Behavioral Simulations in MapReduce

JingYu Yang  Jan 25th, 2011
Outline

- Motivation & Introduction
- Behavior Simulations In The State-Effect Pattern
- MapReduce For Simulations
- Programing Agent Behavior
- Experiments
- Conclusion
What is MapReduce?

• MapReduce is a framework for processing huge datasets on certain kinds of distributable problems using a large number of computers (nodes), collectively referred to as a cluster.

• "Map" step: The master node takes the input, chops it up into smaller sub-problems, and distributes those to worker nodes.

• "Reduce" step: The master node takes the answers to all the sub-problems and combines them in some way to get the output.
What is MapReduce?

- **map**: \((k1;v1)\rightarrow[(k2;v2)]\)
  produces a set of intermediate key-value pairs
- **reduce**: \((k2;[v2])\rightarrow[v3]\)
  collects all of the intermediate pairs with the same key and produces a value
Word counting

map  <word, “1”>

“document1”, “to be or not to be”

“to”, “1”
“be”, “1”
“or”, “1”
...

Small Example
Small Example

Reduce <key, sum>

key = “be”
values = “1”, “1”

key = “not”
values = “1”

key = “or”
values = “1”

key = “to”
values = “1”, “1”

“be”, “2”
“not”, “1”
“or”, “1”
“to”, “2”

6
What are Behavioral Simulations?

- Also called agent-based simulations
- Understand large complex systems
- Tackling the ecological and infrastructure challenges of our society.
- Application Areas
  Traffic, Ecology, Sociology, etc.
Why Behavioral Simulations?

- Traffic
  - Congestion cost $87.2 billion in the U.S. in 2007
  - Evaluating proposed traffic management systems before implementing them

- Ecology
  - Use behavioral simulations to model collective animal motion, such as that of locust swarms or fish schools
  - Crucial for they affect human food security
Challenges of Behavioral Simulations

- Easy to program → not scalable
- Scalable → hard to program
- Purpose: close the gap
Requirements for Simulation Platforms

- Support for Complex Agent Interaction
- Automatic Scalability
- High Performance
- Commodity Hardware
- Simple Programming Model
• show how behavioral simulations can be abstracted in the state-effect pattern
• show how MapReduce can be used to scale behavioral simulations
• present a new scripting language for simulations
• perform an experimental evaluation with two real-world behavioral simulations
Outline

• Motivation & Introduction
• Behavior Simulations In The State-Effect Pattern
• Mapreduce For Simulations
• Programing Agent Behavior
• Experiments
• Conclusion
A Running Example: Fish Schools

• Fish Behavior
  • Avoidance: if too close, repel other fish
  • Attraction: if seen within range, attract other fish
A Running Example: Fish Schools

- Time-stepping: agents proceed in ticks
- Concurrency: agents are concurrent within a tick
- Interactions: agents continuously interact
- Spatial Locality: agents have limited visibility
Traditional Solutions for Concurrency

• Preempt conflicts
• Avoiding conflicts (Rollback in case of conflicts)

• Problems:
  – Frequency of local interactions among agents → many conflicts
  – Poor scalability
    due to either excessive synchronization or frequent rollbacks
State-Effect Pattern

- Programming pattern to deal with concurrency
- Time-stepped model
  ticks represent the smallest time period of interest
- Events occur during same tick can be reordered or parallelized
- Basic Idea: separate read and write operation
  limit the synchronization necessary between agents
State-Effect Pattern

• States:
  – public attributes that are updated only at tick boundaries
    state attributes remain fixed during a tick
    Only need to be synchronized at the end of each tick

• Effects:
  – intermediate computations as agents interact to calculate new states
    effect attribute has an associated decomposable and order-independent combinator function for combining multiple assignments
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 Phase**: agents inspect their environment to compute effects
  - Read states $\rightarrow$ write effects
  - Effect values combined using the appropriate combinator function
  - Effect writes are order-independent

- **Update Phase**: agents update their own state
  - Read effects $\rightarrow$ write states
  - Reads and writes are totally local
  - State writes are order-independent
Two Phases of a Tick

- Only way that agents can communicate is through effect assignments in the query phase
- local assignment
  agent updates one of its own effect attributes
- non-local assignment
  agent writes to an effect attribute of a different agent
• 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 Fish Simulation

• 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
The Neighborhood Property

• Synchronization at tick boundaries may still be very expensive

• Don't needs to query every other agent in the simulated world to compute its effects

• Most behavioral simulations are spatial, and simulated agents can only interact with other agents that are close according to a distance metric
The Neighborhood Property

• **visibility**
  visible region
  the region of space containing agents
  that this agent can read from or assign effects to

• **reachability**
  reachable region
  the region that the agent can move to after the update phase.

