Automatic Scaling Iterative Computations

Guozhang Wang        Cornell University

Aug. 7\textsuperscript{th}, 2012
What are Non-Iterative Computations?

• Non-iterative computation flow
  – Directed Acyclic

• Examples
  – Batch style analytics
    • Aggregation
    • Sorting
  – Text parsing
    • Inverted index
  – etc..
What are Iterative Computations?

• Iterative computation flow
  – Directed *Cyclic*

• Examples
  – Scientific computation
    • Linear/differential systems
    • Least squares, eigenvalues
  – Machine learning
    • SVM, EM algorithms
    • Boosting, K-means
  – Computer Vision, Web Search, etc..
Massive Datasets are Ubiquitous

- Traffic behavioral simulations
  - Micro-simulator cannot scale to NYC with millions of vehicles

- Social network analysis
  - Even computing graph radius on single machine takes a long time

- Similar scenarios in predicative analysis, anomaly detection, etc
Why Hadoop Not Good Enough?

• Re-shuffle/materialize data between operators
  – Increased overhead at each iteration
  – Result in bad performance

• Batch processing records within operators
  – Not every records need to be updated
  – Result in slow convergence
Talk Outline

• Motivation

• Fast Iterations: BRACE for Behavioral Simulations

• Fewer Iterations: GRACE for Graph Processing

• Future Work
Challenges of Behavioral Simulations

• *Easy to program ➔ not scalable*
  – Examples: Swarm, Mason
  – Typically one thread per agent, lots of contention

• *Scalable ➔ hard to program*
  – Examples: TRANSIMS, DynaMIT (traffic), GPU implementation of fish simulation (ecology)
  – Hard-coded models, compromise level of detail
What Do People Really Want?

• A new simulation platform that combines:
  – Ease of programming
    • Scripting language for domain scientists
  – Scalability
    • Efficient parallel execution runtime
A Running Example: Fish Schools

• Adapted from Couzin et al., Nature 2005

• Fish Behavior
  – Avoidance: if too close, repel other fish
  – Attraction: if seen within range, attract other fish
  – Spatial locality for both logics
State-Effect Pattern

• Programming pattern to deal with concurrency

• Follows time-stepped model

• **Core Idea:** Make all actions inside of a tick order-independent
States and Effects

• States:
  – Snapshot of agents at the beginning of the tick
    • position, velocity vector

• Effects:
  – Intermediate results from interaction, used to calculate new states
    • sets of forces from other fish
Two Phases of a Tick

• Query: capture agent interaction
  – Read states \(\rightarrow\) write effects
  – Each effect set is associated with \textit{combinator} function
  – Effect writes are \textit{order-independent}

• Update: refresh world for next tick
  – Read effects \(\rightarrow\) write states
  – Reads and writes are totally local
  – State writes are \textit{order-independent}
A Tick in State-Effect

• Query
  – For fish f in visibility $\alpha$:
    • Write repulsion to f’s effects
  – For fish f in visibility $\rho$:
    • Write attraction to f’s effects

• Update
  – new velocity = combined repulsion + combined attraction + old velocity
  – new position = old position + old velocity
### A Tick in State-Effect

**Query**
- For fish $f$ in visibility $\alpha$:
  - Write repulsion to $f$’s effects
- For fish $f$ in visibility $\rho$:
  - Write attraction to $f$’s effects

**Update**
- new velocity = combined repulsion + combined attraction + old velocity
- new position = old position + old velocity
A Tick in State-Effect

• Query
  – For fish f in visibility $\alpha$:
    • Write repulsion to f’s effects
  – For fish f in visibility $\rho$:
    • Write attraction to f’s effects

• Update
  – new velocity = combined repulsion + combined attraction + old velocity
  – new position = old position + old velocity
A Tick in State-Effect

• Query
  – For fish f in visibility $\alpha$:
    • Write repulsion to f’s effects
  – For fish f in visibility $\rho$:
    • Write attraction to f’s effects

