Cornell University

Asynchronous Large-Scale Graph Processing Made Easy

Guozhang Wang, Wenlei Xie,

Al Demers, Johannes Gehrke

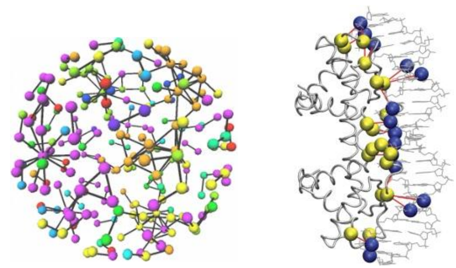
Cornell University

Graphs are ***ubiquitous***..

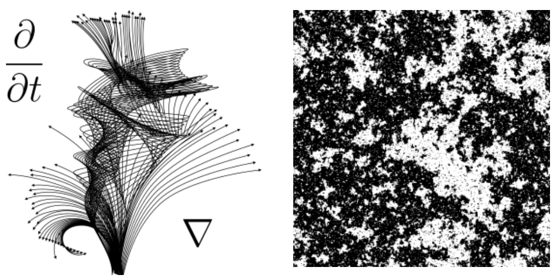
Social Networks

Web

Retail Advertising



DNA Analysis



Physical Simulations

Computer Vision

- Capture complex ***dependencies*** and ***interactions***

- Become ***essential*** in knowledge discovery and scientific studies

Existing Graph Processing Frameworks

GraphLab

GOLDENORB

Pregel

Naiad

PROJECT PEGASUS

APACHE GIRAPH

APACHE GIRAPH

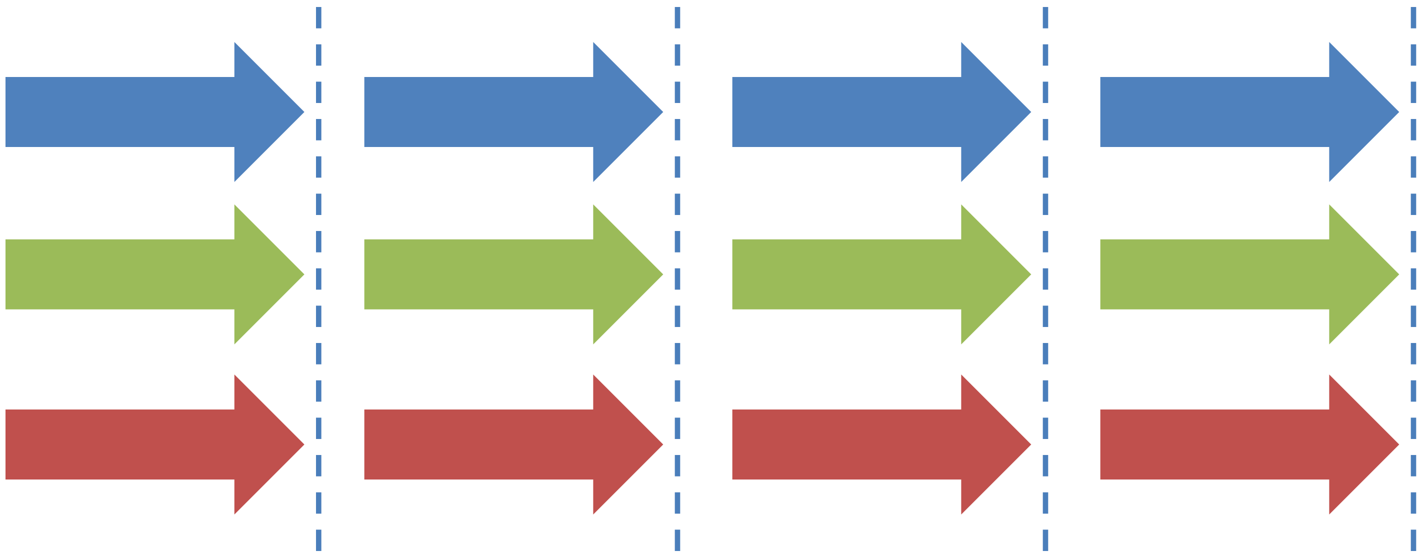
Galois

PrIter

Existing Graph Processing Frameworks

- Either follow BSP to compute ***synchronously***
  - Data is updated simultaneously and iteratively
  - Easy to program

Existing Graph Processing Frameworks



- Either follow BSP to compute ***synchronously***
  - Data is updated simultaneously and iteratively
  - Easy to program

Existing Graph Processing Frameworks

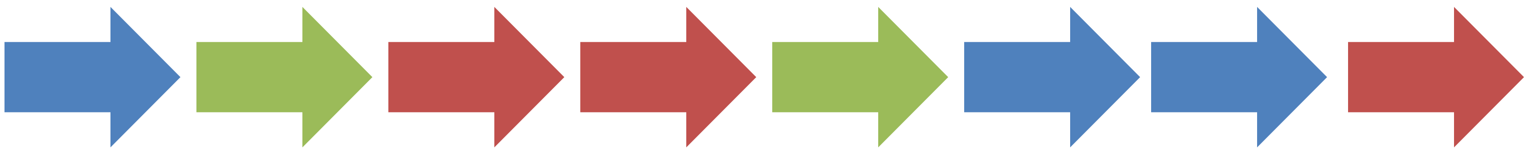
GraphLab

Naiad

Galois

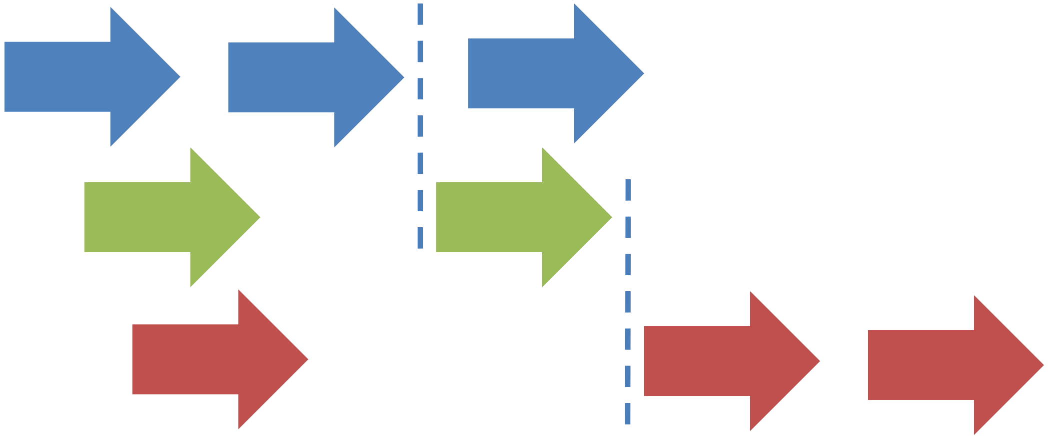
- Or compute ***asynchronously***
  - Data updates are (carefully) ordered
  - Data is updated using whatever available dependent state
  - Fast convergence

Existing Graph Processing Frameworks



- Or compute ***asynchronously***
  - Data updates are (carefully) ordered
  - Data is updated using latest available dependent state
  - Fast convergence

Existing Graph Processing Frameworks



- Or compute ***asynchronously***
  - Data updates are (carefully) ordered
  - Data is updated using latest available dependent state
  - Fast convergence

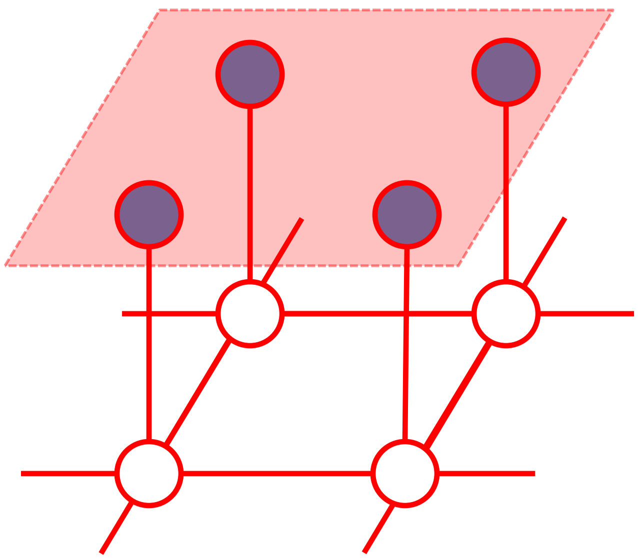
Research Goal:

A new graph computation framework that allows:

- *Sync. implementation for easy programming*
- *Async. execution for better performance*
    - ***Without** reimplementing everything*