  (reachable region will be a subset of its visible region, is not required)
Outline

• Motivation & Introduction
• Behavior Simulations In The State-Effect Pattern
• Mapreduce For Simulations
• Programing Agent Behavior
• Experiments
• Conclusion
Simulations as Iterated Spatial Joins

• Since agents only query other agents within their visible regions, processing a tick is similar to a spatial selfjoin

• Join each agent with the set of agents in its visible region and perform the query phase using only these agents

• Update phase: agents move to new positions within their reachable regions and we perform a new iteration of the join during the next tick
Iterated Spatial Joins in MapReduce

- Map task
  spatially partitioning agents into a number of disjoint regions
- Reduce task
  join the agents using their visible regions

<table>
<thead>
<tr>
<th>effects</th>
<th>$\text{map}_1^{t-1}$</th>
<th>$\text{reduce}_1^{t}$</th>
<th>$\text{map}_2^{t}$</th>
<th>$\text{reduce}_2^{t}$</th>
<th>$\text{map}_1^{t+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>update$^{t-1}$</td>
<td>query$^{t}$</td>
<td>—</td>
<td>—</td>
<td>update$^{t}$</td>
</tr>
<tr>
<td></td>
<td>distribute$^{t}$</td>
<td></td>
<td></td>
<td></td>
<td>distribute$^{t+1}$</td>
</tr>
<tr>
<td>non-local</td>
<td>update$^{t-1}$</td>
<td>non-local effect$^{t}$</td>
<td>—</td>
<td>effect aggregation$^{t}$</td>
<td>update$^{t}$</td>
</tr>
<tr>
<td></td>
<td>distribute$^{t}$</td>
<td></td>
<td></td>
<td></td>
<td>distribute$^{t+1}$</td>
</tr>
</tbody>
</table>

Table 1: The state-effect pattern in MapReduce
• Partition simulation space into regions, each handled by a separate node
Communication Between Partitions

• 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.
- The map task replicates each agent to every partition that contains an in its visible region.
• Visible Region: agents in it are not owned, but need to be seen by the node

State Communication
Communication Between Partitions

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

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

Query

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

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

Update

*Owned*  *Visible*
Communication Between Partitions

- 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
Local Effects Assignment

- Map\(t_1\): tick t begins when the first map task, assigns each agent to a partition (distribute\(t\)).

- Reduce\(t_1\): outputs a copy of each agent it owns after executing the query phase and updating the agent’s effects.

- The tick ends when the next map task, map\(t_{+1}\), executes the update phase (update\(t\)).

<table>
<thead>
<tr>
<th>Effects</th>
<th>Map(t_1)</th>
<th>Reduce(t_1)</th>
<th>Map(t_2)</th>
<th>Reduce(t_2)</th>
<th>Map(t_{+1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>update(t-1)</td>
<td>query(t)</td>
<td>—</td>
<td>—</td>
<td>update(t)</td>
</tr>
<tr>
<td></td>
<td>distribute(t)</td>
<td></td>
<td></td>
<td></td>
<td>distribute(t_{+1})</td>
</tr>
<tr>
<td>Non-local</td>
<td>update(t-1)</td>
<td>non-local effect(t)</td>
<td>—</td>
<td>effect aggregation(t)</td>
<td>update(t)</td>
</tr>
<tr>
<td></td>
<td>distribute(t)</td>
<td></td>
<td></td>
<td>distribute(t_{+1})</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The state-effect pattern in MapReduce
Local Effects Assignment

Do not have non-local effects

<table>
<thead>
<tr>
<th>effects</th>
<th>map₁</th>
<th>reduce₁</th>
<th>map₂</th>
<th>reduce₂</th>
<th>map₁⁺¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>update⁻¹</td>
<td>query</td>
<td>—</td>
<td>—</td>
<td>update⁺¹</td>
</tr>
<tr>
<td></td>
<td>distribute</td>
<td></td>
<td></td>
<td></td>
<td>distribute⁺¹</td>
</tr>
<tr>
<td>non-local</td>
<td>update⁻¹</td>
<td>non-local effect</td>
<td>—</td>
<td>effect aggregation</td>
<td>update⁺¹</td>
</tr>
<tr>
<td></td>
<td>distribute</td>
<td></td>
<td></td>
<td>distribute⁺¹</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The state-effect pattern in MapReduce
Non-Local Effects Assignment

