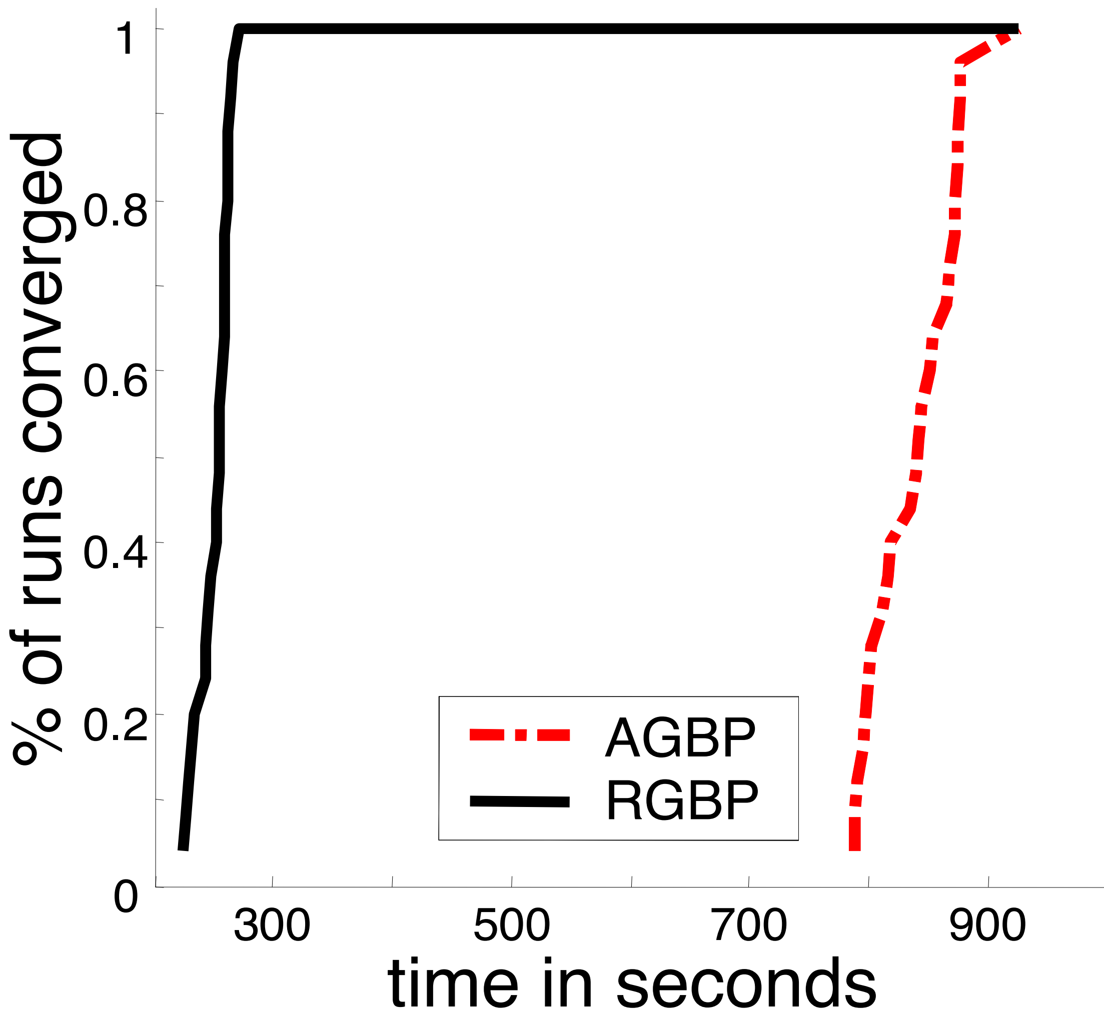
Residual BP Implementation



Duke UNIVERSITY

Residual BP converges faster

[Elidan et al UAI 2006]



Duke UNIVERSITY

Summary

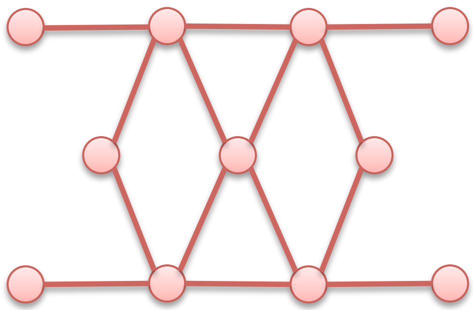
- Asynchronous serial graph algorithms can converge faster than synchronous parallel graph algorithms

- Is there a way to **correctly** transform asynchronous serial algorithms to run in a parallel setting?

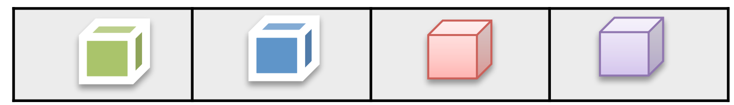
GRAPHLAB

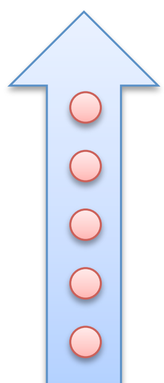
GraphLab

Data Graph

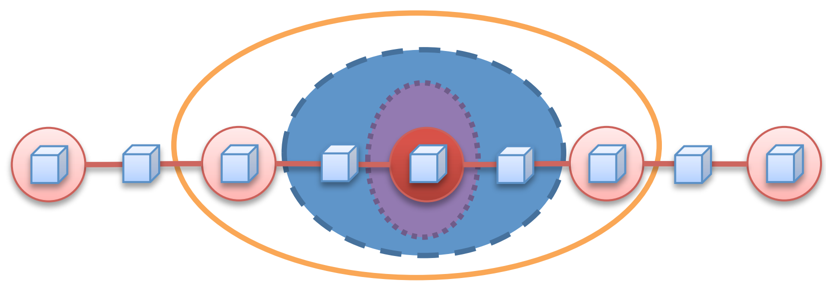


Shared Data Table





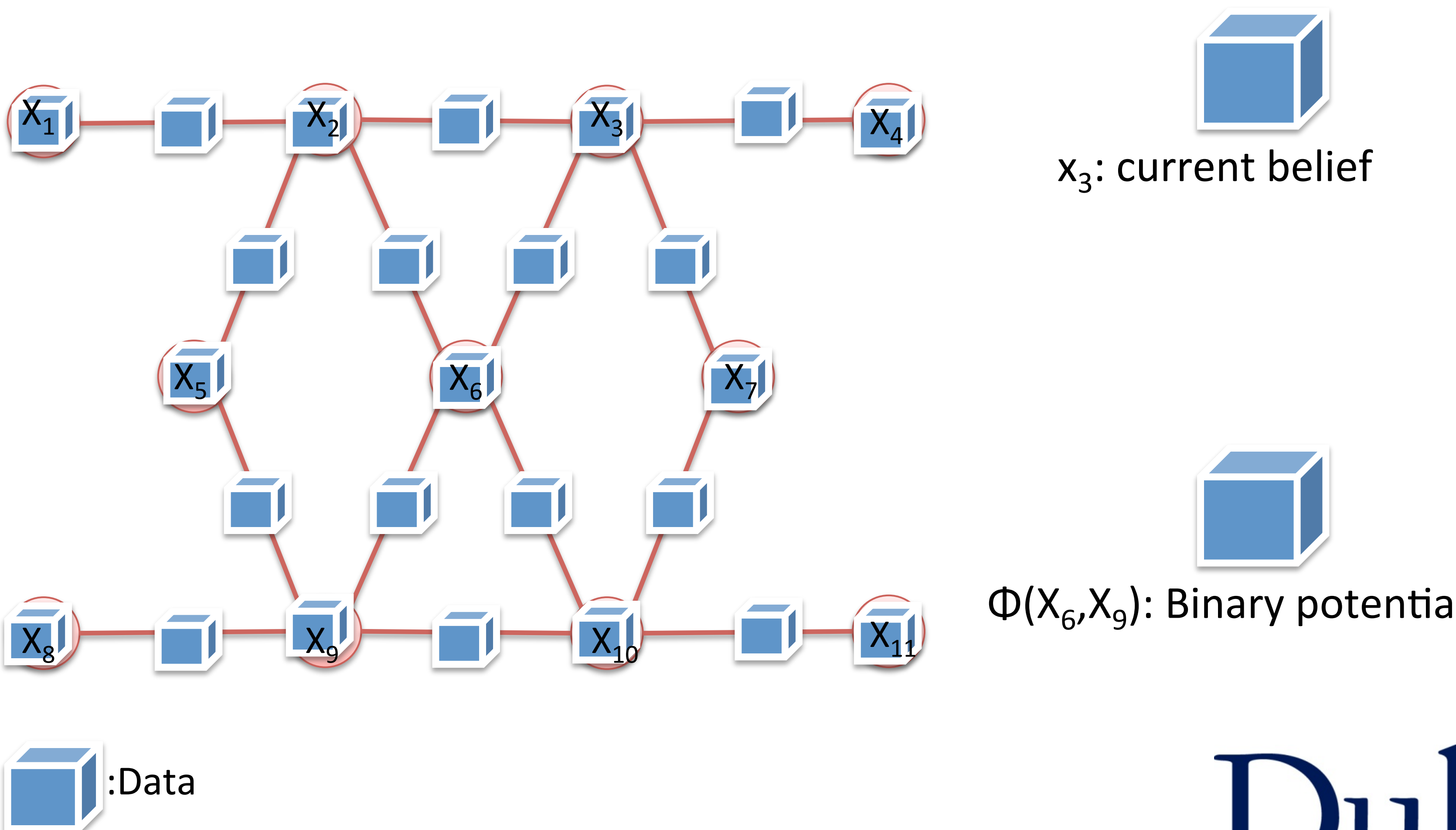
Scheduling



Update Functions and Scopes

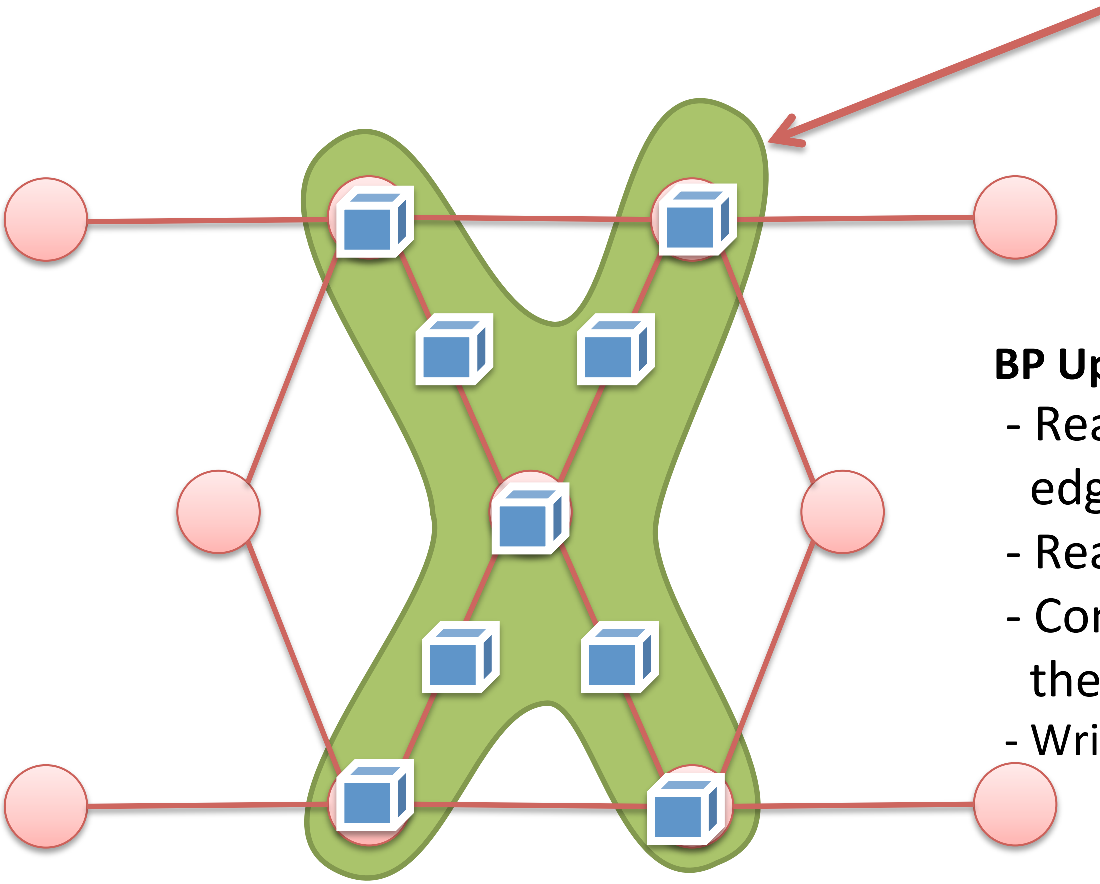
Data Graph

A **Graph** with data associated with every vertex and edge.



