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
- APACHE GIRAPH
- Project Pegasus
- Naiad
- PrIter
- Galois
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
• Or compute *asynchronously*
  – Data updates are (carefully) ordered
  – Data is updated using whatever available dependent state
  – Fast convergence
• 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:

\[ b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \rightarrow u}(x_u) \quad (1) \]
Running Example: Belief Propagation

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

\[ b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \rightarrow u}(x_u) \tag{1} \]
Running Example: Belief Propagation

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

\[
b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w \rightarrow u}(x_u) \quad (1)
\]

\[
m_{u \rightarrow v}(x_v) \propto \sum_{x_u \in \Omega} \phi_{u,v}(x_u, x_v) \cdot \frac{b_u(x_u)}{m_{v \rightarrow u}(x_u)} \quad (2)
\]
Original BP Implementation
Original BP Implementation

A - B - C
D - E - F
G - H - I
Original BP Implementation
Residual BP Implementation

Scheduler
Residual BP Implementation

Scheduler

A → B → C
D → E → F
G → H → I
Residual BP Implementation

Scheduler
Residual BP Implementation

Scheduler
Residual BP Implementation

Scheduler
Residual BP Implementation
Comparing Original and Residual BPs

- 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

```java
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

```java
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

```java
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

```java
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

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

- Use the message received from the last tick

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

  ```java
  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

```c
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

![Graph showing KL-Divergence vs. # Updates for different methods like GE - S-J, GE - AS-E, GE - AS-P, GL - FIFO, GL - Prior. The graph is on a log-log scale with KL-Divergence on the y-axis and # Updates on the x-axis, showing the performance of different methods over iterations.](image-url)
Results: Image Restoration with BP

![Graph showing speedup vs number of worker threads for different algorithms: Ideal, GE - AS-P, GL - Prior. The graph indicates that the speedup increases with the number of worker threads, with Ideal being the most efficient, followed by GE - AS-P, and then GL - Prior.](image-url)
Conclusions

• 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/bigreaddata/grace/