- Using two MapReduce passes
- The first map task, map^{t_1}, is the same
- The first reduce task, reduce^{t_1}, performs non-local effect assignments to its replicas (non-local effect^{t_1})
- Second map task: only necessary for distribution, not perform any computation
- reduce^{t_2}: computes the final value for each aggregate (effect aggregation^{t_1})
- Also called map-reduce-reduce model

<table>
<thead>
<tr>
<th>effects</th>
<th>map^{t_1}</th>
<th>reduce^{t_1}</th>
<th>map^{t_2}</th>
<th>reduce^{t_2}</th>
<th>map^{t_1+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>update^{t-1}</td>
<td>query^{t}</td>
<td>—</td>
<td>—</td>
<td>update^{t}</td>
</tr>
<tr>
<td></td>
<td>distribute^{t}</td>
<td></td>
<td>—</td>
<td>—</td>
<td>distribute^{t+1}</td>
</tr>
<tr>
<td>non-local</td>
<td>update^{t-1}</td>
<td>non-local effect^{t}</td>
<td>—</td>
<td>effect aggregation^{t}</td>
<td>update^{t}</td>
</tr>
<tr>
<td></td>
<td>distribute^{t}</td>
<td></td>
<td>effect aggregation^{t}</td>
<td>distribute^{t+1}</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The state-effect pattern in MapReduce
Non-Local Effects Assignment

- Map\_1 t
  - Distribute data
- Reduce\_1 t
  - Assign effects (partial)
- Map\_2 t
  - Forward data
- Reduce\_2 t
  - Aggregate effects

<table>
<thead>
<tr>
<th>effects</th>
<th>map_1^t</th>
<th>reduce_1^t</th>
<th>map_2^t</th>
<th>reduce_2^t</th>
<th>map_1^{t+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>update_1^{t-1}</td>
<td>query_1^t</td>
<td>—</td>
<td>—</td>
<td>update_1^{t}</td>
</tr>
<tr>
<td></td>
<td>distribute_1^t</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>distribute_1^{t+1}</td>
</tr>
<tr>
<td>non-local</td>
<td>update_1^{t-1}</td>
<td>non-local effect_1^t</td>
<td>—</td>
<td>effect aggregation_1^t</td>
<td>update_1^{t}</td>
</tr>
<tr>
<td></td>
<td>distribute_1^t</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>distribute_1^{t+1}</td>
</tr>
</tbody>
</table>

Table 1: The state-effect pattern in MapReduce
From State-Effect to Map-Reduce

Tick
Query
state $\rightarrow$ effects
Communicate Effects
Update
effects $\rightarrow$ new state
Communicate New State
From State-Effect to Map-Reduce

Tick

Query
state → effects

Communicate
Effects

Update
effects → new state

Communicate
New State

Map₁ t

... Distribute data
From State-Effect to Map-Reduce

- **Tick**
  - **Query** \( \text{state} \rightarrow \text{effects} \)
  - **Communicate Effects**
  - **Update** \( \text{effects} \rightarrow \text{new state} \)
  - **Communicate New State**

- **Map** \( t \)
  - **Reduce** \( t \)
  - **Distribute data**
  - **Assign effects (partial)**
From State-Effect to Map-Reduce

Tick

Query state $\rightarrow$ effects

Communicate Effects

Update effects $\rightarrow$ new state

Communicate New State

$\cdots$

Map$_1$ t

$\cdots$

Distribute data

Reduce$_1$ t

Assign effects (partial)

Map$_2$ t

Forward data

Reduce$_2$ t

Aggregate effects
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

... Distrbute data
... Assign effects (partial)
... Forward data
... Aggregate effects
... Update Redistribute data
BRACE (Big Red Agent Computation Engine)

• Special-purpose MapReduce engine for behavioral simulations

• Goal of BRACE: process a very large number of ticks efficiently, and to avoid I/O or communication overhead

• Why introducing Brace?
  behavioral simulations have considerably different characteristics than traditional MapReduce applications
BRACE (Big Red Agent Computation Engine)

Figure 1: BRACE Architecture Overview
BRACE(Big Red Agent Computation Engine)

- Shared-Nothing, Main-Memory Architecture
  expect data volumes to be modest, so BRACE executes map and reduce tasks entirely in main memory
- Fault Tolerance
  employ epoch synchronization with the master to trigger coordinated checkpoints
- Partitioning and Load Balancing
- Collocation of Tasks
Outline