Update Functions

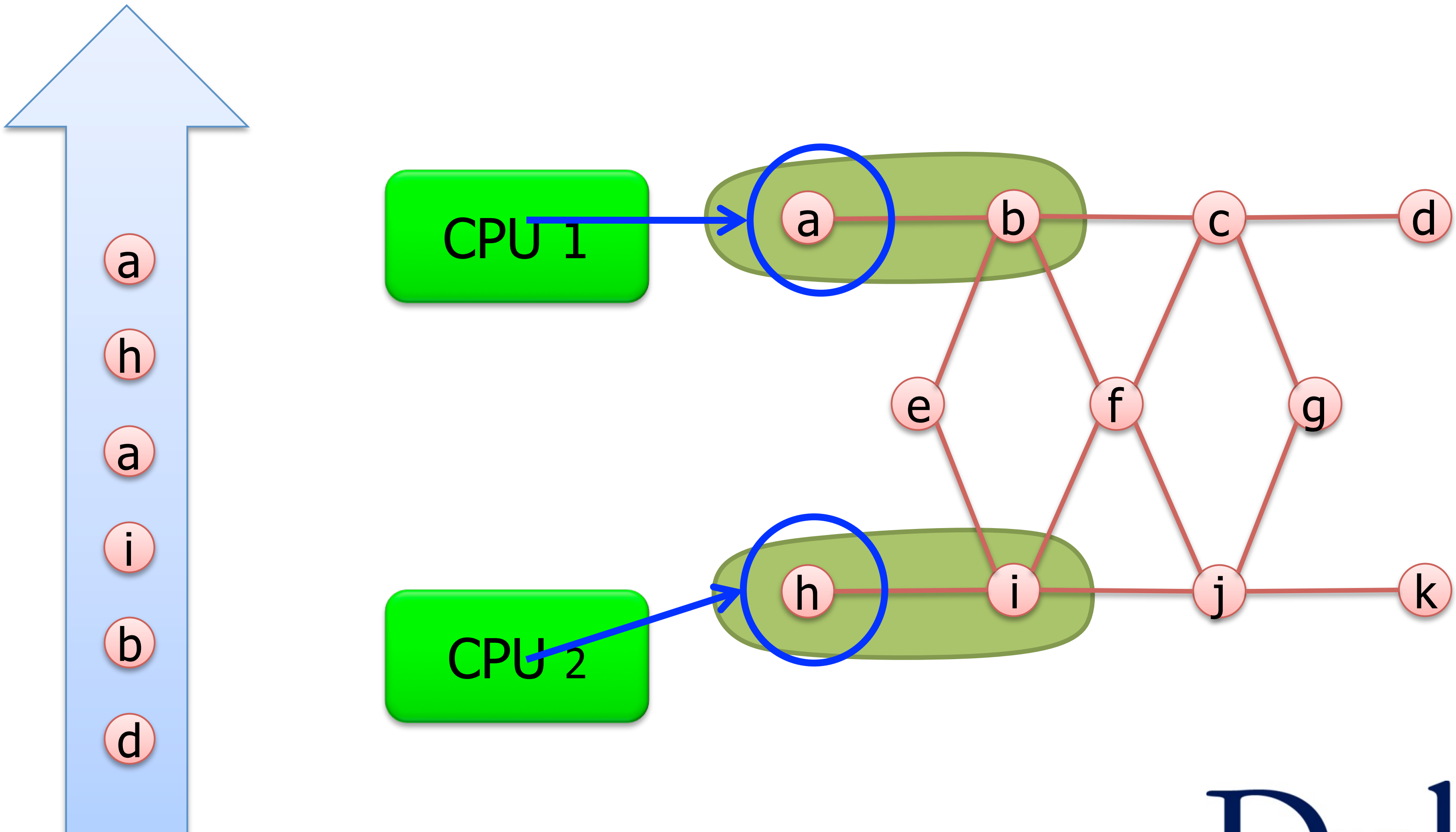
Update Functions are operations which are applied on a vertex and transform the data in the **scope** of the vertex



BP Update:

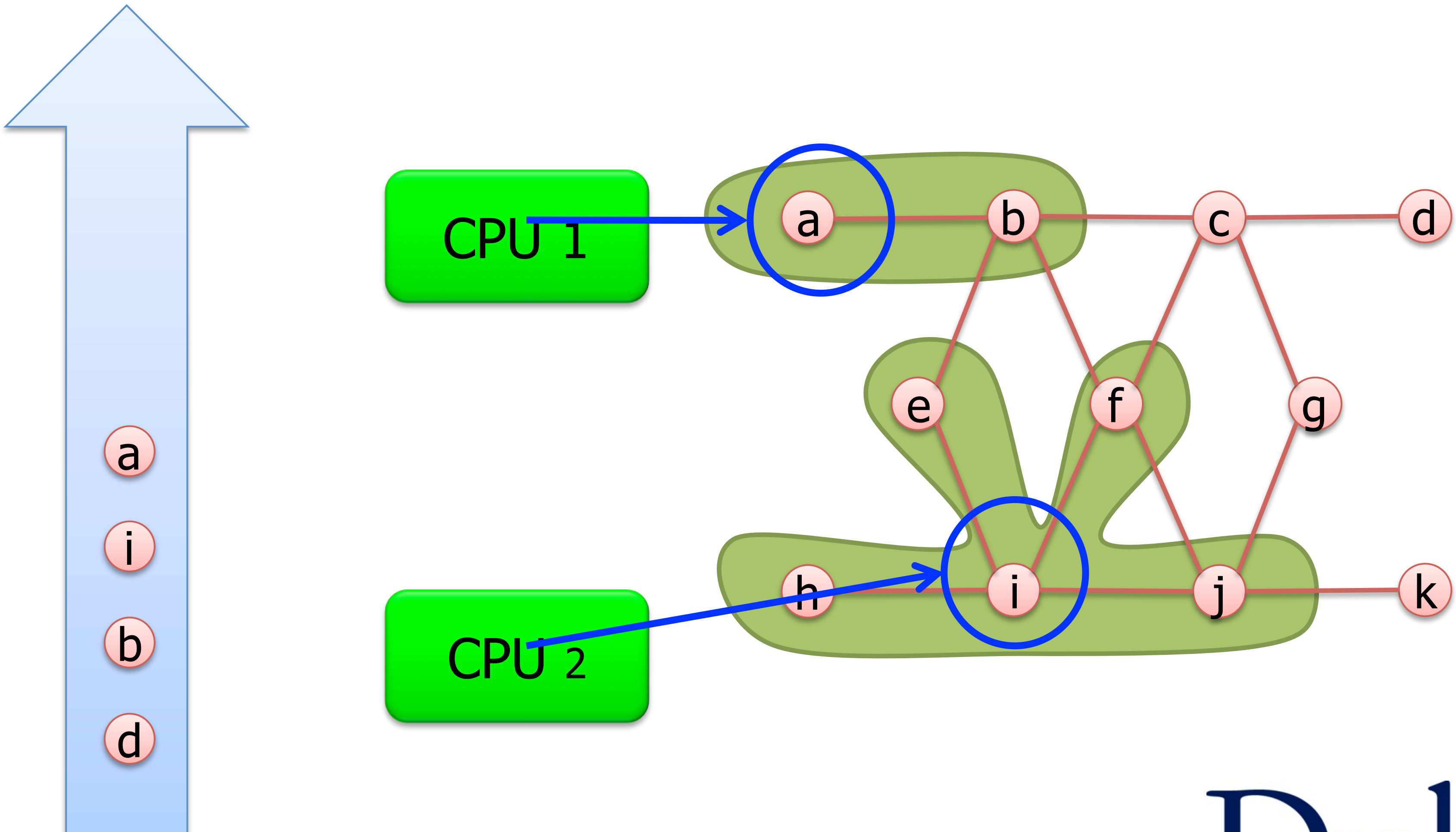
- Read messages on adjacent edges
- Read edge potentials
- Compute a new belief for the current vertex
- Write new messages on edges

Update Function Schedule



Duke UNIVERSITY

Update Function Schedule



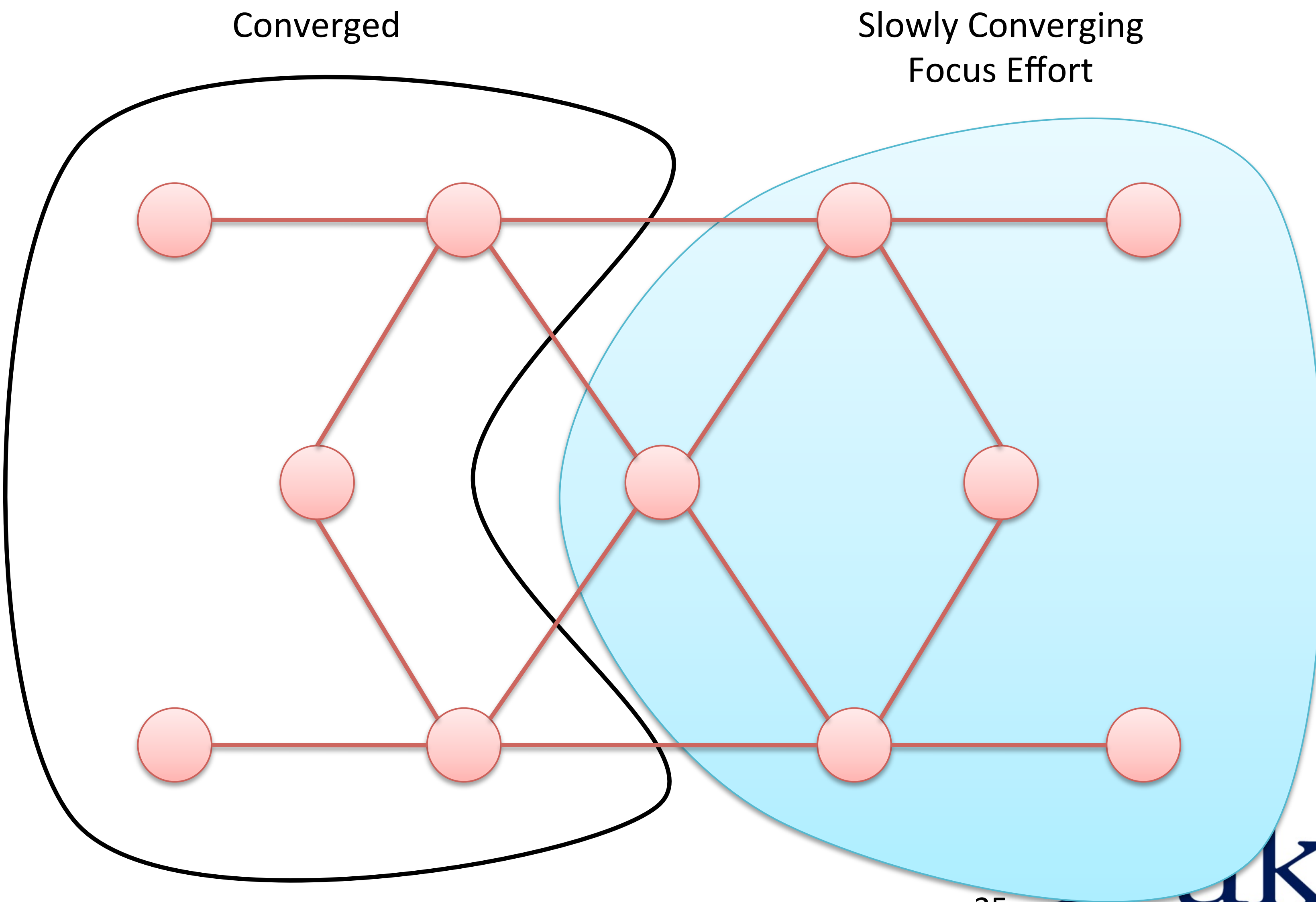
Static Schedule

Scheduler determines the
order of Update Function Evaluations

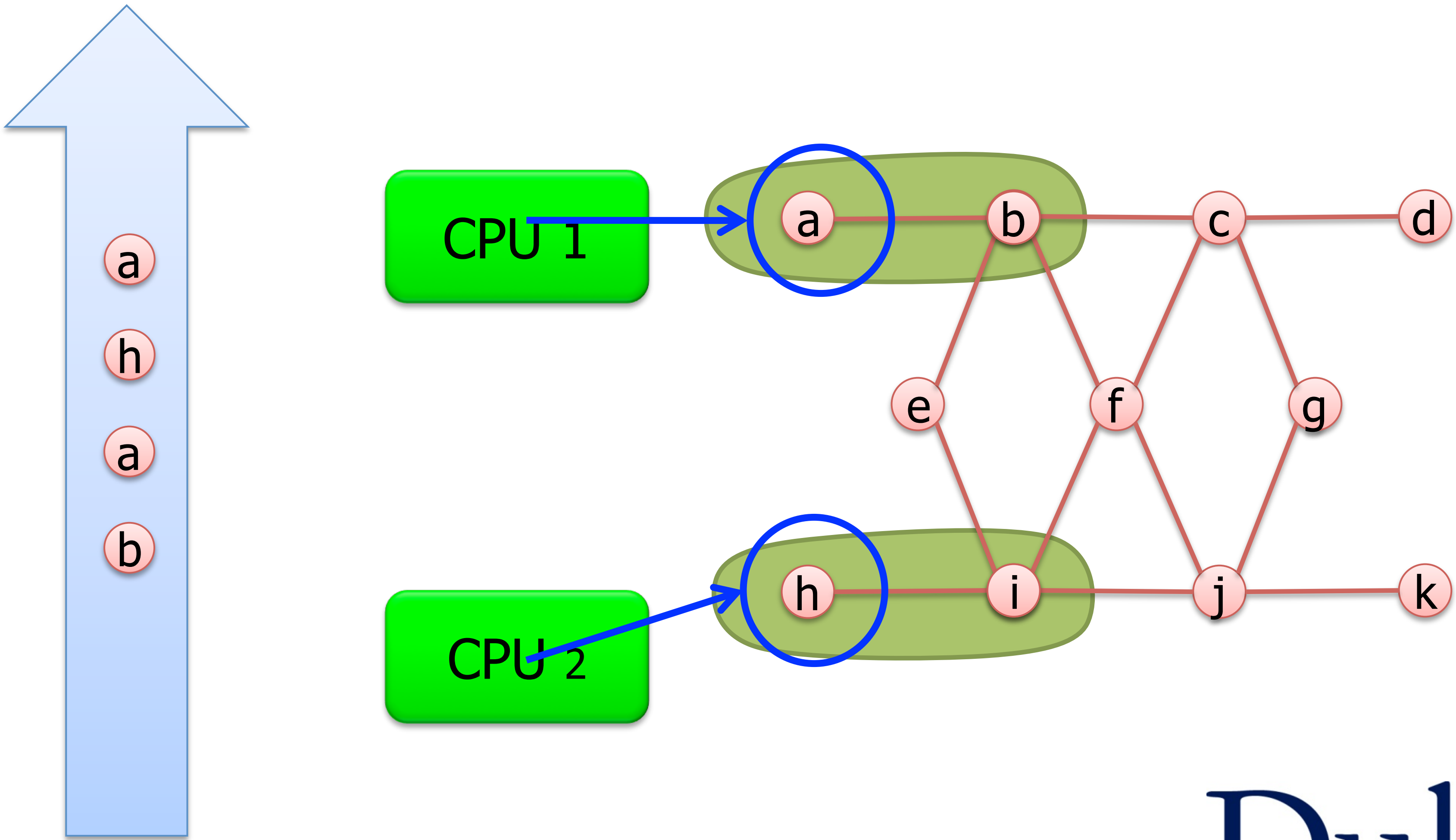
Synchronous Schedule:
Every vertex updated simultaneously