Running Example: Belief Propagation

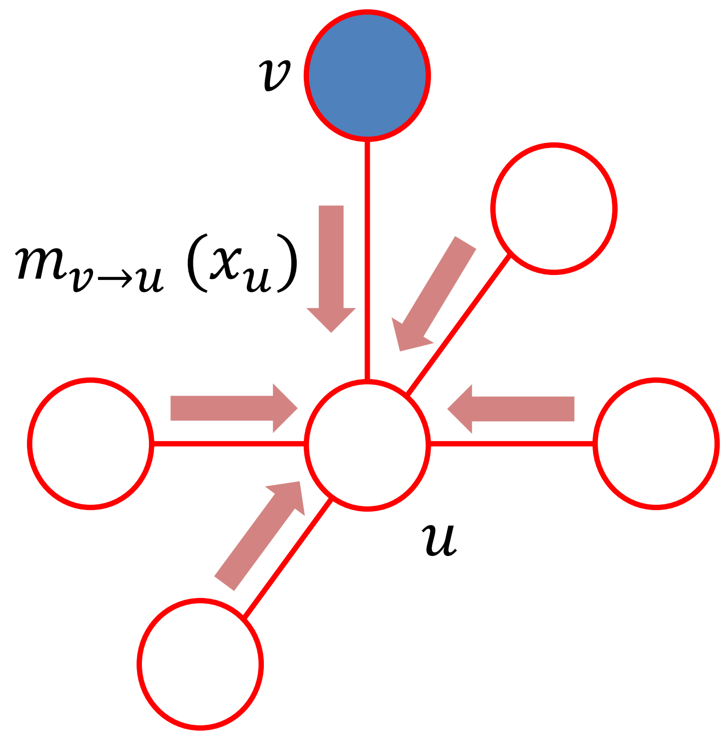
- Core procedure for many inference tasks in graphical models
  - Example: MRF for Image Restoration



Running Example: Belief Propagation

- Based on message passing to update local belief of each vertex:

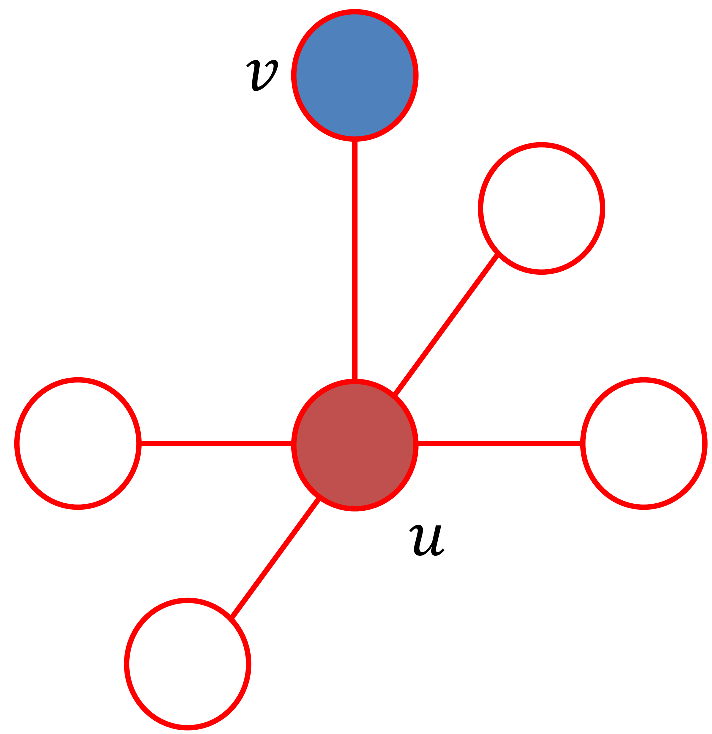
$$b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \to u}(x_u) \quad (1)$$



Running Example: Belief Propagation

- Based on message passing to update local belief of each vertex:

$$b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \to u}(x_u) \quad (1)$$

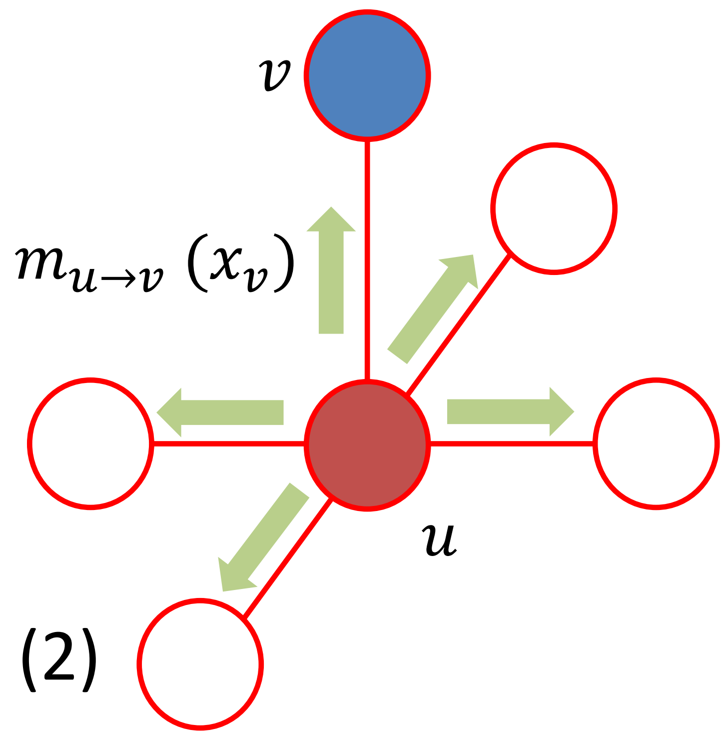


Running Example: Belief Propagation

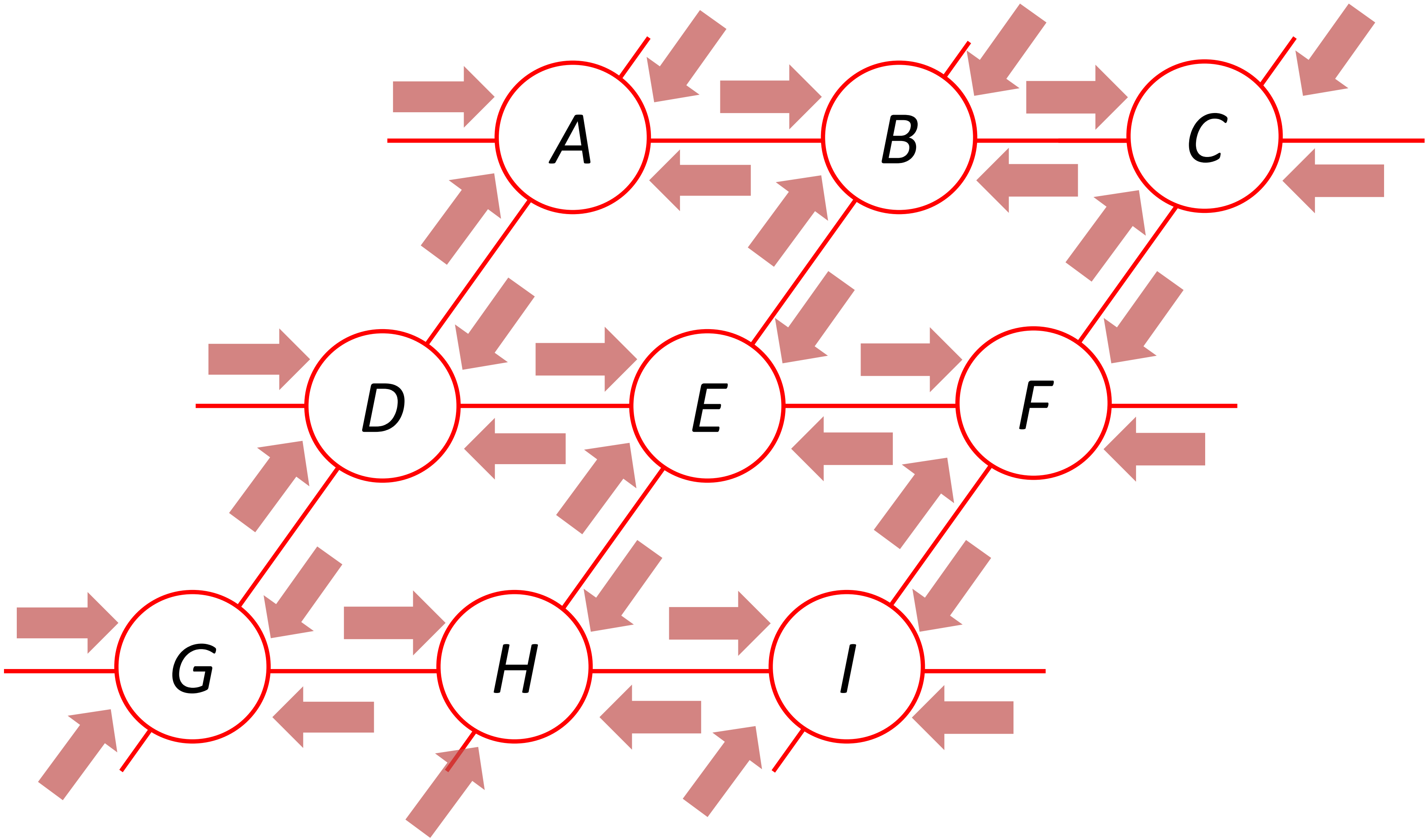
- Based on message passing to update local belief of each vertex:

$$b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \to u}(x_u) \quad (1)$$

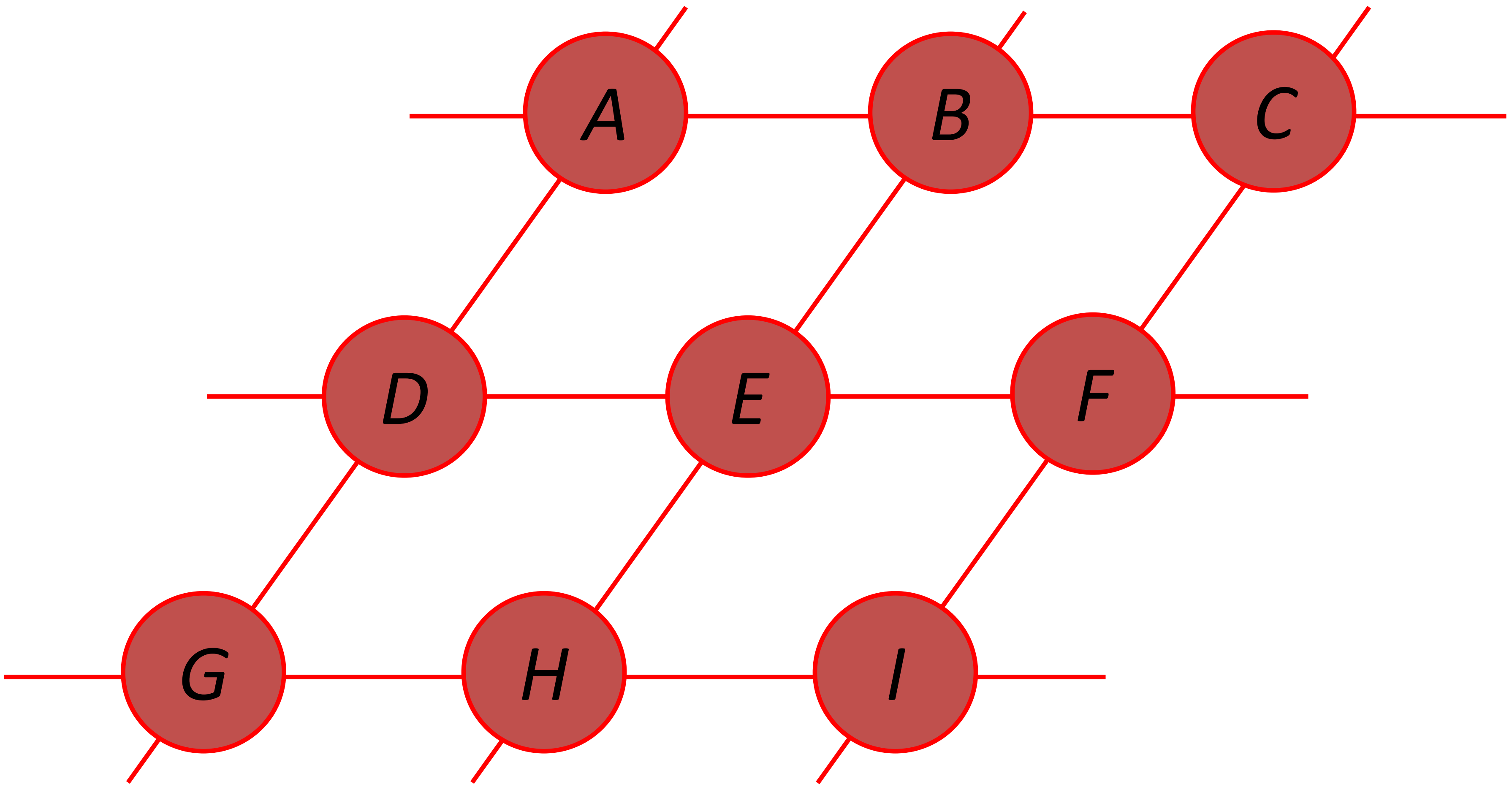
$$m_{u \to v}(x_v) \propto \sum_{x_u \in \Omega} \phi_{u,v}(x_u, x_v) \cdot \frac{b_u(x_u)}{m_{v \to u}(x_u)} \quad (2)$$



