Declarative languages for distributed systems

- A research group at UC Berkeley lead by Prof. Hellerstein:
  - claims that the problems with distributed software come from the usage of imperative sequential programming languages to describe systems that are inherently non-sequential
  - resulting systems tend to be much smaller: 20KLOC / 1KLOC for HDFS
- Related PhD theses we’ve studied in this class:
  - Peter Alvaro: Data-centric Programming for Distributed Systems, 2015
  - Peter Bailis: Coordination Avoidance in Distributed Databases, 2015. I-Confluence
Project goals

- Decided to verify claims on applicability of declarative logic programming for development of distributed software systems
- Decided to build one of the distributed data processing models presented in class
- Decided to implement Google’s Pregel, as a simple synchronous model for parallel computation based on Valiant’s Bulk Synchronous Parallel BSP model
- To test correctness of our Pregel model - implemented PageRank on top of it
Bloom Bud declarative framework

- All data is represented as collections of facts (or tables containing records)
- New facts can be derived by declaring transformational rules
  
  ```
  workers_list <= connect{ worker
       [worker.worker_addr, worker.id, false]
  }
  ```

- No shared state: nodes exchange data as network messages (Overlog)
  
  ```
  channel <= message ["IP:port recipient", "IP:port sender", payload_object]
  ```

- Introduction of notion of time - data collections evolve over time (Dedalus)
  
  ```
  counter(To,X+1) <= counter(To,X), request(To,)
  response(@From,X) <= counter(@To,X), request(@To,From)
  ```

  **When is `counter` incremented?**

  **What does `response` contain?**

**Overview**
Building Pregel using Bud Bloom declarative framework
Pregel distributed graph processing model

Worker1

Worker2

Worker3

Superstep 1

Superstep 2

Incoming Buffer

Outgoing Buffer

Incoming Buffer

Outgoing Buffer

compute

compute

compute

compute

Synchronization Barrier

Pregel implementation
Master node superstep coordination

```csharp
# start iterating
supersteps <= master_stdio {{ network_message |
  [0, false, false] if(network_message.message=="start" and graph_loaded.reveal and supersteps.empty?)
}
# table :supersteps, [:id] => [:request_sent, :completed]

# if all workers completed the superstep, start a new one
supersteps <+ workers_list.group([], bool_and(:superstep_completed)) {{ columns|
  if columns.first == true and supersteps_count.reveal < MAX_SUPERSTEPS |
  [supersteps_count+1, false, false]
end

# for the latest superstep tuple in "supersteps", send a request to Workers to start the superstep
multicast <= supersteps.argmax([], :id) {{ superstep|
  if(!superstep.request_sent and !superstep.completed)
  superstep.request_sent=true |
  ["start", {:superstep=>superstep.id}]
end
```
Worker node superstep processing

```java
state do

    table :vertices, [id] => [value, total_adjacent_vertices, vertices_to, messages_inbox]
    table :queue_in_next, [vertex_id, vertex_from] => [message_value]
    table :queue_out, [adjacent_vertex_worker_id, vertex_from, vertex_to] => [message, sent, delivered]

end

# Generating vertex messages on superstep start
queue_out <- (vertices * worker_input).pairs().flat_map do |vertex, worker_input_command|
  if(worker_input_command.message.command == "start")
    vertex_messages = @pregel_vertex_processor.compute(vertex)
  end
end

# delivery of the vertex messages to adjacent vertices for the next Pregel superstep
vertex_pipe <- (queue_out * workers_list).pairs(:adjacent_vertex_worker_id => id) do |vertex_message, worker|
  if(vertex_message.sent == false)
    vertex_message.sent = true
    [worker.worker_addr, ip_port(), vertex_message]
  end
end

# remove all outgoing vertex messages from "queue_out" in next timestep
# They are sent to recipients in the current timestep.
queue_out <- (queue_out * queue_out.group([], bool_and(:sent))).lefts

# send back a confirmation to Master that the superstep is complete
# This message is sent after all vertex_messages were *sent*, not *delivered*.
control_pipe <- queue_out.group([], bool_and(:sent)) [[vertex_messages_sent]
  [[@master_address, ip_port, "success"] if vertex_messages_sent == true
```
class PageRankVertexProcessor
  def compute(vertex)
    messages = []
    if (!vertex.messages_inbox.nil? and !vertex.messages_inbox.empty?)
      new_vertex_value = 0
      vertex.messages_inbox.each { |message|
        new_vertex_value += message[1]
      }
      vertex.value = 0.15/@graph_loader.vertices_all.size + 0.85*new_vertex_value
    end