Round Robin Schedule:
Every vertex updated sequentially

Need for Dynamic Scheduling



Dynamic Schedule



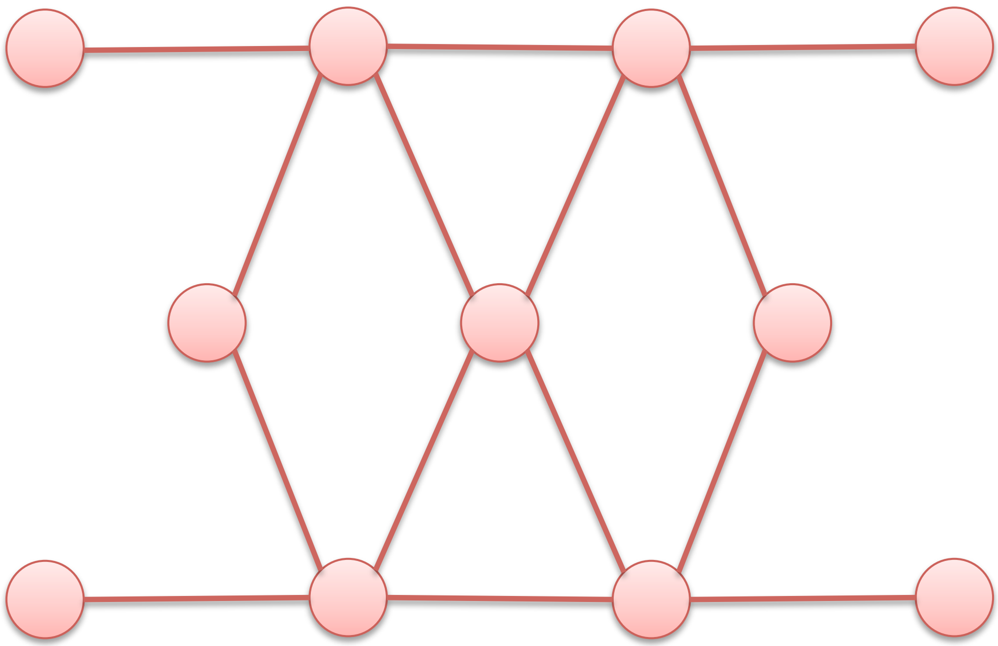
Duke
UNIVERSITY

Dynamic Schedule

Update Functions can insert new tasks into the schedule

FIFO Queue	Wildfire BP [Selvatici et al.]
Priority Queue	Residual BP [Elidan et al.]
Splash Schedule	Splash BP [Gonzalez et al.]

Global Information



What if we need global information?

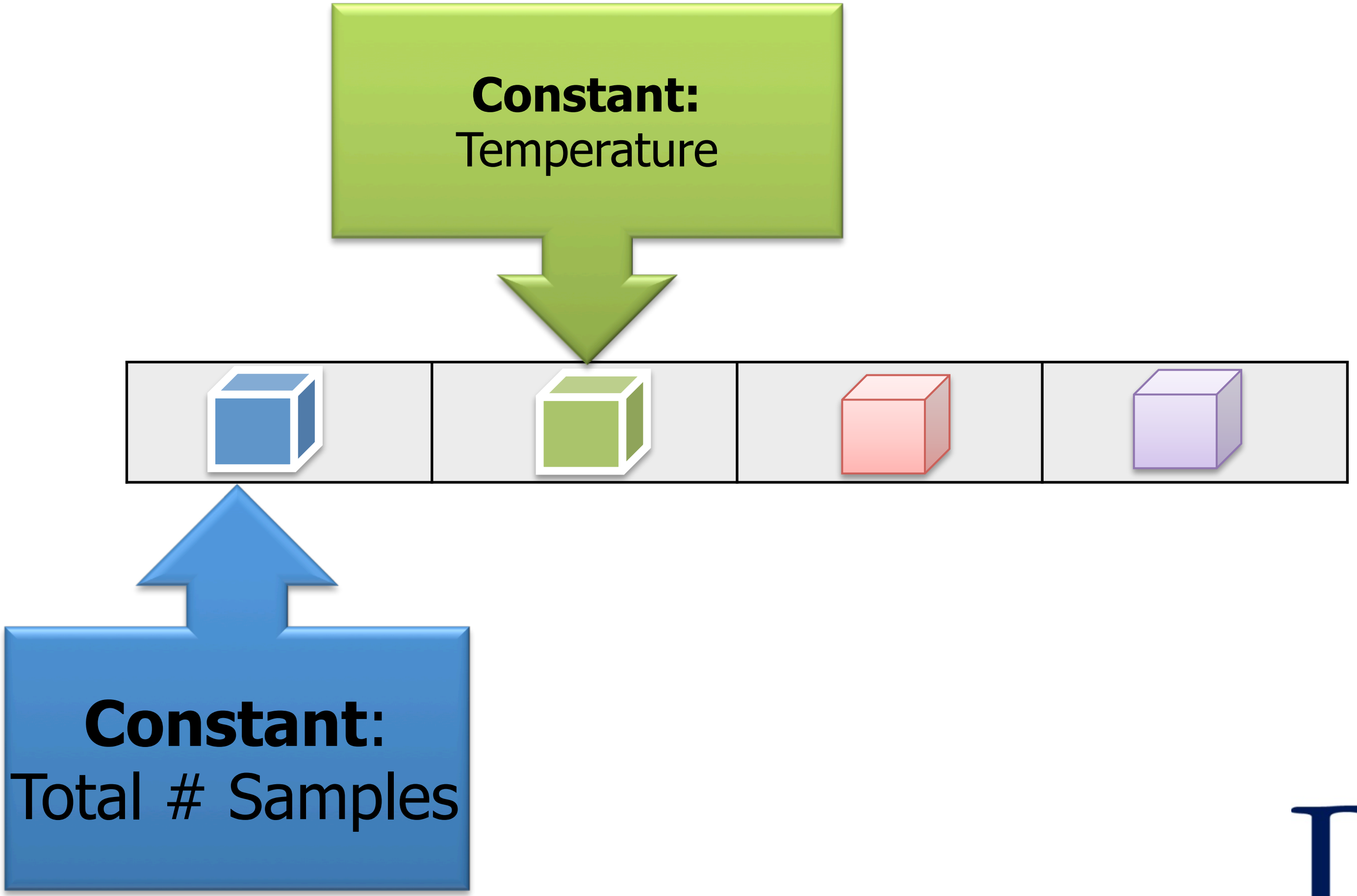
Algorithm Parameters?

Sufficient Statistics?

Sum of all the vertices?

Shared Data Table (SDT)

- Global constant parameters

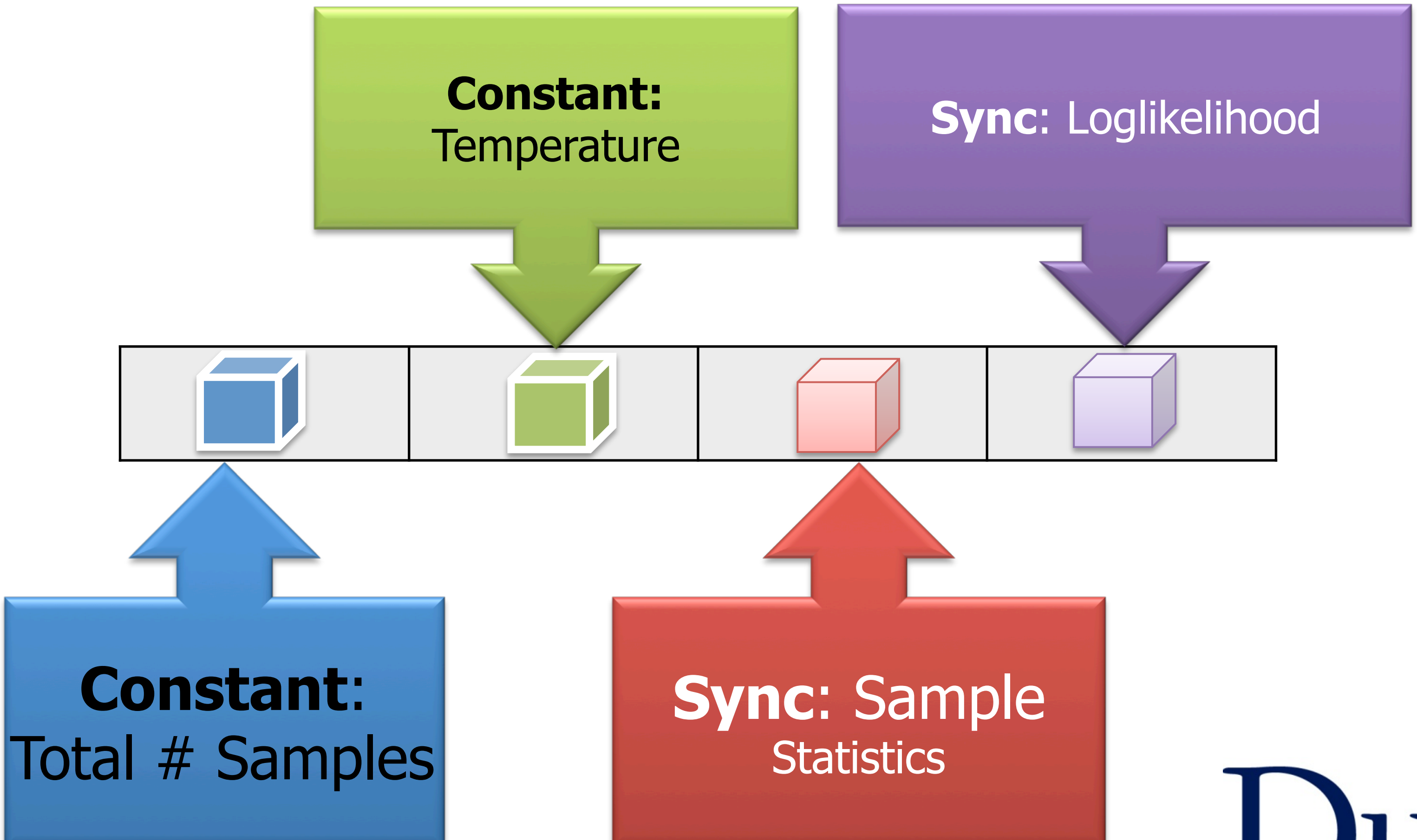


Sync Operation

- **Sync** is a fold/reduce operation over the graph
- **Accumulate** performs an aggregation over vertices
- **Apply** makes a final modification to the accumulated data
- **Example:** Compute the average of all the vertices

Shared Data Table (SDT)

- Global constant parameters
- Global computation (**Sync Operation**)