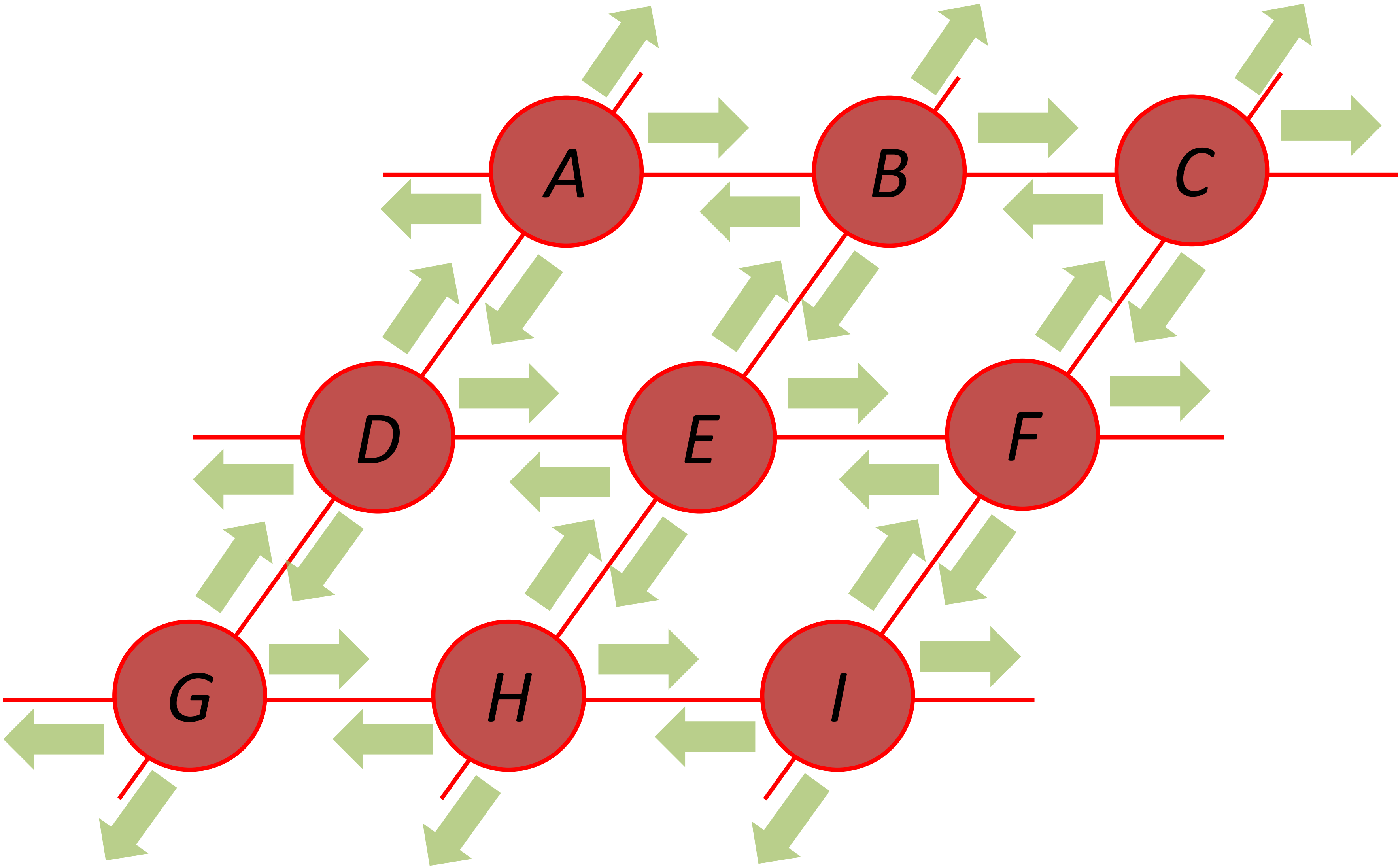
Original BP Implementation



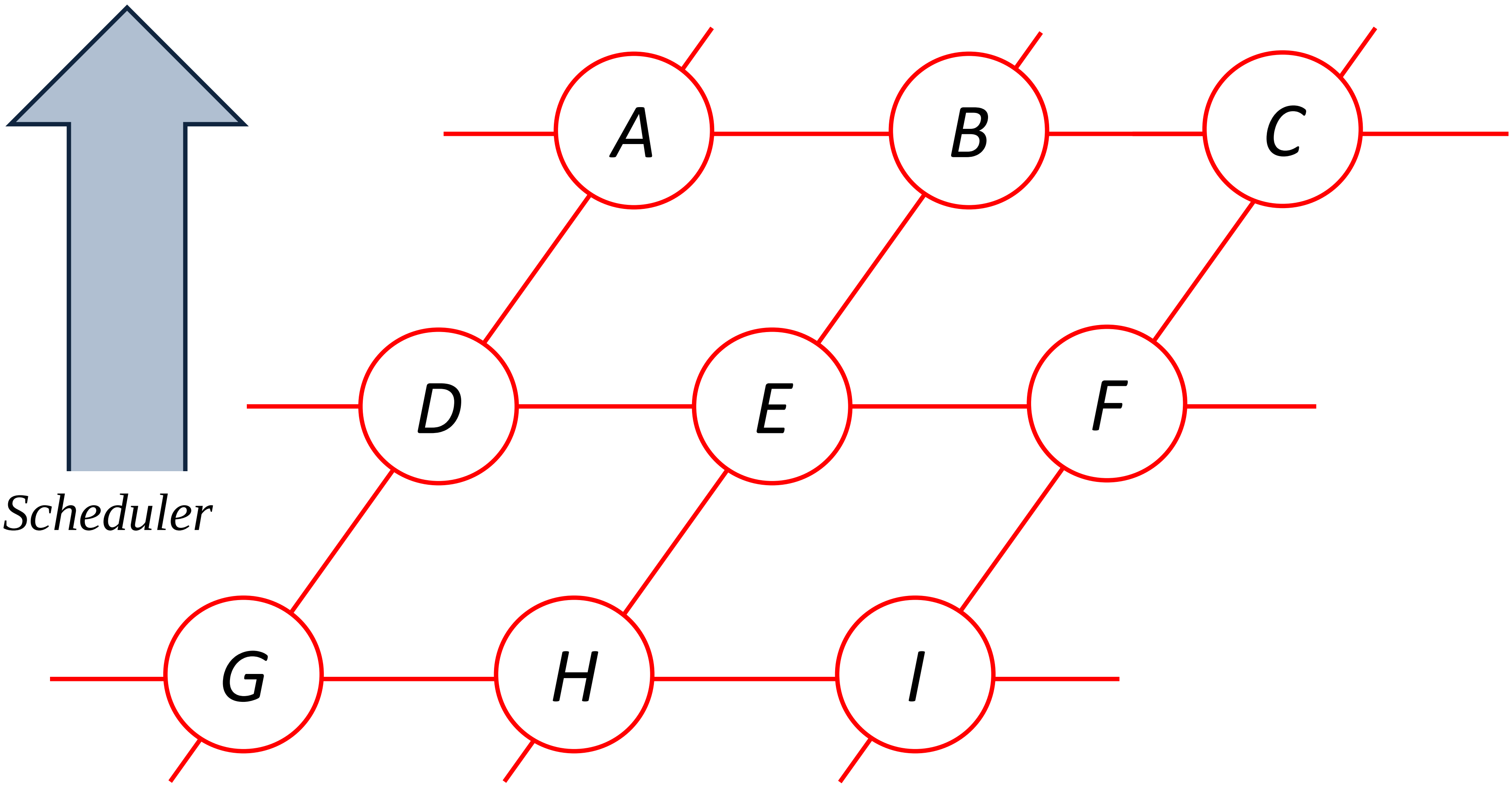
Original BP Implementation



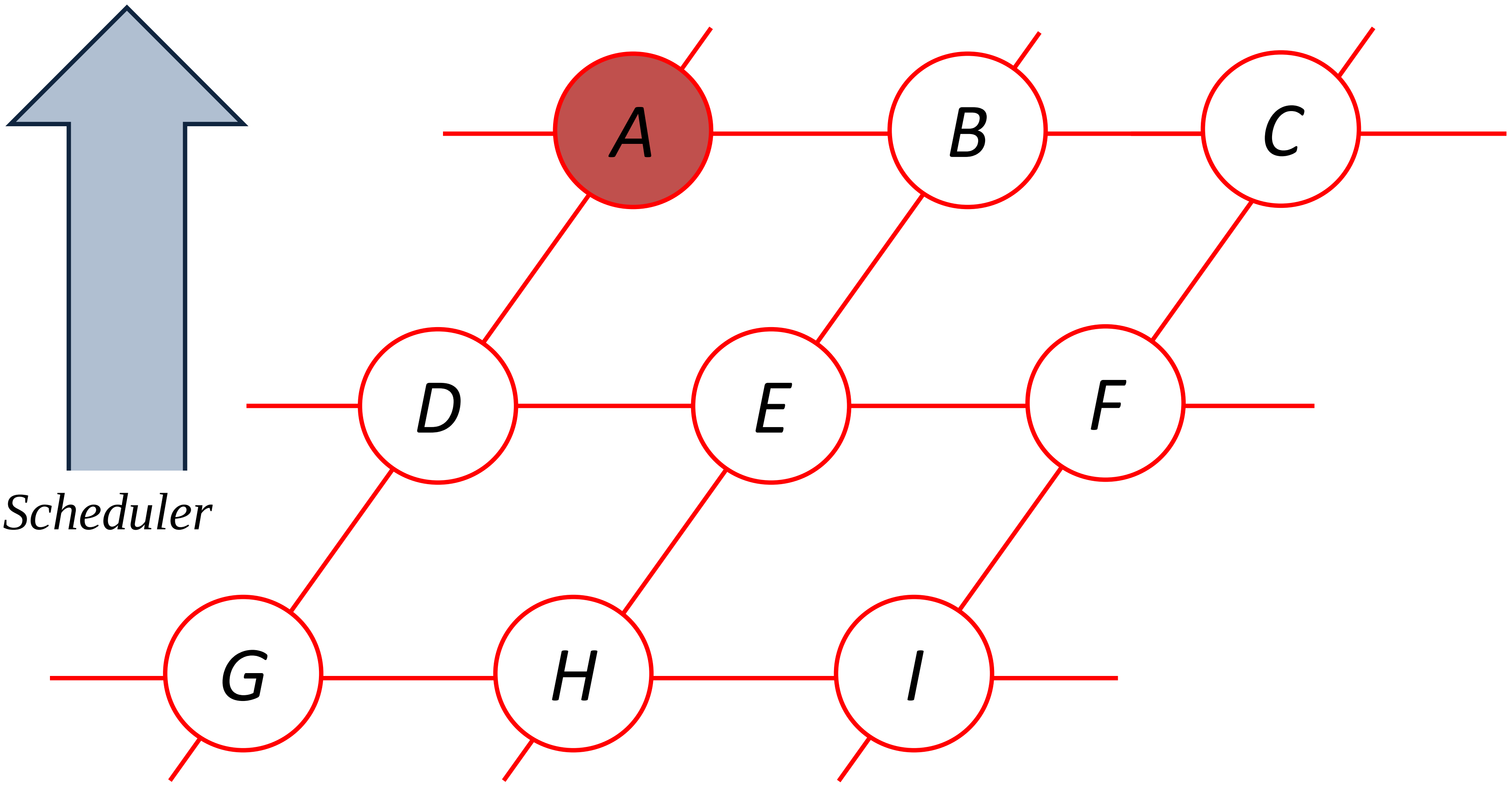
Original BP Implementation



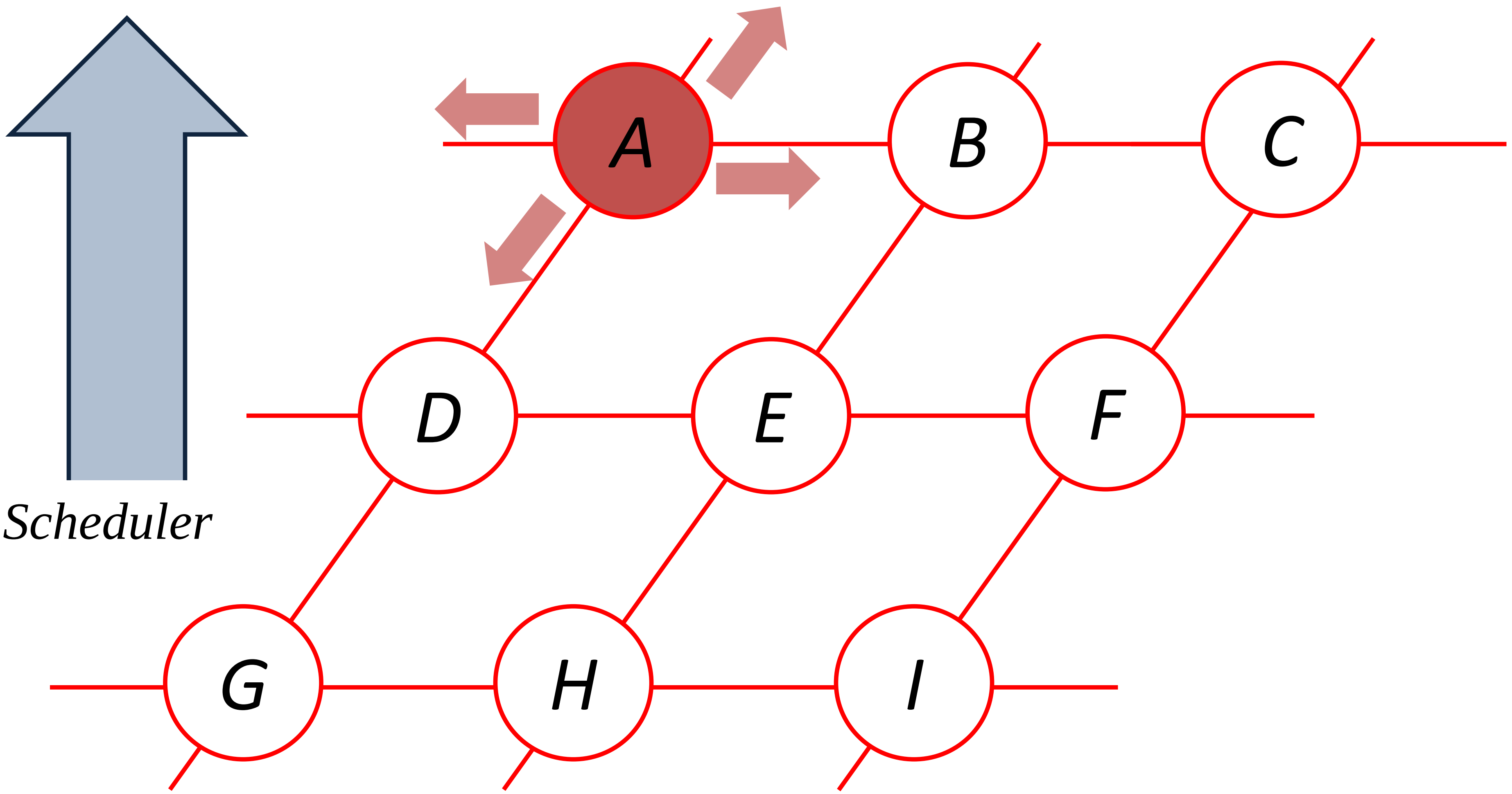
Residual BP Implementation



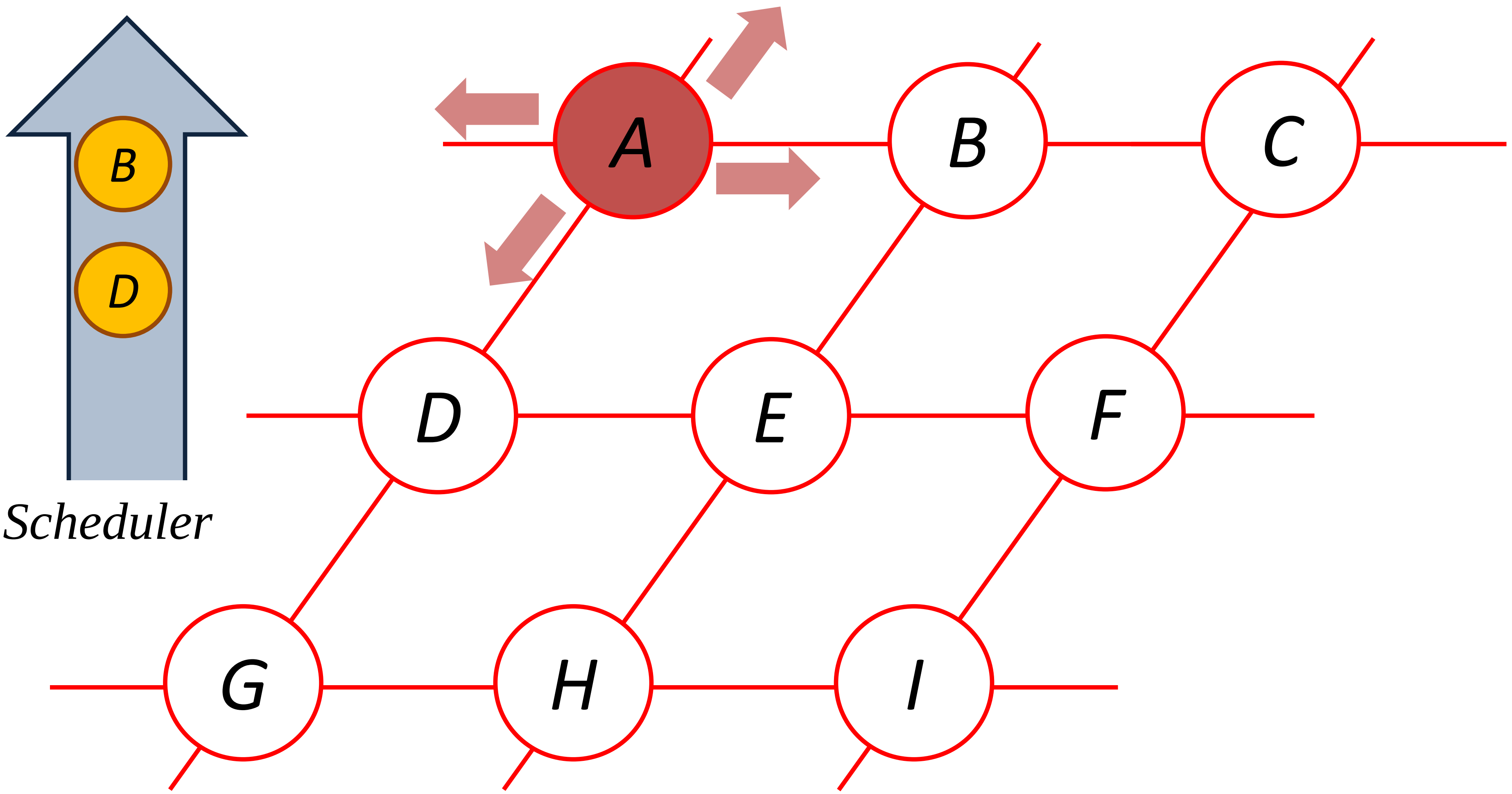
Residual BP Implementation



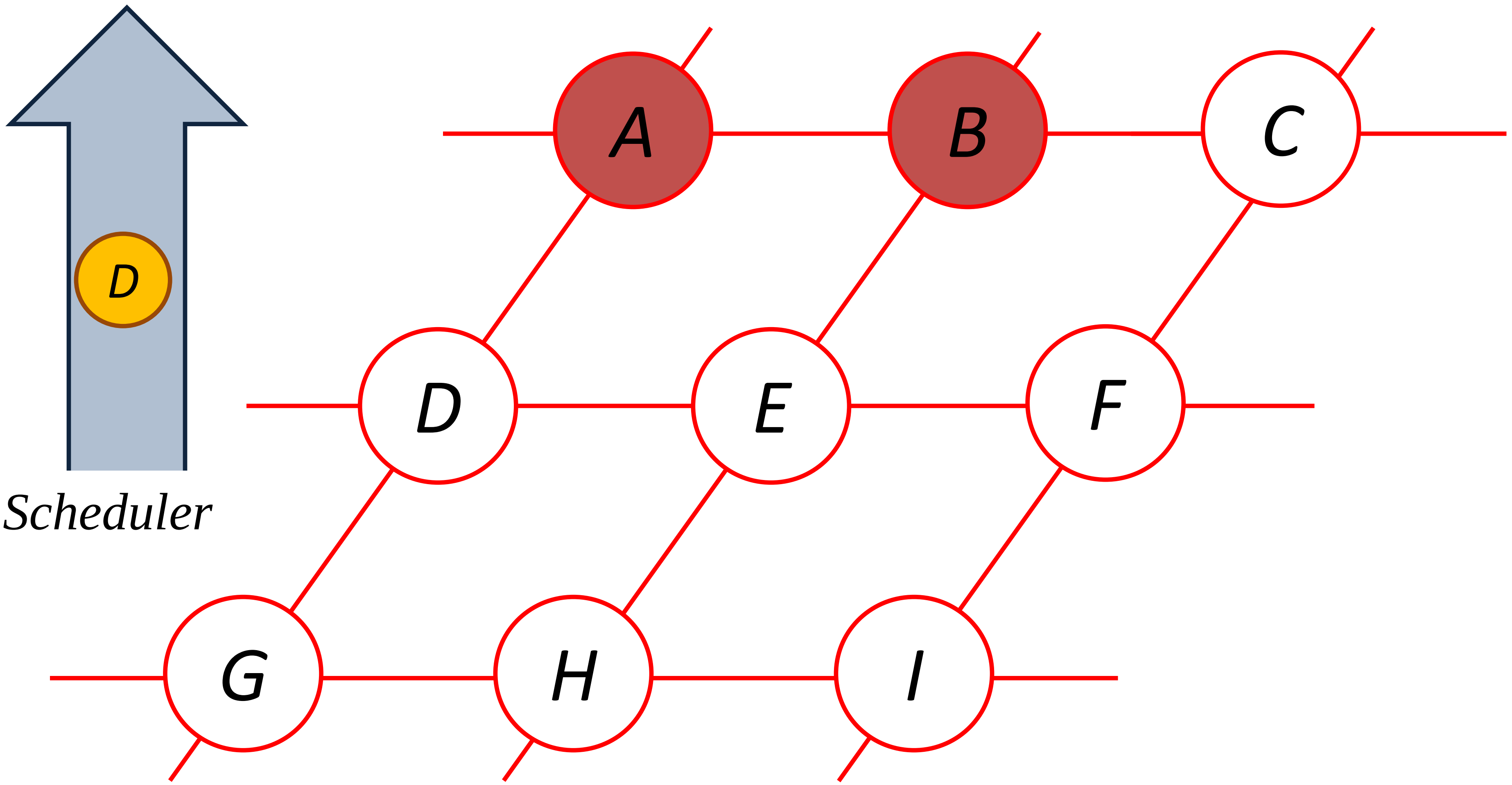
Residual BP Implementation



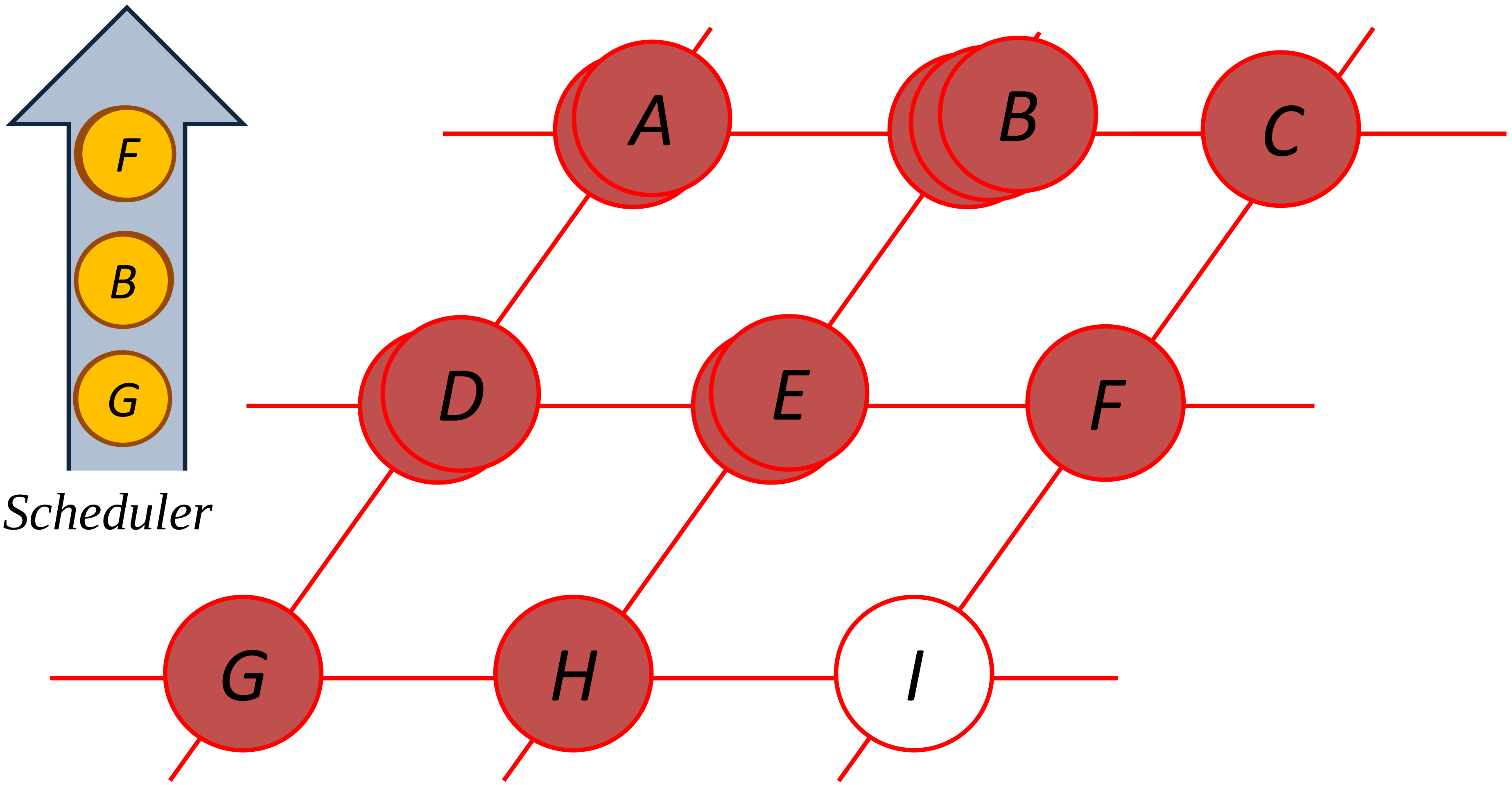
Residual BP Implementation



Residual BP Implementation



Residual BP Implementation



Comparing Original and Residual BPs

Algorithm 1: Original BP Algorithm

Initialize $b_u^{(0)}$ as ϕ_u for all $u \in V$;
Calculate the message $m_{u\to v}^{(0)}$ using $b_u^{(0)}$ according to Eq. 2 for all $u \to v \in E$;
Initialize $t = 0$;
repeat
  $t = t + 1$;
  foreach $u \in V$ **do**
    Calculate $b_u^{(t)}$ using $m_{w\to u}^{(t-1)}$ according to Eq. 1 ;
    foreach outgoing edge $e_{u,v}$ *of* u **do**
      Calculate $m_{u\to v}^{(t)}$ using $b_u^{(t)}$ according to Eq. 2 ;
    end
  end
until $\forall u \in V, ||b_u^{(t)} - b_u^{(t-1)}|| \le \epsilon$;

Algorithm 2: Residual BP Algorithm

Initialize $b_u^{(new)}$ as ϕ_u and $b_u^{(old)}$ as uniform distribution for all $u \in V$;