• Motivation & Introduction
• Behavior Simulations In The State-Effect Pattern
• Mapreduce For Simulations
• Programing Agent Behavior
• Experiments
• Conclusion
BRASIL (Big Red Agent Simulation Language)

- High-level language for domain scientists closer to the scientific models that describe agent behavior
- Object-oriented language
- Programs specify behavior logic of individual agents
BRASIL (Big Red Agent SLimulationLanguage)

• looks superficially like Java
  The programmer can specify fields, methods, and constructors
• each field in class must be tagged as either state or effect
• query phase expressed by `run()` method
• State fields are read-only
• Effect assignments are aggregated at the effect field
• Has some important restrictions
class Fish {
    // The fish location & vel (x)
    public state float x : (x+vx); #range[-1,1];
    public state float vx : vx + rand() + avoidx / count * vx;
    // Used to update our velocity (x)
    private effect float avoidx : sum;
    private effect int count : sum;
    ...
}
class Fish {
    // The fish location & vel (x)
    ...

    /** The query-phase for this fish. */
    public void run() {
        // Use "forces" to repel fish too close
        foreach(Fish p : Extent<Fish>) {
            p.avoidx <- 1 / abs(x - p.x);
            ...
            p.count <- 1;
        }
    }
}
Effect Inversion

• An important optimization that is unique to our framework involves eliminating non-local effects.
• Rewritten expression does not change the results of the simulation, but only assigns effects locally.

```java
foreach(Fish p : Extent<Fish>) {
    avoidx <- 1 / abs(p.x - x);
    avoidy <- 1 / abs(p.y - y);
    count <- 1;
}
```
Effect Inversion

• Theorem: Every behavioral simulation written in BRASIL that uses non-local effects can be rewritten to an equivalent simulation that uses local effects only

  –Proof in the VLDB 2010 paper
Outline

• Motivation & Introduction
• Behavior Simulations In The State-Effect Pattern
• Mapreduce For Simulations
• Programing Agent Behavior
• Experiments
• Conclusion
Experimental Setup

• Implementation
  – BRACE MapReduce runtime implemented in C++, Our BRASIL compiler, written in Java and directly generates C++
  – Grid partitioning
    assigns each grid cell to a separate slave node
  – Include KD-Tree spatial indexing, rebuild every tick
  – Basic load balancing
  – Checkpointing is not yet integrated

• Simulation Workloads
  implemented realistic traffic and fish school simulations

• Hardware: Cornell WebLabCluster (60 nodes, 2xQuadCore Xeon 2.66GHz, 4MB cache, 16GB RAM)
Traffic: Indexing vs. Seg. Length

• Compares the performance of MITSIM against BRACE using BRASIL

• Without spatial indexing: Brace's Performance Degrades quadratically with increasing segment length
Fish: Indexing vs. Visibility

- increase the visibility range:
  KD-tree indexing performance decreases

- indexing yields from two to three times improvement over a range of visibility values.

Figure 4: Fish: Indexing vs. Visibility
Predator: Effect Inversion

- Effect Inversion increases agent tick throughput
  - from 3.59 million (Idx-Only) to 4.36 million (Idx+Inv) with KD-tree indexing enabled
  - from 2.95 million (No-Opt) to 3.63 million (Inv-Only) with KD-tree indexing disabled

Figure 5: Predator: Effect Inversion
• Nearly linear scalability

• Sudden drop is an artifact of IP routing in the multi-switch configuration
Fish: Scalability

- move in two different fixed directions
- Without load balancing: form in nodes at the extremes of simulated space, load at all other nodes falls to zero
- With load balancing: throughput increases linearly with the number of nodes

Figure 7: Fish: Scalability
Fish: Load Balancing

- With load balancing, the time per simulation epoch is essentially flat.
- With load balancing, the epoch time gradually increases, reflecting all agents being simulated by only two nodes.

Figure 8: Fish: Load Balancing
Conclusions

- MapReduce can be used to scale behavioral simulations across clusters
- New programming environment for behavioral simulations
  - Easy to program: Simulations in the state-effect pattern → BRASIL
    - Hides all the complexities of modeling computations in MapReduce
    - Parallel programming from domain scientists
  - Scalable: State-effect pattern in special-purpose MapReduce Engine → BRACE
    - Shared-nothing, in-memory MapReduce framework
    - Exploits collocation of mappers and reducers to bound communication overhead
Thanks!