Duke UNIVERSITY

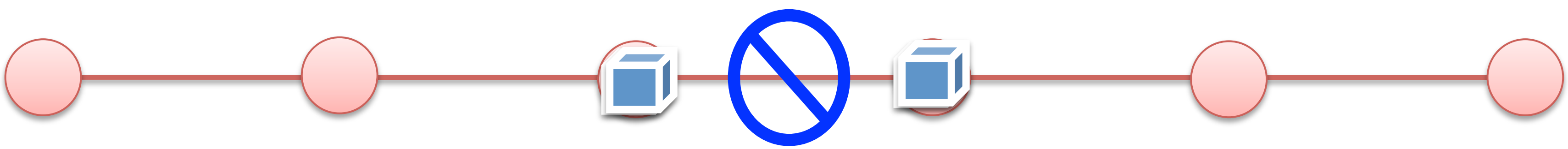
Safety

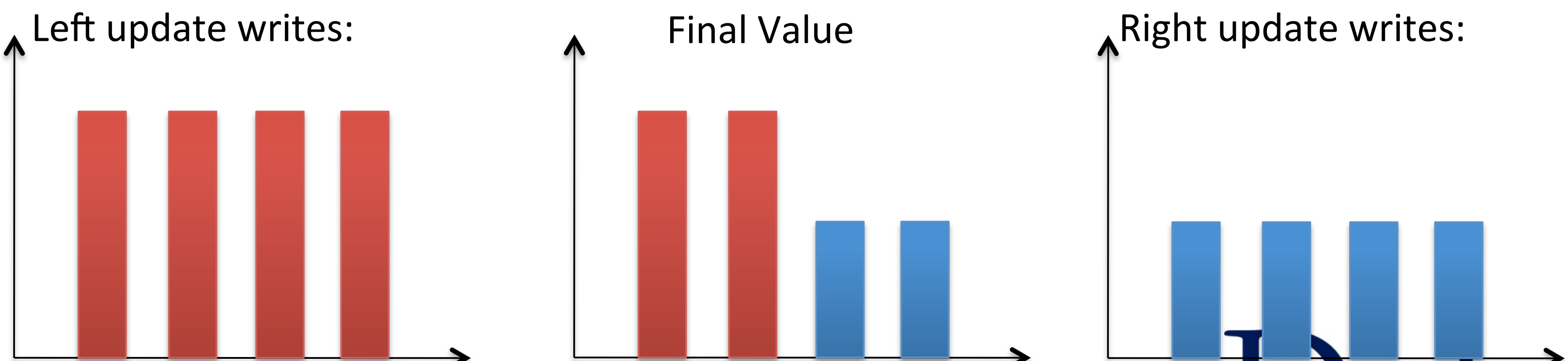
and

Consistency

Write-Write Race

Write-Write Race
If adjacent update functions write simultaneously



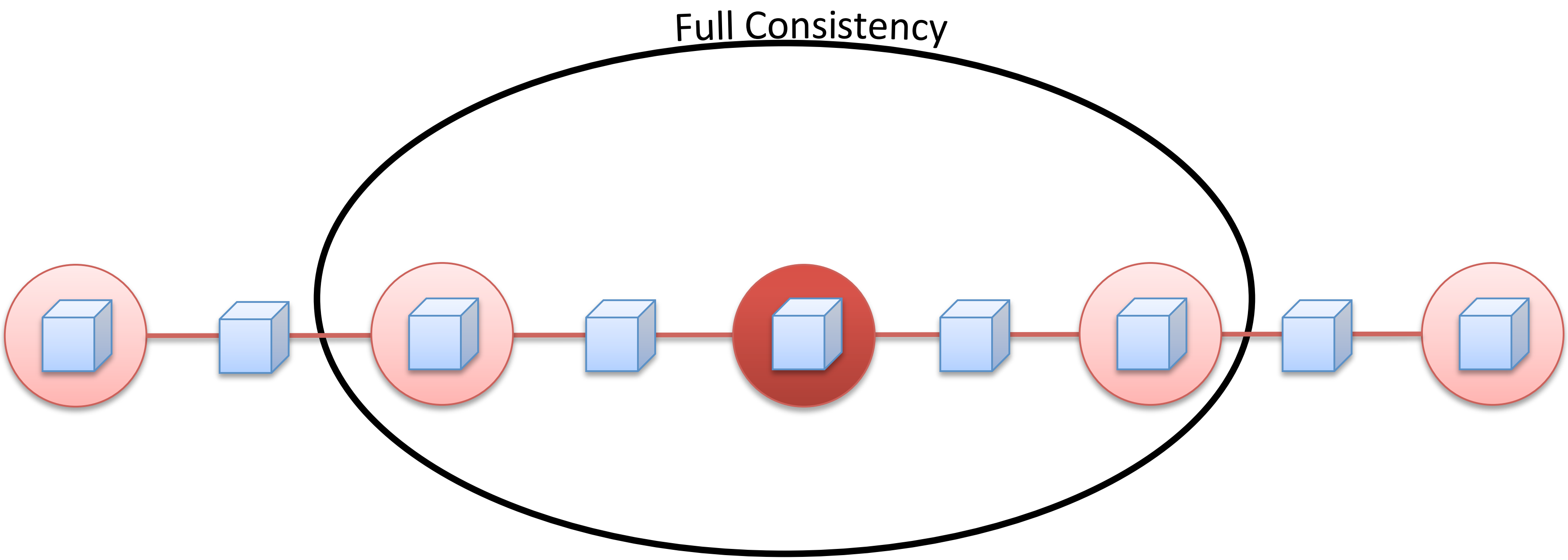


Race Conditions + Deadlocks

- Just one of the many possible races
- Race-free code **is extremely difficult** to write

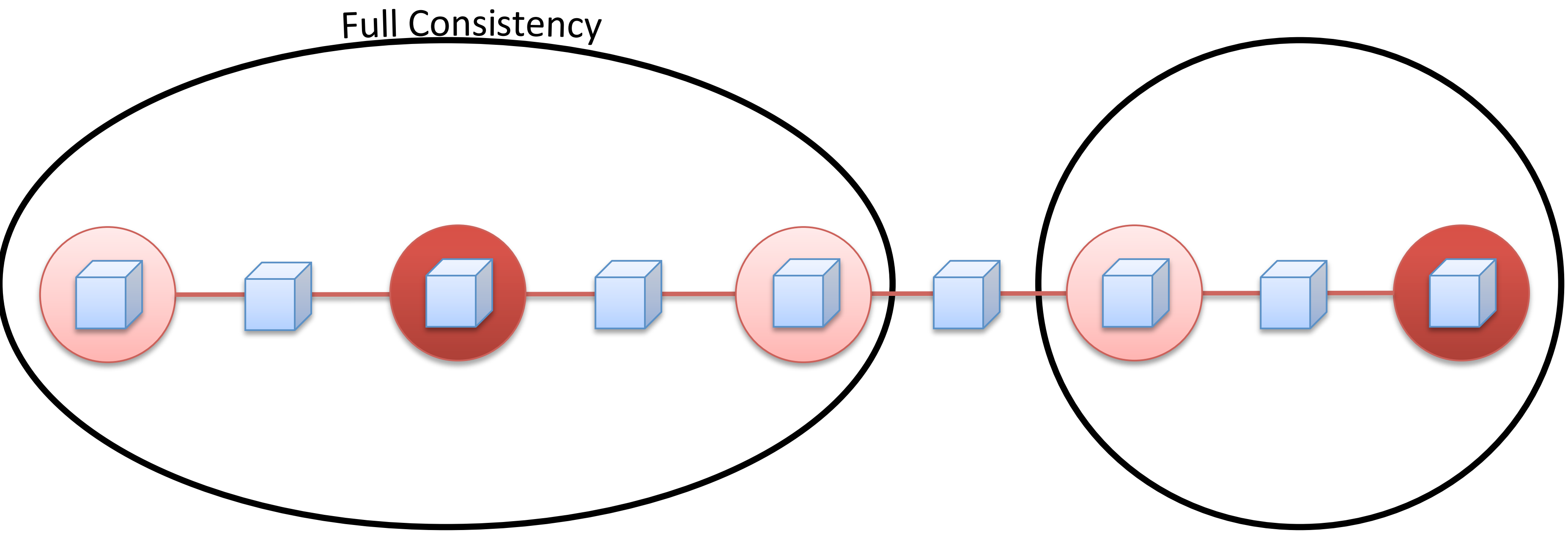
GraphLab design ensures
race-free operation

Scope Rules



Guaranteed safety for all update functions

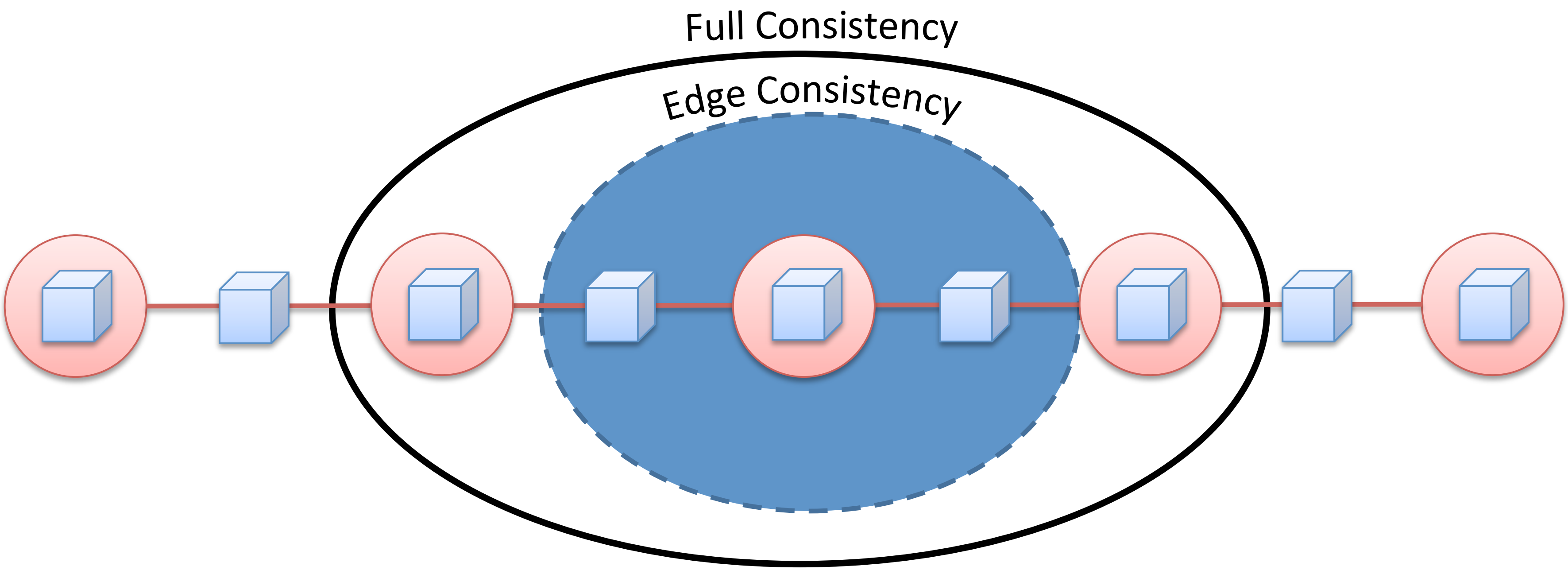
Full Consistency



Only allow update functions two vertices apart to be run in parallel
Reduced opportunities for parallelism

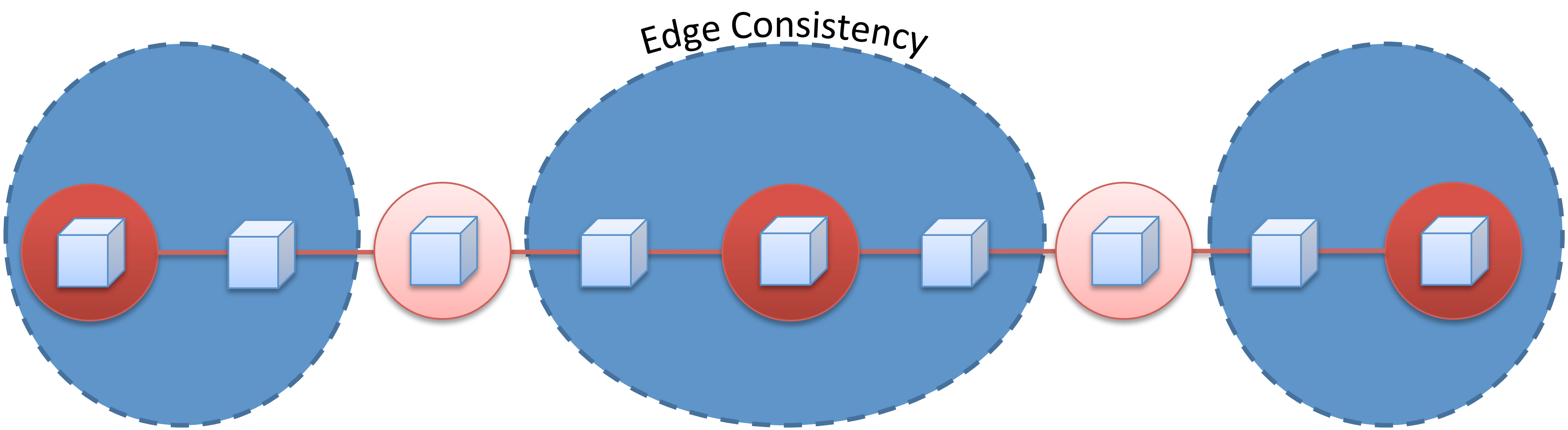
Obtaining More Parallelism

Not all update functions will modify the entire scope!