    vertex.vertices_to.each { |adjacent_vertex|
      adjacent_vertex_worker_id = @graph_loader.graph_partition_for_vertex(adjacent_vertex)
      messages << [adjacent_vertex_worker_id, vertex.id, adjacent_vertex, vertex.value.to_f / vertex.total_adjacent_vertices]
    }
    messages
  end
end
Comparing declarative and imperative programming
Advantages - less code

```ruby
#send commands to all workers
control_pipe <- (workers_list * multicast).combos do |worker, message|
  [worker.worker_addr, ip_port, message]
end

# update workers list on job-completion messages
workers_list += (workers_list * control_pipe)
  .pairs(workers_list.worker_addr => control_pipe.from) do |worker, command|
    if(command.message.command == "load" and command.message.params[:status] == "success")
      [worker.worker_addr, worker.id, true, worker.superstep_completed]
    elsif(command.message.command == "start" and command.message.params[:status] == "success")
      #worker completed the current superstep
      [worker.worker_addr, worker.id, worker.graph_loaded, true]
    end
  end
```
Troubles, limitations

```ruby
bloom :superstep_initialization do
  # table :queue_in_next, [:vertex_id, :vertex_from] => [:message_value]
  vertices <- (vertices * control_pipe).pairs do [vertex, payload]
    if payload.message.command == "start"
      messages = []
      queue_in_next.each { |message|
        if vertex.id == message.vertex_id
          messages <<= [message[1], message[2]]
        end
      }
      [vertex.id, vertex.value, vertex.total_adjacent_vertices, vertex.vertices_to, messages]
    end
  end
end
```

Iterating over all network messages, imperative code
Demo
PageRank by matrix multiplication

<table>
<thead>
<tr>
<th>y</th>
<th>a</th>
<th>m</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
<th>r12</th>
<th>r13</th>
<th>r14</th>
<th>r15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>0.333</td>
<td>0.333</td>
<td>0.417</td>
<td>0.375</td>
<td>0.417</td>
<td>0.385</td>
<td>0.411</td>
<td>0.391</td>
<td>0.408</td>
<td>0.394</td>
<td>0.405</td>
<td>0.396</td>
<td>0.403</td>
<td>0.397</td>
<td>0.402</td>
</tr>
<tr>
<td>0.5</td>
<td>0</td>
<td>1</td>
<td>0.333</td>
<td>0.500</td>
<td>0.333</td>
<td>0.458</td>
<td>0.354</td>
<td>0.438</td>
<td>0.370</td>
<td>0.424</td>
<td>0.380</td>
<td>0.416</td>
<td>0.387</td>
<td>0.410</td>
<td>0.392</td>
<td>0.407</td>
<td>0.394</td>
</tr>
<tr>
<td>0</td>
<td>0.5</td>
<td>0</td>
<td>0.333</td>
<td>0.167</td>
<td>0.250</td>
<td>0.167</td>
<td>0.229</td>
<td>0.177</td>
<td>0.219</td>
<td>0.185</td>
<td>0.212</td>
<td>0.190</td>
<td>0.208</td>
<td>0.194</td>
<td>0.205</td>
<td>0.196</td>
<td>0.203</td>
</tr>
</tbody>
</table>

PageRank calculation by matrix multiplication:

- \( Y \text{\_next\_iteration\_value} = [0.0, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1] \)
- \( A \text{\_next\_iteration\_value} = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1] \)
- \( M \text{\_next\_iteration\_value} = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1] \)

A step:
- Vert y: send message \( \frac{1}{3} \) to g
- Vert a: send message \( \frac{1}{3} \) to a
- Vert m: send message \( \frac{1}{3} \) to a
Thank you!
TCP network communication (instead of UDP)

Support TCP channels + TCP semantics #100

neilconway opened this issue on Mar 12, 2011 · 5 comments

We currently provide unreliable channels, implemented via UDP. This is perfectly reasonable, but most applications will want something a bit more sophisticated:

1. Flow control / congestion avoidance
2. Packet fragmentation and reassembly -- trying to send large tuples via UDP is unlikely to be successful
3. Reliable delivery
4. Ordered delivery -- at least in some cases (#7)

We could implement this stuff in Bloom on top of reliable channels, but another approach would be to provide support for sending messages via TCP. This raises some interesting questions:

- TCP's reliable delivery and ordering properties are defined with respect to an individual session. How should this behavior be mapped to language semantics?
- How should error handling work?