Initialize $m_{u\to v}^{(old)}$ as uniform distribution for all $u \to v \in E$;
Calculate message $m_{u\to v}^{(new)}$ using $b_u^{(new)}$ according to Eq. 2 for all $u \to v \in E$;
repeat
  $u = \arg\max_v (\max_{(w,v)\in E} ||m_{w\to v}^{\text{new}} - m_{w\to v}^{\text{old}}||)$;
  Set $b_u^{(old)}$ to $b_u^{(new)}$;
  Calculate $b_u^{(new)}$ using $m_{w\to u}^{(new)}$ according to Eq. 1 ;
  foreach outgoing edge $e_{u,v}$ *of* u **do**
    Set $m_{u\to v}^{(old)}$ to $m_{u\to v}^{(new)}$;
    Calculate $m_{u\to v}^{(new)}$ using $b_u^{(new)}$ according to Eq. 2 ;
  end
until $\forall u \in V, ||b_u^{(new)} - b_u^{(old)}|| \le \epsilon$;

- Computation logic is actually identical: Eq 1 and 2
- Only differs in when/how to apply this logic

GRACE:

- *Separate **vertex-centric computation** from **execution policies***
- *Customizable BSP-style runtime that enables asynchronous execution features*

Vertex-Centric Programming Model

- Update vertex data value based on received messages
- Generate new messages for outgoing edges
- Send out messages to neighbors and vote for halt

```
List<Msg> Proceed(List<Msg> msgs) {
 Distribution newBelief = potent;
 for (Msg m in msgs) {