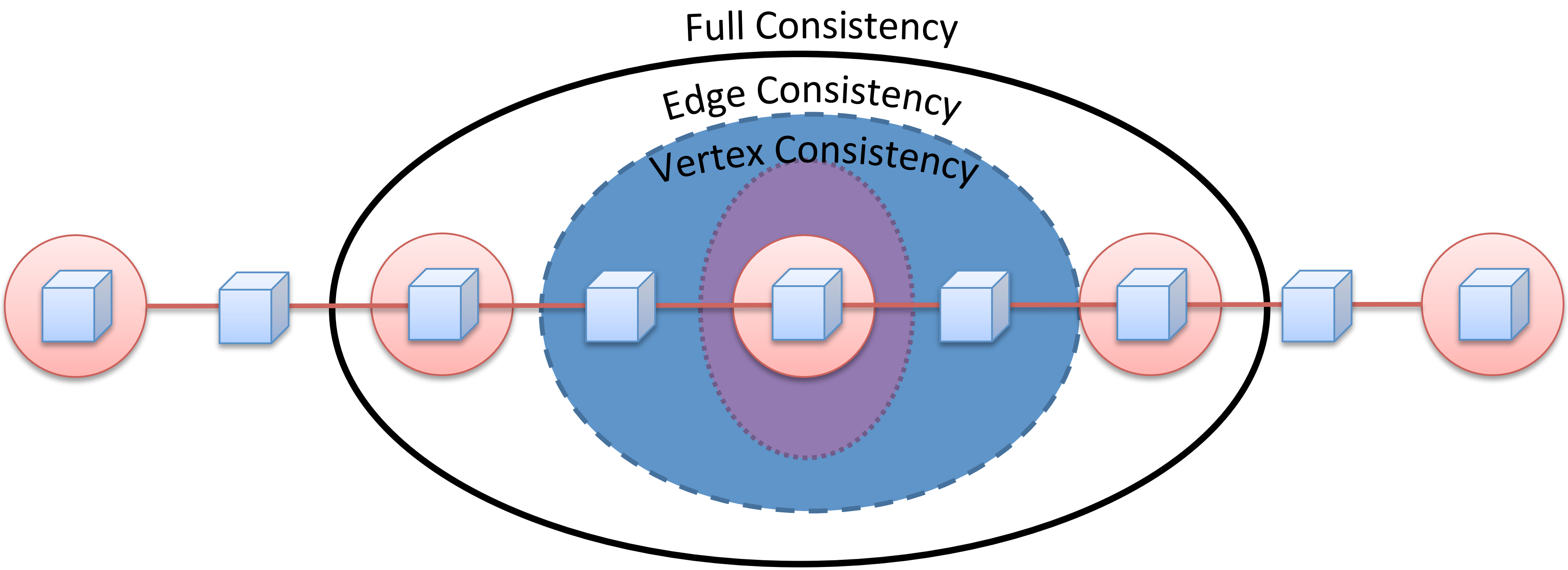


Belief Propagation: Only uses edge data
Gibbs Sampling: Only needs to read adjacent vertices

Edge Consistency

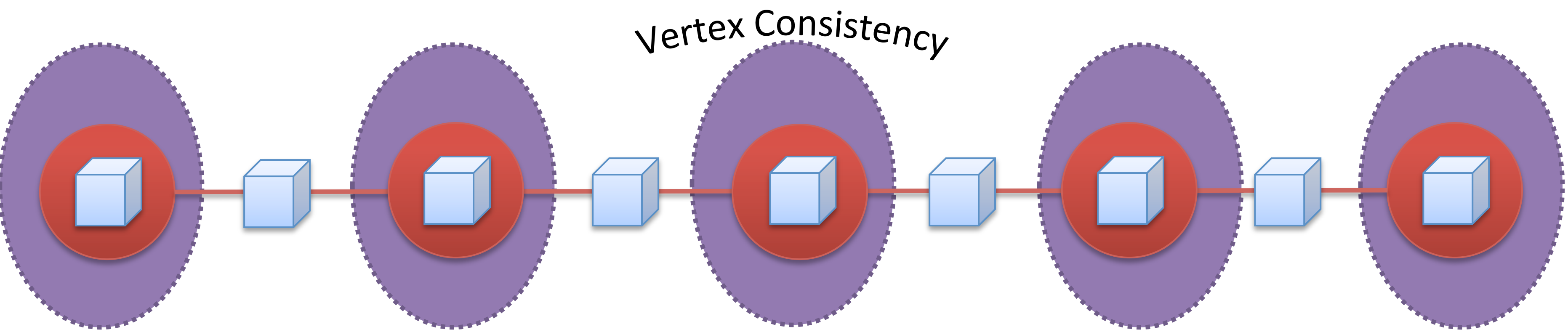


Obtaining More Parallelism



“Map” operations. Feature extraction on vertex data

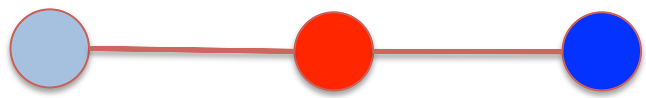
Vertex Consistency



Sequential Consistency

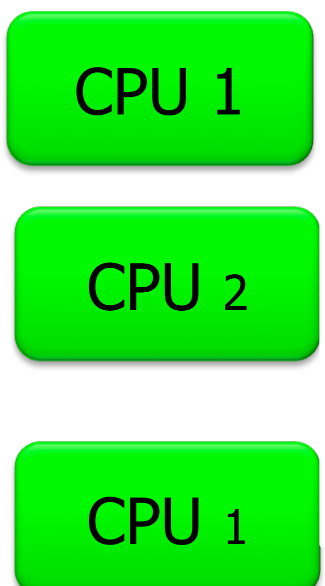
GraphLab guarantees **sequential consistency**

For **every parallel execution**, there exists a **sequential execution** of update functions which will produce the same result.



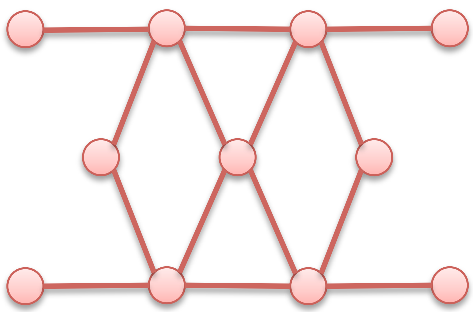
Parallel

Sequential

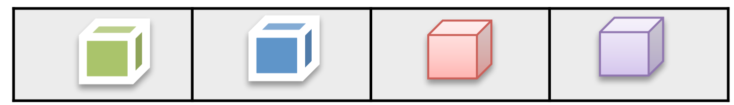


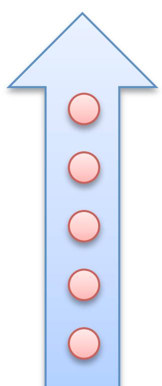
GraphLab

Data Graph

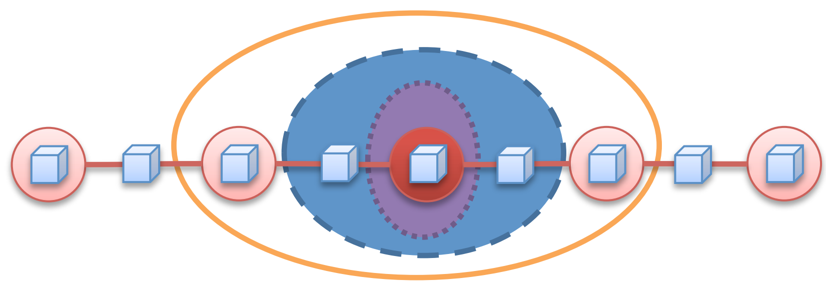


Shared Data Table





Scheduling



Update Functions and Scopes

Duke UNIVERSITY

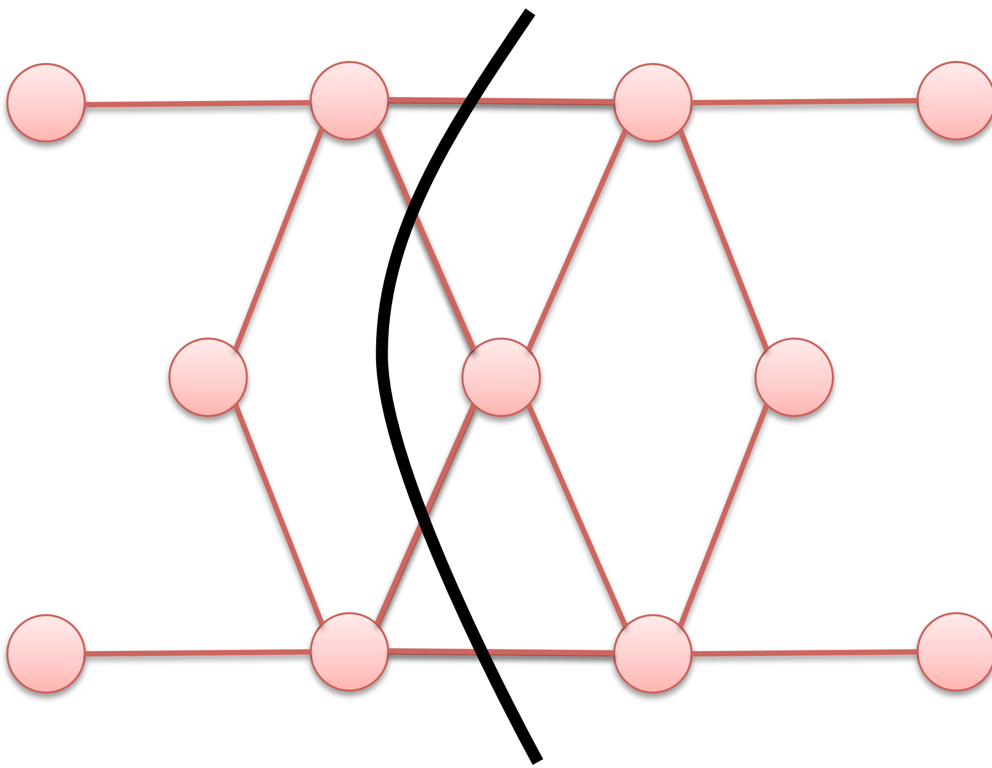
DISTRIBUTED GRAPHLAB

Distributing GraphLab

- NOT SHARED-NOTHING (unlike MapReduce / Pregel)
  - Need to have distributed shared memory
- No change to the update step
- Need to to distributed scheduling
- Need to ensure distributed consistency
- Need to ensure fault tolerance

Distributed Graph

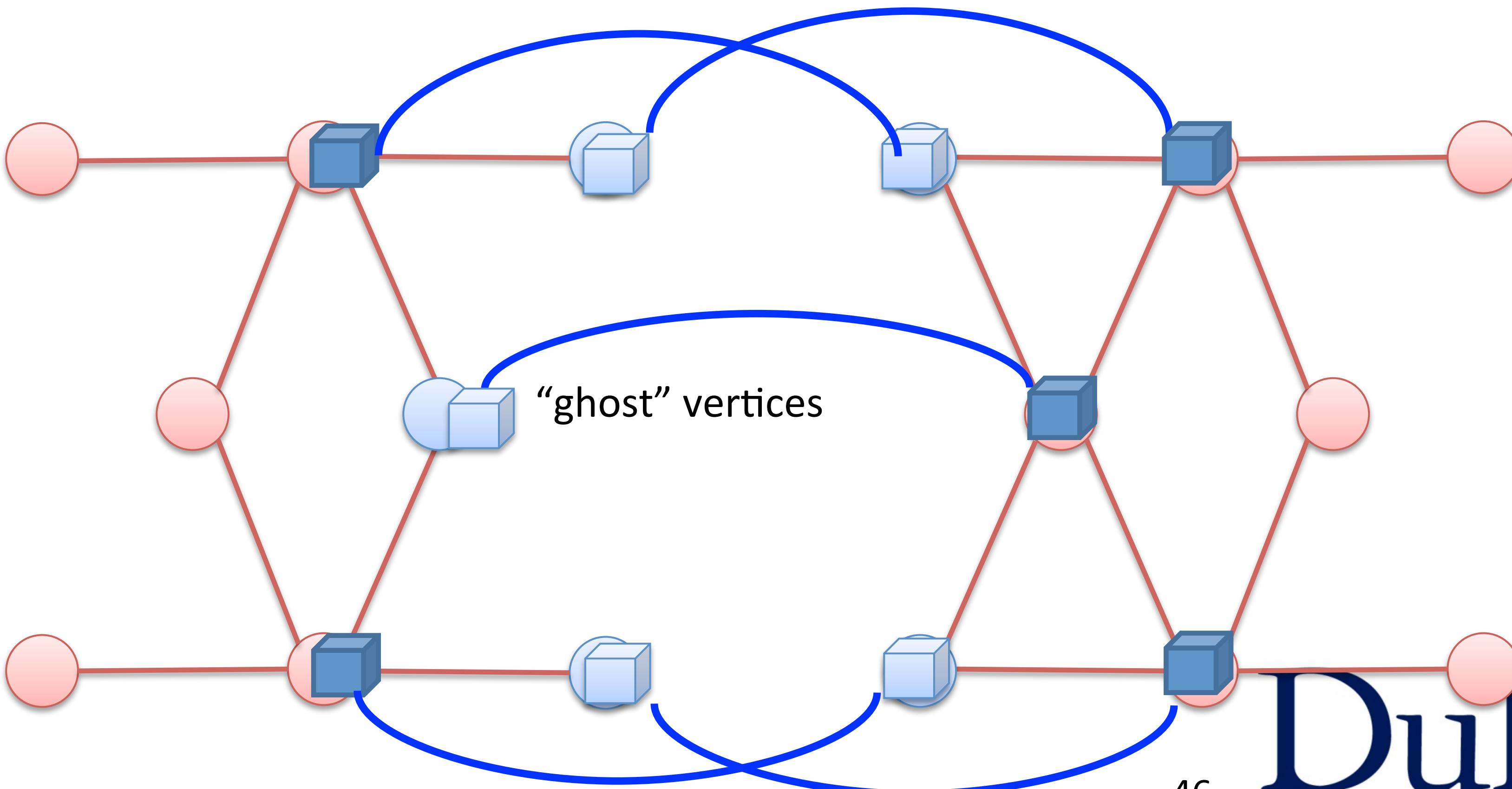
Partition the graph across multiple machines.