• Update
  – new velocity = combined repulsion + combined attraction + old velocity
  – new position = old position + old velocity
A Tick in State-Effect

• Query
  – For fish f in visibility $\alpha$:
    • Write repulsion to f’s effects
  – For fish f in visibility $\rho$:
    • Write attraction to f’s effects

• Update
  – new velocity = combined repulsion + combined attraction + old velocity
  – new position = old position + old velocity
A Tick in State-Effect

• Query
  – For fish f in visibility $\alpha$:
    • Write repulsion to f’s effects
  – For fish f in visibility $\rho$:
    • Write attraction to f’s effects

• Update
  – new velocity = combined repulsion + combined attraction + old velocity
  – new position = old position + old velocity
A Tick in State-Effect

• Query
  – For fish f in visibility $\alpha$:
    • Write repulsion to f’s effects
  – For fish f in visibility $\rho$:
    • Write attraction to f’s effects

• Update
  – new velocity = combined repulsion + combined attraction + old velocity
  – new position = old position + old velocity
A Tick in State-Effect

• Query
  – For fish f in visibility $\alpha$:
    • Write repulsion to f’s effects
  – For fish f in visibility $\rho$:
    • Write attraction to f’s effects

• Update
  – new velocity = combined repulsion + combined attraction + old velocity
  – new position = old position + old velocity
From State-Effect to Map-Reduce

Tick

Query
state $\rightarrow$ effects

Communicate Effects

Update
effects $\rightarrow$ new state

Communicate New State

Map$_1$ $t$

Reduce$_1$ $t$

Map$_2$ $t$

Reduce$_2$ $t$

Map$_1$ $t+1$

... Distribute data

... Assign effects (partial)

... Forward data

... Aggregate effects

... Update Redistribute data

Tick

Query
state $\rightarrow$ effects

Communicate Effects

Update
effects $\rightarrow$ new state

Communicate New State

Map$_1$ $t$

Reduce$_1$ $t$

Map$_2$ $t$

Reduce$_2$ $t$

Map$_1$ $t+1$

... Distribute data

... Assign effects (partial)

... Forward data

... Aggregate effects

... Update Redistribute data
BRACE (Big Red Agent Computation Engine)

• BRASIL: High-level scripting language for domain scientists
  – Compiles to iterative MapReduce work flow

• Special-purpose MapReduce runtime for behavioral simulations
  – Basic Optimizations
  – Optimizations based on Spatial Locality
Spatial Partitioning

- Partition simulation space into regions, each handled by a separate node
• *Owned Region*: agents in it are owned by the node
Communication Between Partitions

- **Visible Region**: agents in it are not owned, but need to be seen by the node
• **Visible Region**: agents in it are not owned, but need to be seen by the node

• Only need to communicate with neighbors to
  – refresh states
  – forward assigned effects
Experimental Setup

- BRACE prototype
  - Grid partitioning
  - KD-Tree spatial indexing
  - Basic load balancing

- Hardware: Cornell WebLab Cluster (60 nodes, 2xQuadCore Xeon 2.66GHz, 4MB cache, 16GB RAM)
Scalability: Traffic

- Scale up the size of the highway with the number of the nodes
- Notch consequence of multi-switch architecture
Talk Outline

• Motivation

• Fast Iterations: BRACE for Behavioral Simulations

• Fewer Iterations: GRACE for Graph Processing

• Conclusion
Large-scale Graph Processing

• Graph representations are everywhere
  – Web search, text analysis, image analysis, etc.

• Today’s graphs have scaled to millions of edges/vertices

• Data parallelism of graph applications
  – Graph data updated *independently* (i.e. on a per-vertex basis)
  – Individual vertex updates only depend on connected neighbors
Synchronous v.s. Asynchronous

• Synchronous graph processing
  – Proceeds in batch-style “ticks”
  – Easy to program and scale, slow convergence
  – Pregel, PEGASUS, PrIter, etc

• Asynchronous processing
  – Updates with most recent data
  – Fast convergence but hard to program and scale
  – GraphLab, Galois, etc
What Do People Really Want?