  newBelief = times(newBelief, m.belief);
 }
 List<Msg> outMsgs(outDegree);
 for (Edge e in outgoingEdges) {
  Distribution msgBelief;
  msgBelief = divide(newBelief, Msg[e]);
  msgBelief = convolve(msgBelief, e.potent);
  msgBelief = normalize(msgBelief);
  outMsg[e] = new Msg(msgBelief);
 }
 if (L1(newBelief, belief) < eps) voteHalt();
 belief = newBelief;
 return outMsgs;
}
```

Vertex-Centric Programming Model

- Update vertex data value based on received messages
- Generate new messages for outgoing edges
- Send out messages to neighbors and vote for halt

```
List<Msg> Proceed(List<Msg> msgs) {
  Distribution newBelief = potent;
  for (Msg m in msgs) {
    newBelief = times(newBelief, m.belief);
  }
  List<Msg> outMsgs(outDegree);
  for (Edge e in outgoingEdges) {
    Distribution msgBelief;
    msgBelief = divide(newBelief, Msg[e]);
    msgBelief = convolve(msgBelief, e.potent);
    msgBelief = normalize(msgBelief);
    outMsg[e] = new Msg(msgBelief);
  }
  if (L1(newBelief, belief) < eps) voteHalt();
  belief = newBelief;
  return outMsgs;
}
```

Vertex-Centric Programming Model

- Update vertex data value based on received messages
- Generate new messages for outgoing edges
- Send out messages to neighbors and vote for halt