Duke UNIVERSITY

Distributed Graph

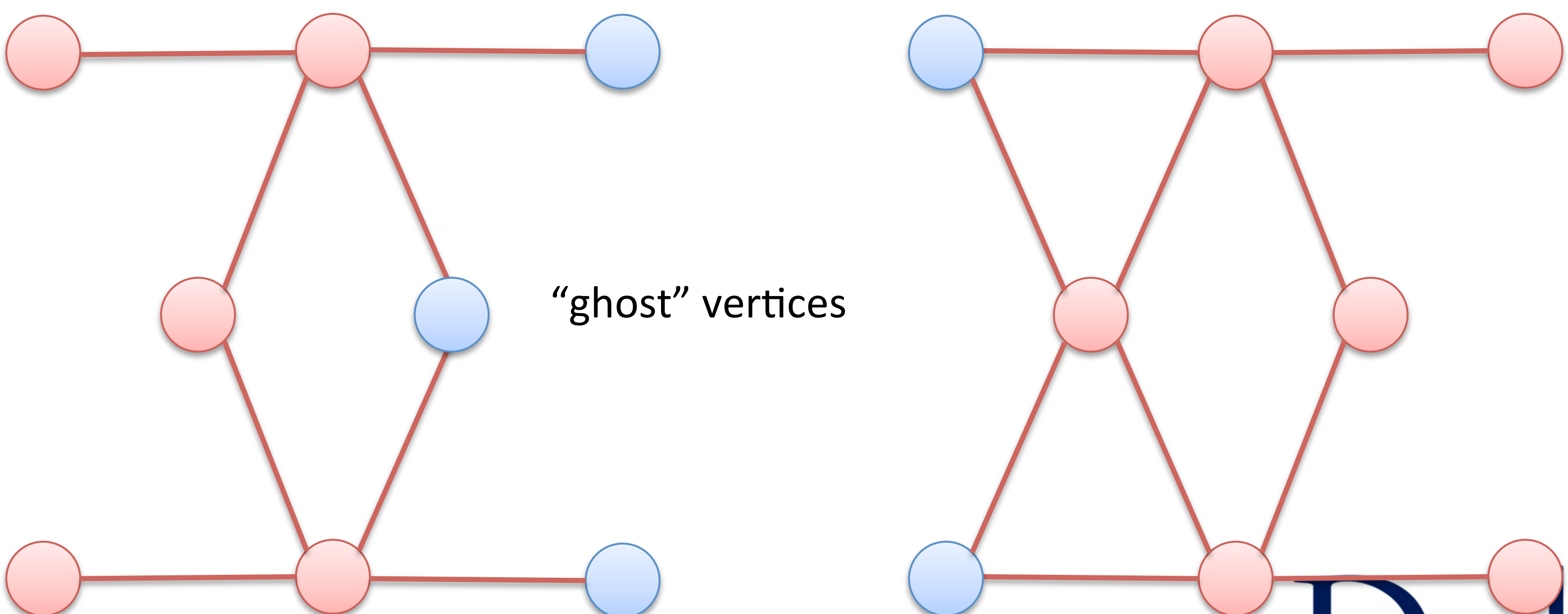
- Ghost vertices maintain adjacency structure and replicate remote data.



Duke UNIVERSITY

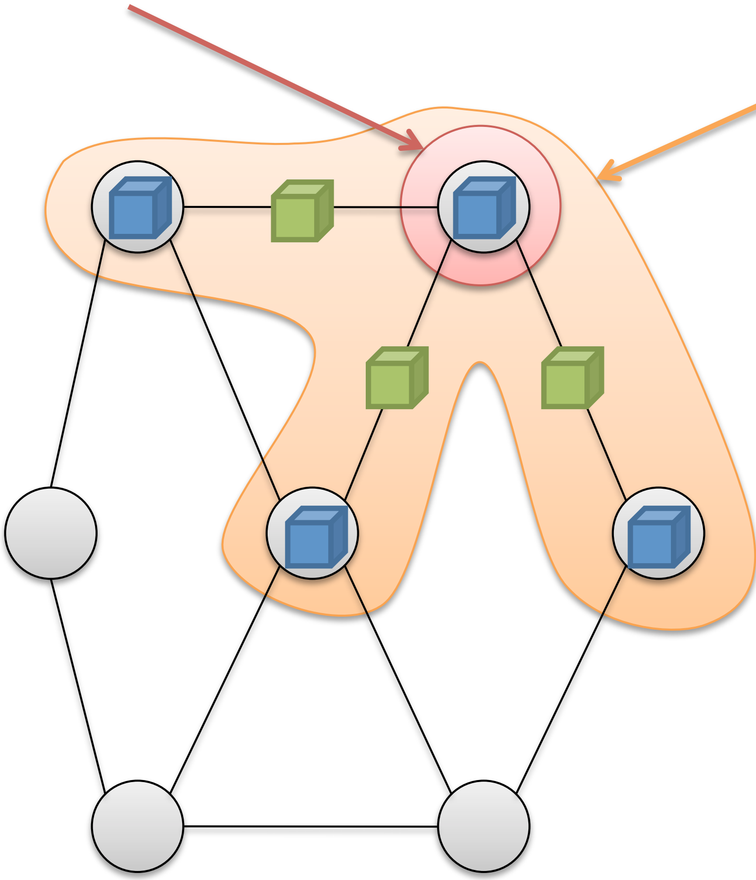
Distributed Graph

- Cut efficiently using HPC Graph partitioning tools (ParMetis / Scotch / …)



Update Functions

User-defined program: applied to a **vertex** and transforms data in **scope** of vertex



Pagerank(scope){
// Update the current vertex data

$\text{vertex.PageRank} = \alpha$

ForEach inPage:

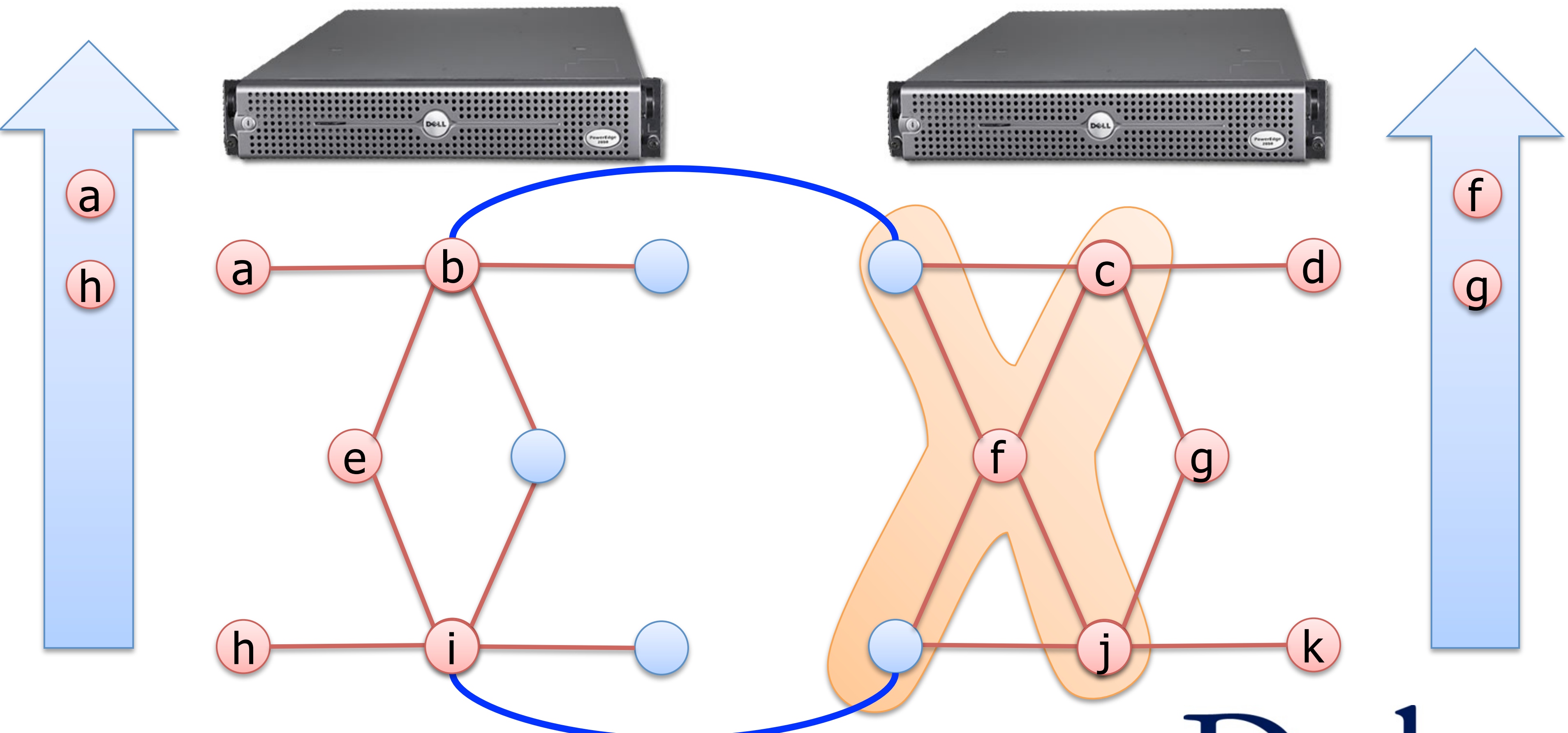
$\text{vertex.PageRank} \mathrel{+}= (1-\alpha) \times \text{inPage.PageRank}$

// Reschedule Neighbors if needed
if vertex.PageRank changes then
reschedule_all_neighbors;
}

Duke UNIVERSITY

Distributed Scheduling

Each machine maintains a schedule over the vertices it owns.



Distributed Consensus used to identify completion

Distributed Consistency

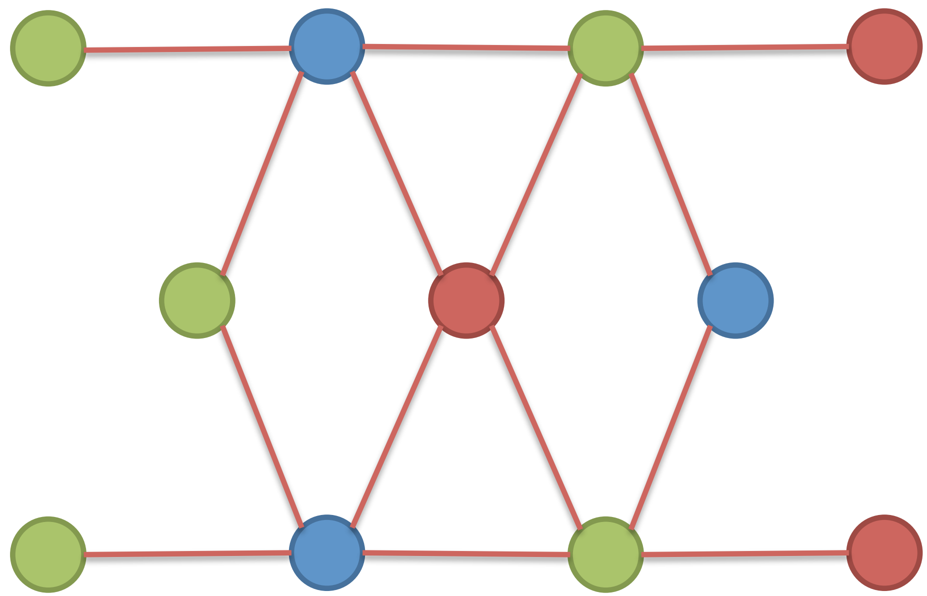
Solution 1

Graph Coloring

Solution 2

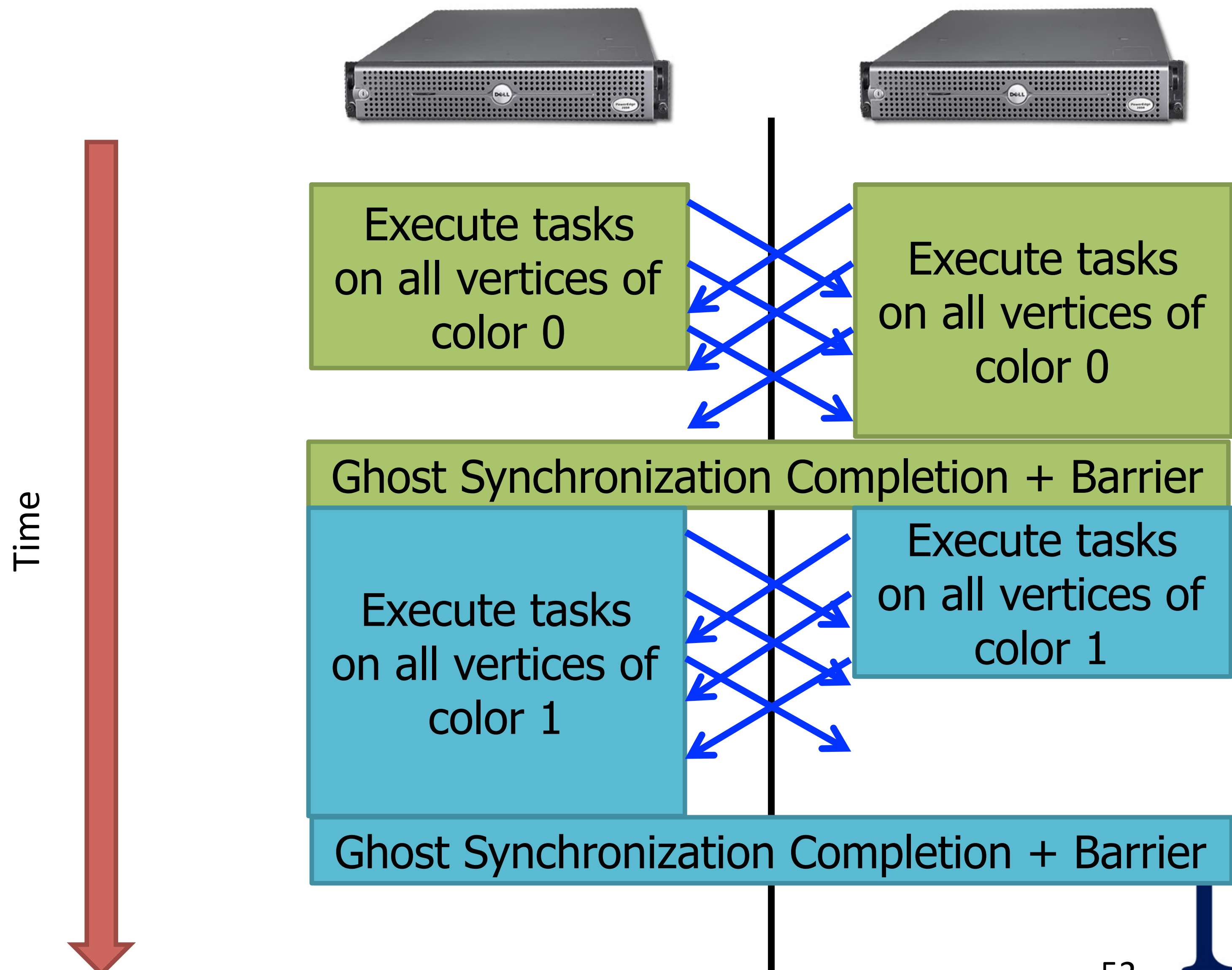
Distributed Locking

Edge Consistency via Graph Coloring



Vertices of the same color are all at least one vertex apart.
Therefore, All vertices of the same color can be run in parallel!