• Sync. Implementation at first
  – Easy to think, program and debug

• Async. execution for better performance
  – Without re-implementing everything
GRACE (GRAph Computation Engine)

• Iterative synchronous programming model
  – Update logic for individual vertex
  – Data dependency encoded in message passing

• Customizable bulk synchronous runtime
  – Enabling various async. features through relaxing data dependencies
Running Example: Belief Propagation

• Core procedure for many inference tasks in graphical models

• Upon update, each vertex first computes its new belief distribution according to its incoming messages:
  \[ b_u(x_u) \propto \phi_u(x_u) \prod_{e_{w,u} \in E} m_{w\rightarrow u}(x_u) \]

• Then it will propagate its new belief to outgoing messages:
  \[ m_{u\rightarrow v}(x_v) \propto \sum_{x_u \in \Omega} \phi_{u,v}(x_u, x_v) \cdot \frac{b_u(x_u)}{m_{v\rightarrow u}(x_u)} \]
Sync. vs. Async. Algorithms

- Update logic are actually the same: Eq 1 and 2
- Only differs in when/how to apply the update logic
Vertex Update Logic

```c
List<Message> Proceed(List<Message> msgs)
```

- Read in one message from each of the incoming edge
- Update the vertex value
- Generate one message on each of the outgoing edge
Belief Propagation in Proceed

```java
List<Msg> Proceed(List<Msg> msgs) {
    // Compute new belief from received messages
    Distribution newBelief = potent;
    for (Msg m in msgs) {
        newBelief = times(newBelief, m.belief);
    }
    // Compute and send out messages
    List<Msg> outMsgs(outDegree);
    for (Edge e in outgoingEdges) {
        Distribution msgBelief = divide(newBelief, Msg[e]);
        msgBelief = convolve(msgBelief, e.potent);
        msgBelief = normalize(msgBelief);
        outMsg[e] = new Msg(msgBelief);
    }
    // Vote to terminate upon convergence
    if (L1(newBelief, belief) < eps) voteHalt();
    return outMsgs;
}
```

• Consider fix point achieved when the new belief distribution does not change much
Customizable Execution Interface

- Each vertex is associated with a scheduling priority value

- Users can specify logic for:
  - Updating vertex priority upon receiving a message
  - Deciding vertex to be processed for each tick
  - Selecting messages to be used for Proceed

- We have implemented 4 different execution policies for users to directly choose from
Original Belief Propagation

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

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

void OnPrepare(List<Vertex> vertices) {
    ScheduleAll(Everyone);
}
```

- Use last received message upon calling Proceed, and schedule all vertices to be processed for each tick
Residual Belief Propagation

```java
void OnRecvMsg(Edge e, Message msg) {
    Distn lastBelief = LastUsedMsg(e).belief;
    float residual = L1(newBelief, msg.belief);
    UpdatePriority(residual, max);
}

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

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

- Use message residual as its “contribution” to vertex’s priority, and only update vertex with highest priority
Experimental Setup

• GRACE prototype
  – Shared-memory
  – Policies
    • Jacobi
    • GaussSeidel
    • Eager
    • Prior

• Hardware: 32-core Computer with 8 quad-core processors and quad channel 128GB RAM.
GRACE’s prioritized policy achieve comparable convergence with GraphLab’s async scheduling, while achieve near linear speedup.
Conclusions

Thank you!

• Iterative computations are common patterns in many applications
  – Requires programming simplicity and automatic scalability
  – Needs special care for performance

• Main-memory approach with various optimization techniques
  – Leverage data locality to minimize communication
  – Relax data dependency for fast convergence
Acknowledgements