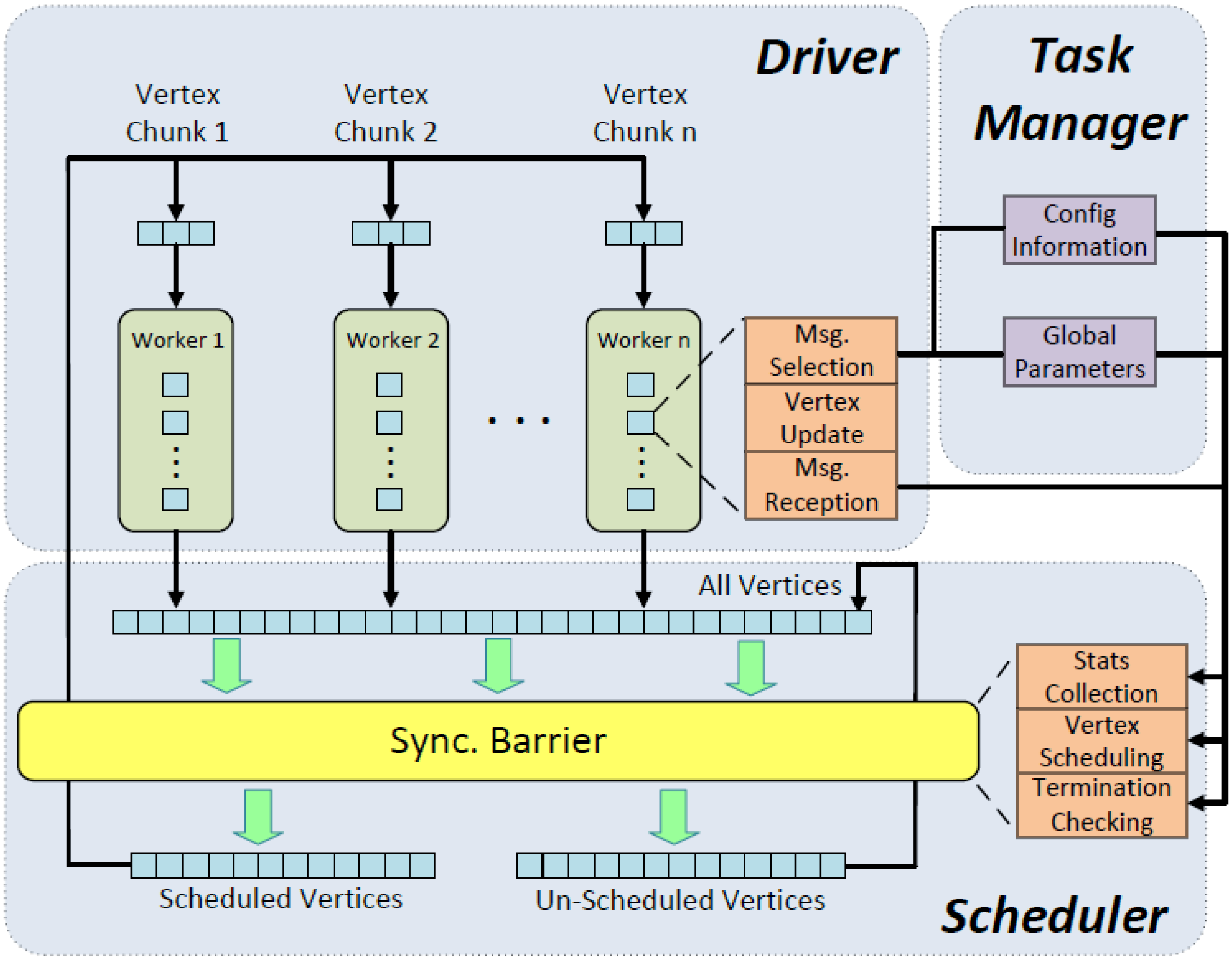
```
List<Msg> Proceed(List<Msg> msgs) {
  Distribution newBelief = potent;
  for (Msg m in msgs) {
    newBelief = times(newBelief, m.belief);
  }
  List<Msg> outMsgs(outDegree);
  for (Edge e in outgoingEdges) {
    Distribution msgBelief;
    msgBelief = divide(newBelief, Msg[e]);
    msgBelief = convolve(msgBelief, e.potent);
    msgBelief = normalize(msgBelief);
    outMsg[e] = new Msg(msgBelief);
  }
  if (L1(newBelief, belief) < eps) voteHalt();
  belief = newBelief;
  return outMsgs;
}
```

Vertex-Centric Programming Model

- Update vertex data value based on received messages
- Generate new messages for outgoing edges
- Send out messages to neighbors and vote for halt

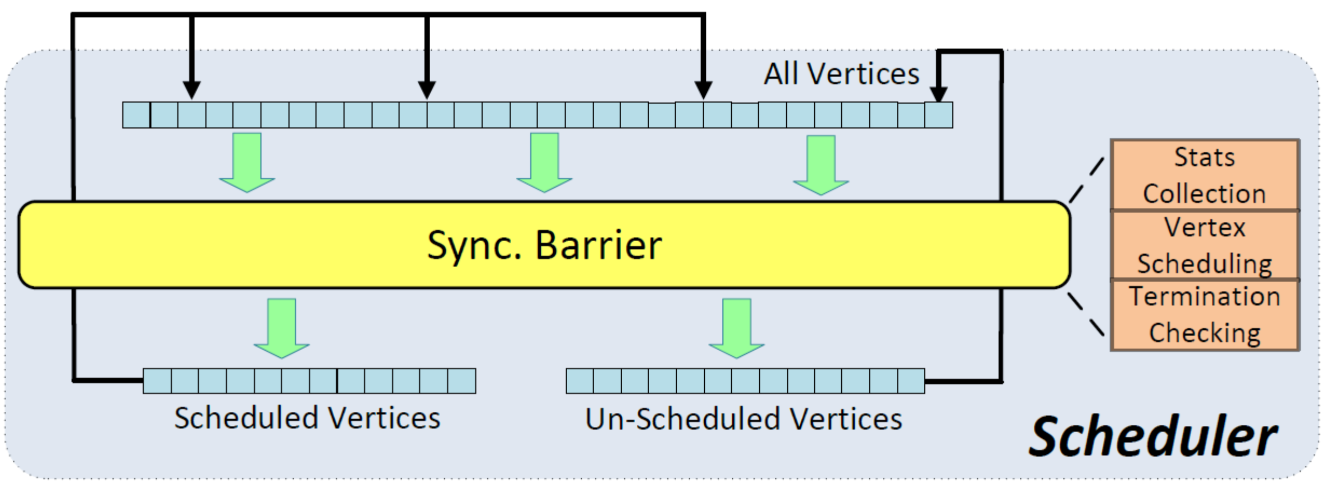
```
List<Msg> Proceed(List<Msg> msgs) {
  Distribution newBelief = potent;
  for (Msg m in msgs) {
    newBelief = times(newBelief, m.belief);
  }
  List<Msg> outMsgs(outDegree);
  for (Edge e in outgoingEdges) {
    Distribution msgBelief;
    msgBelief = divide(newBelief, Msg[e]);
    msgBelief = convolve(msgBelief, e.potent);
    msgBelief = normalize(msgBelief);
    outMsg[e] = new Msg(msgBelief);
  }
  if (L1(newBelief, belief) < eps) voteHalt();
  belief = newBelief;
  return outMsgs;
}
```

Customizable BSP-Style Runtime



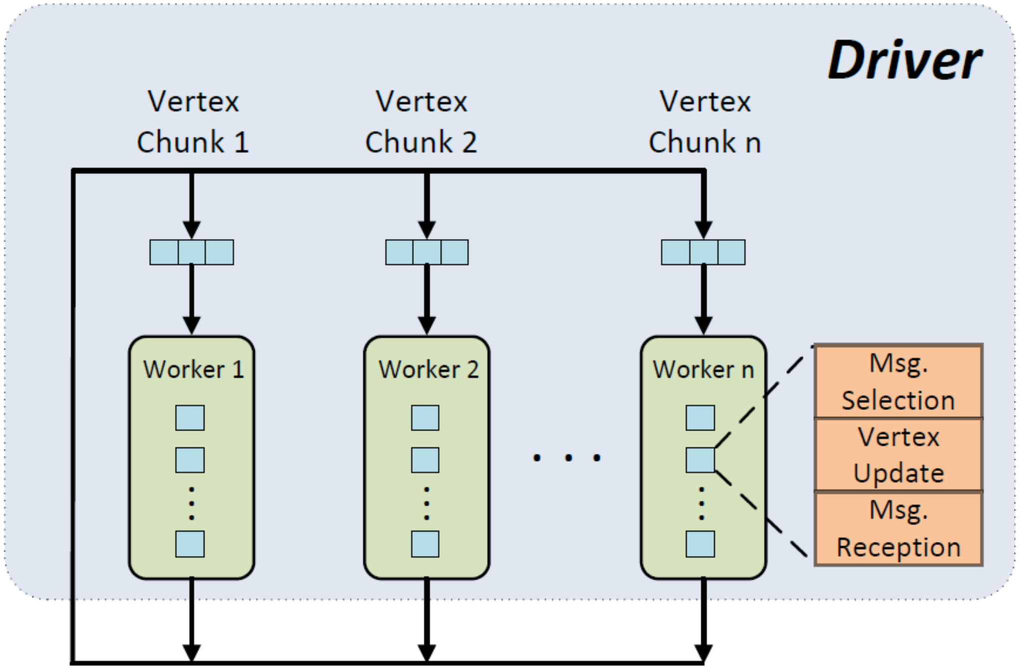
Scheduler

- At each tick barrier:
  - Check if the computation can stop
  - Collect graph data snapshot
  - Schedule the subset of vertices for the next tick



Driver

- For each worker:
  - Get a partition of the graph
  - Apply update function for scheduled vertices
  - Send newly generated messages to neighbors



- When update a vertex:
  - Choose which received messages to use
  - Specify what to do with the newly received messages

Back to Original BP

- Schedule all vertices at the tick barrier

```
void OnPrepare(List<Vertex> vertices) {
    scheduleAll(true);
}
```

- Use the message received from the last tick

```
Msg OnSelectMsg(Edge e) {
    return PrevRecvdMsg(e);
}

void OnRecvMsg(Edge e, Message msg) {
    // Do nothing since every vertex
    // will be scheduled
}
```

Back to Residual BP

- Schedule only one vertex with the highest residual
- Use the most recently received message

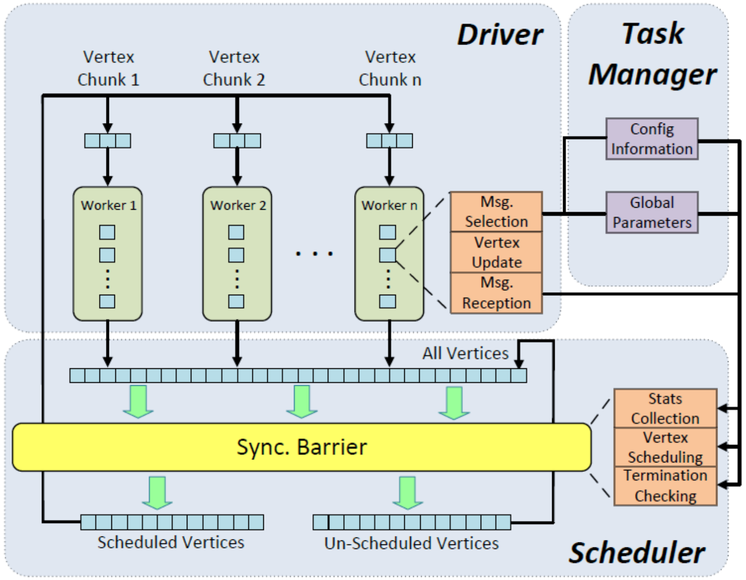
```
void OnPrepare(List<Vertex> vertices) {
   Vertex selected = vertices[0];
   for (Vertex vtx in vertices)
      if (vtx.priority > selected.priority)
       selected = vtx;
   Schedule(selected);
}

Msg OnSelectMsg(Edge e) {
   return GetLastRecvdMsg(e);
}

void OnRecvMsg(Edge e, Message msg) {
   Distn lastBelief = GetLastUsedMsg(e).belief;
   float residual = L1(newBelief, msg.belief);
   UpdatePrior(GetRecVtx(e), residual, sum);
}
```