Chromatic Distributed Engine



Problems

- Require a **graph coloring** to be available.

- **Frequent Barriers** make it extremely inefficient for highly dynamic systems where only a small number of vertices are active in each round.

Distributed Consistency

Solution 1

Graph Coloring

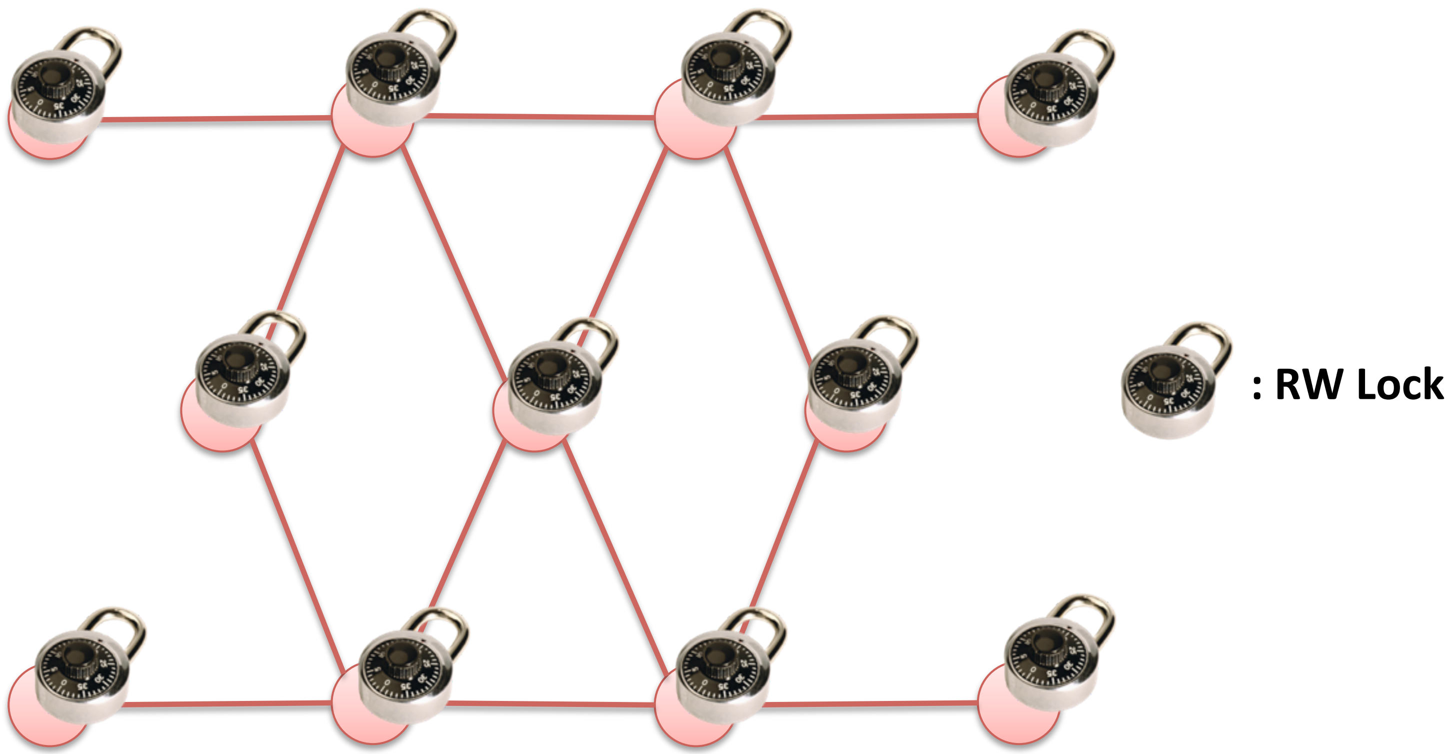
Solution 2

Distributed Locking

Duke
UNIVERSITY

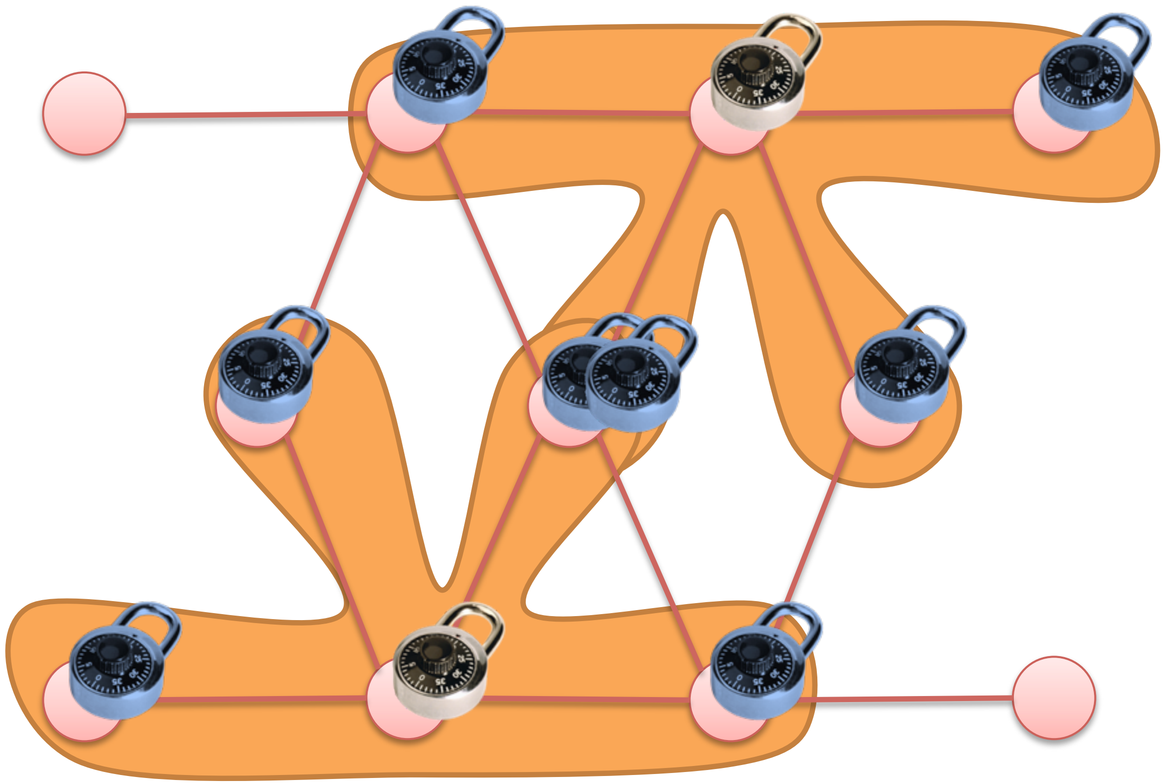
Distributed Locking

Edge Consistency can be guaranteed through locking.



Consistency Through Locking

Acquire write-lock on center vertex, read-lock on adjacent.

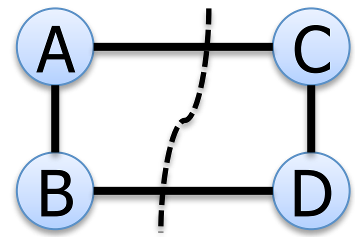


Duke UNIVERSITY

Consistency Through Locking

Multicore Setting

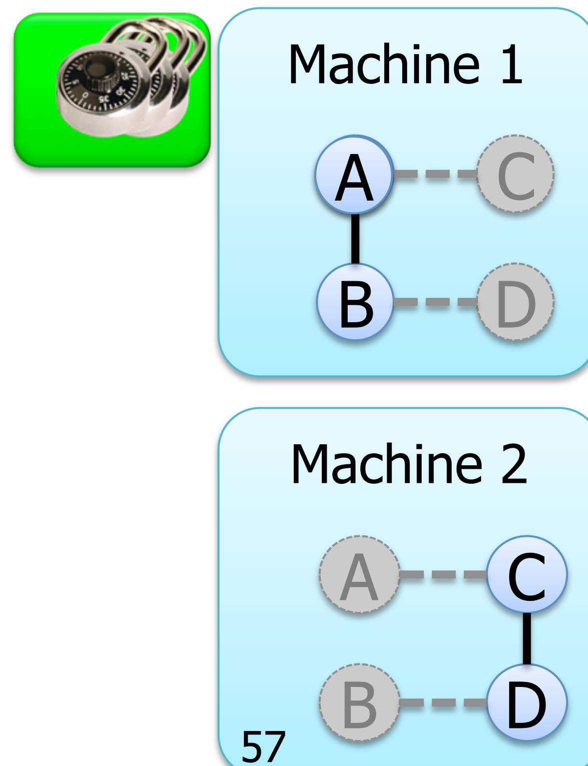
- PThread RW-Locks



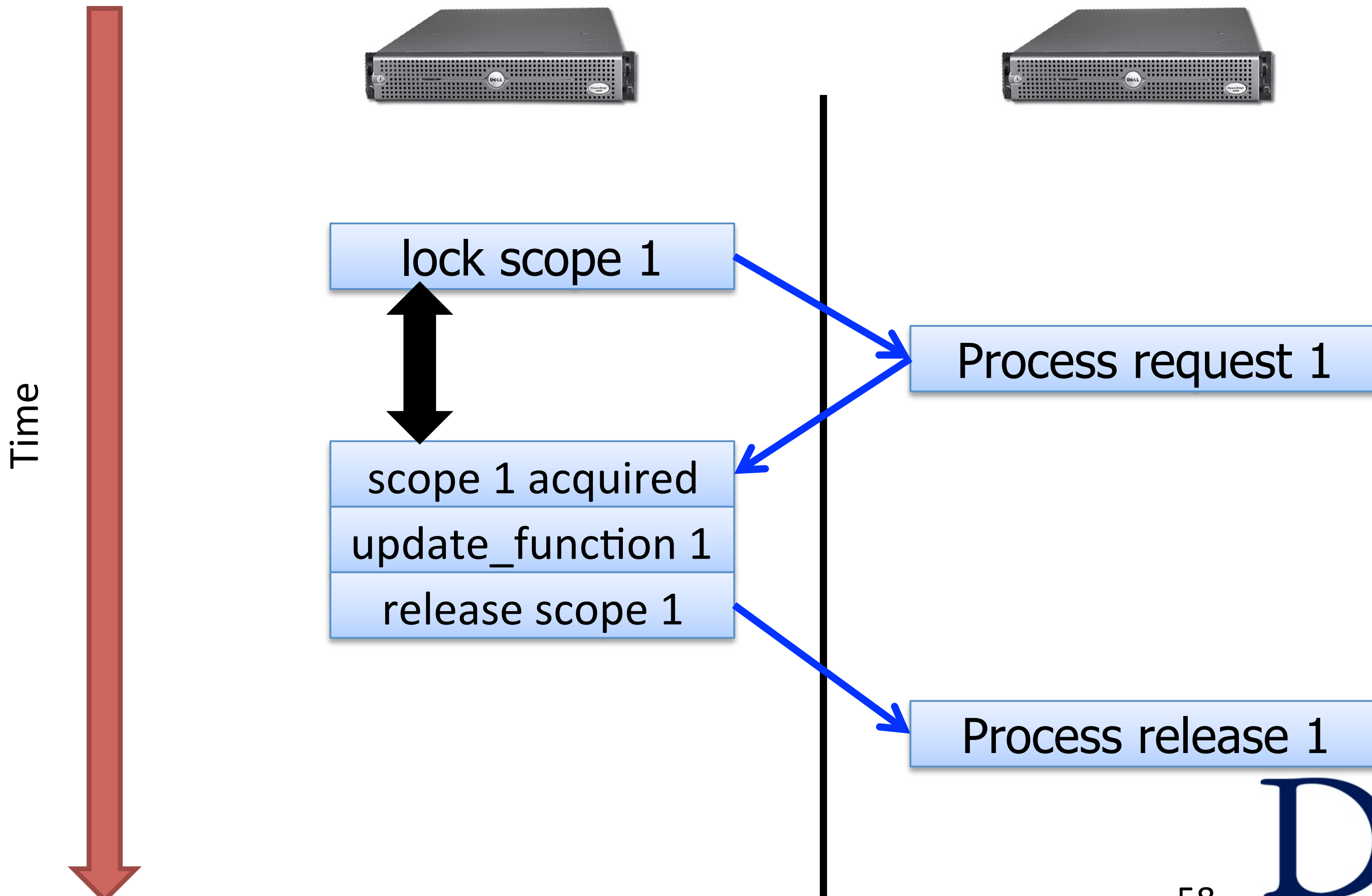
Distributed Setting

- Distributed Locks

- Challenges
  - Latency
- Solution
  - Pipelining

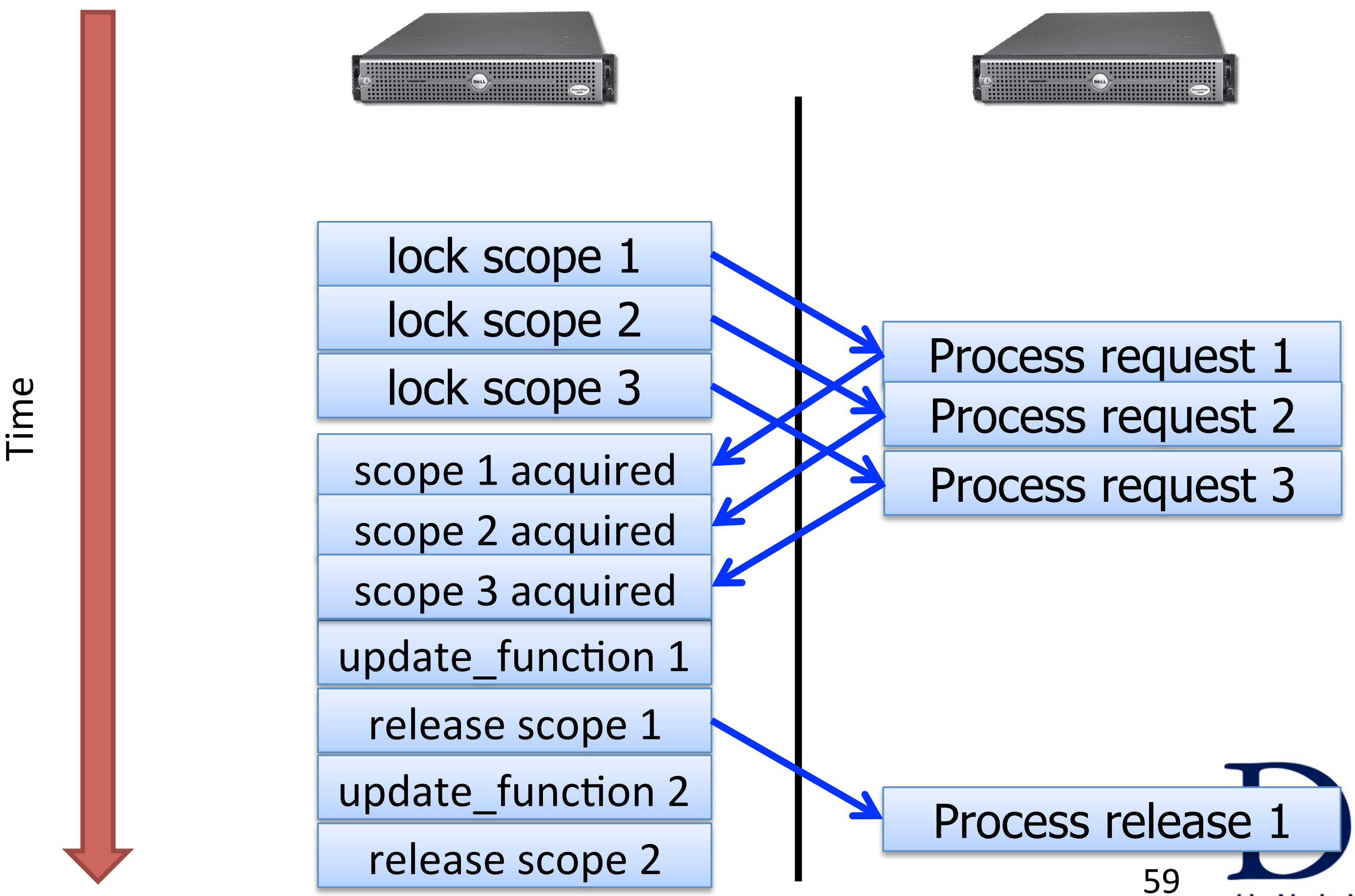


No Pipelining



Pipelining / Latency Hiding

Hide latency using pipelining



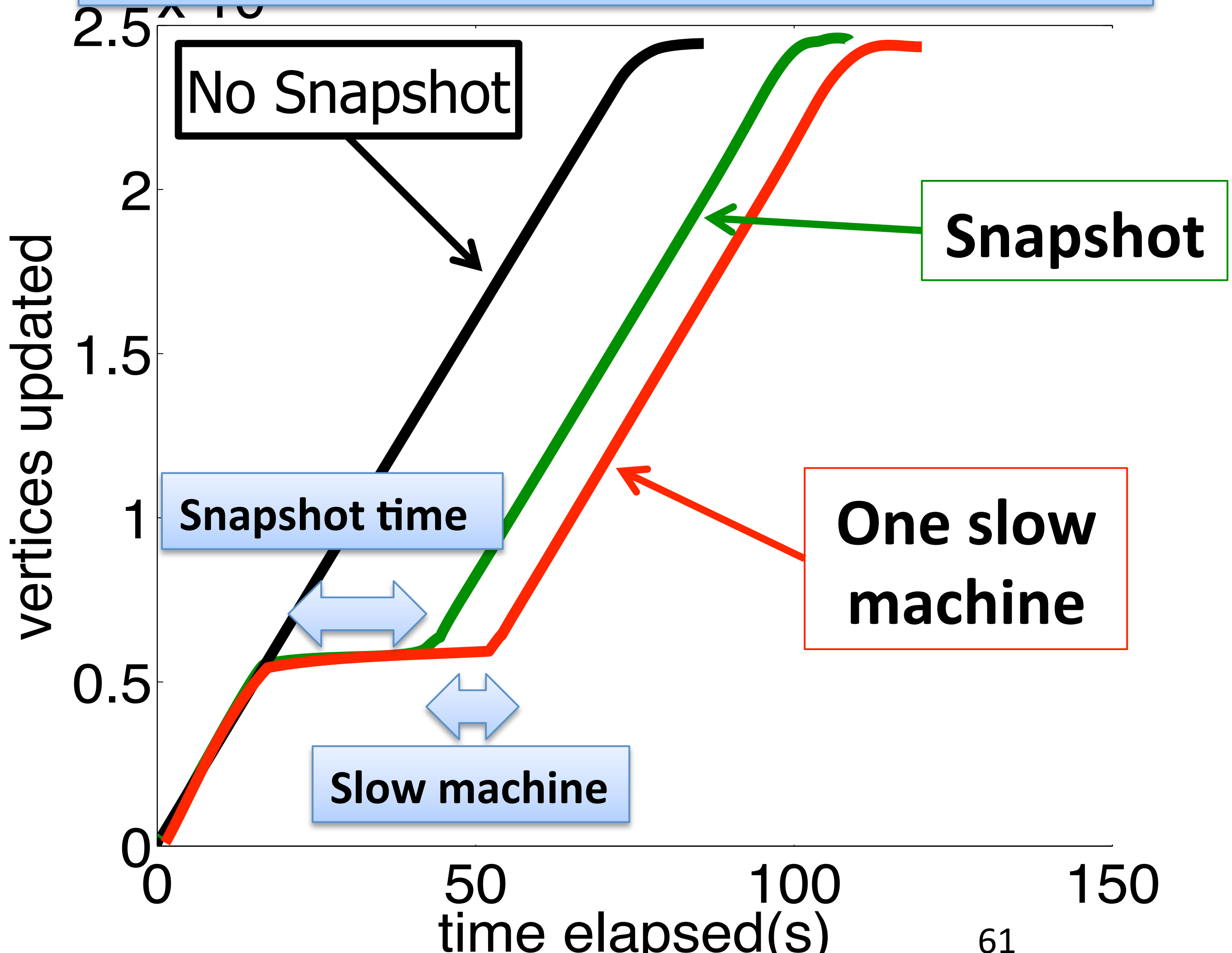
Checkpoints for Fault Tolerance

1: Stop the world

2: Write state to disk

Snapshot Performance

Because we have to stop the world,
One slow machine slows everything down!



Better Checkpointing

- Based on [Chandy, Lamport '85]
- Edge consistent update function

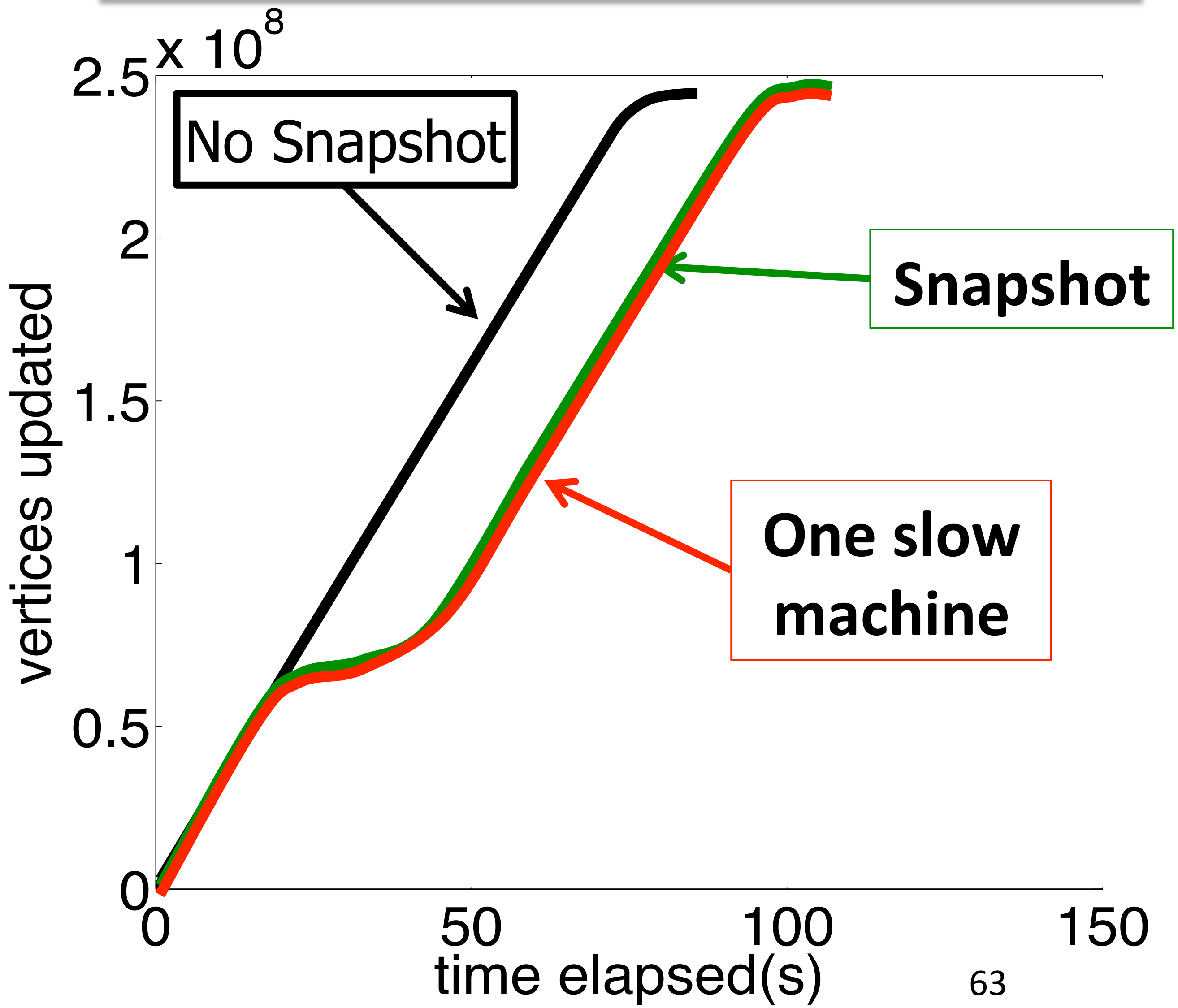
Algorithm 5: Snapshot Update on vertex v

if v *was already snapshotted* **then**
  Quit
Save D_v `// Save current vertex`
foreach $u \in \mathbf{N}[v]$ **do** `// Loop over neighbors`
  if u *was not snapshotted* **then**
    Save data on edge $D_{u \leftrightarrow v}$
    `Schedule` u for a Snapshot Update
Mark v as snapshotted

Duke
UNIVERSITY

Async. Snapshot Performance

No penalty incurred by the slow machine!



Summary

- Asynchronous serial graph algorithms can converge faster than synchronous parallel graph algorithms

- GraphLab provides high level abstractions for writing asynchronous graph algorithms
  - Takes care of consistency and scheduling

- Distributed GraphLab
  - Graph processing using color-steps
  - Consistency ensured via pipelined distributed locking
  - Fault tolerance via fine grained checkpointing