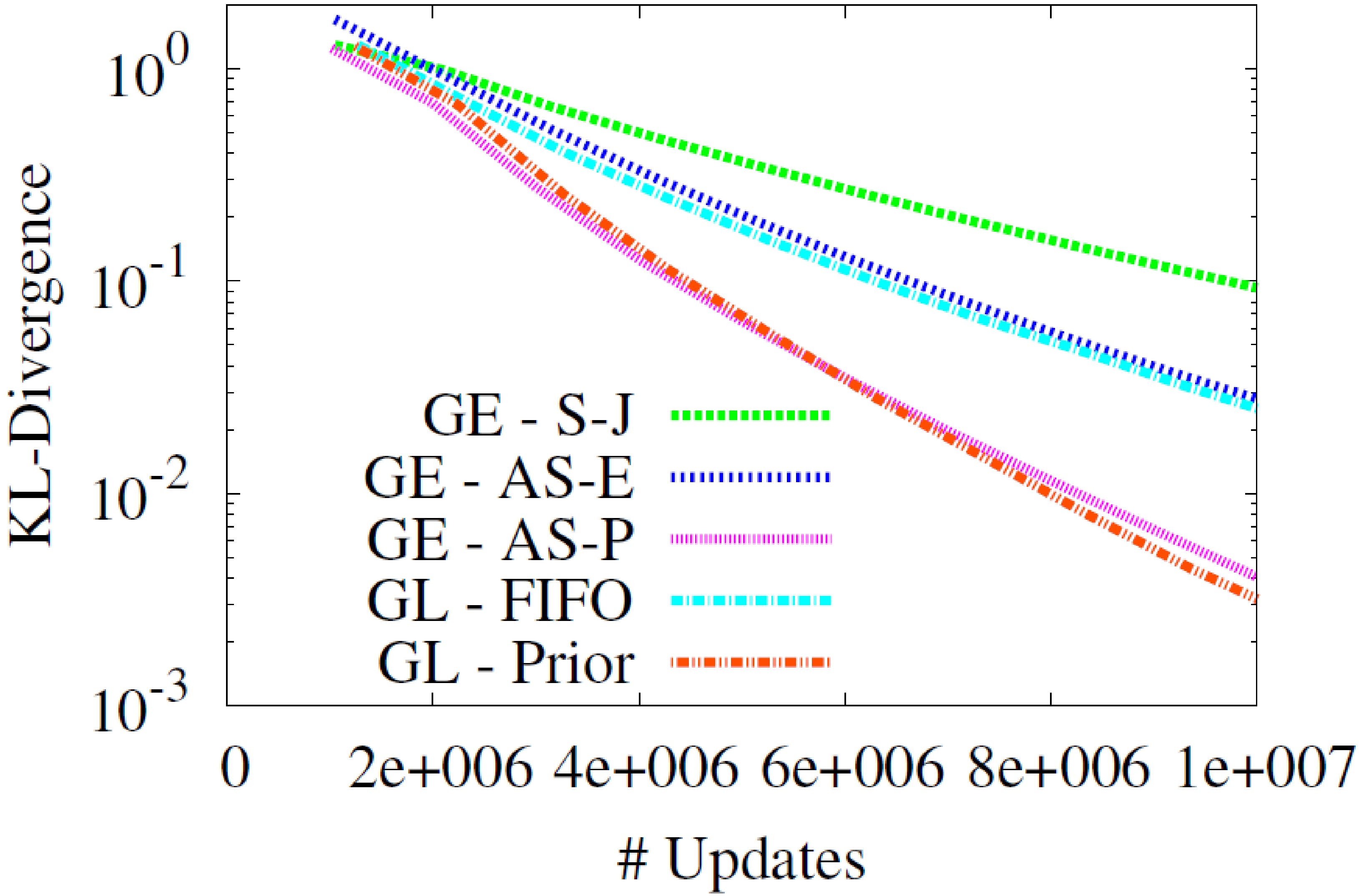
Experimental Setup

- Implementation
  - Multi-core prototype
  - Static graph partitioning
  - Four execution policies
    - Jacobi, Gauss-Seidel, Eager, Prioritized

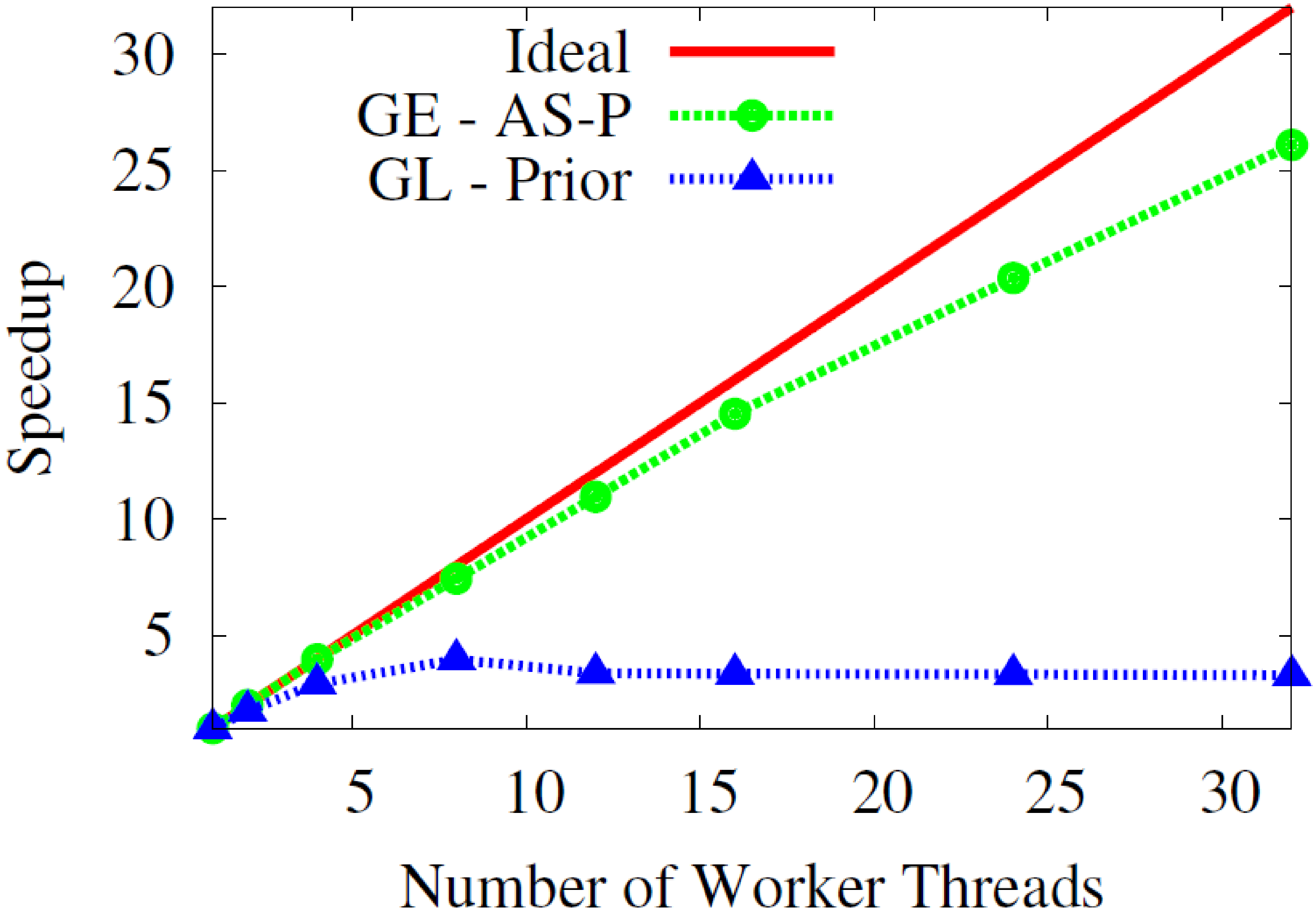


- Hardware: 8 quad-cores with 128GB RAM

Results: Image Restoration with BP



Results: Image Restoration with BP



Conclusions

Thank you!

- Graph processing: Code synchronously while execute asynchronously (if it is better)

- We can make such a development cycle easy
  - Code-once with vertex-centric programming model
  - Customizable BSP-style runtime to allow switching with various execution policies

http://www.cs.cornell.edu/bigreddata/grace/