Exploration of declarative languages applicability to development of large-scale data processing systems

December 2016

Slavik Derevyanko, Anil Pacaci

<~ bloom

UNIVERSITY OF WATERLOO

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)

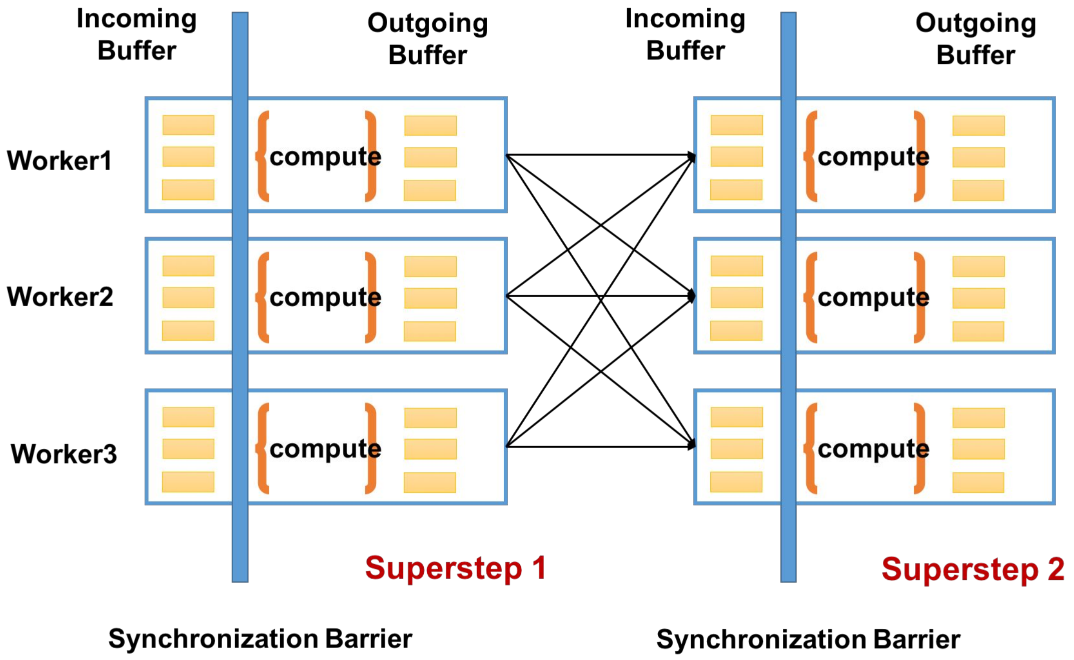
When is **counter** incremented?

```
counter(To,X+1) <= counter(To,X), request(To,_)
response(@From,X) <~ counter(@To,X), request(@To,From)
```

What does **response** contain?

Building Pregel using Bud Bloom declarative framework

Pregel distributed graph processing model



Master node superstep coordination

```
#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

```
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
}
```

PageRank implementation

```ruby
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

UNIVERSITY OF WATERLOO

Advantages - less code

```
#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

```
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
```

Iterating over all network messages, imperative code

Demo

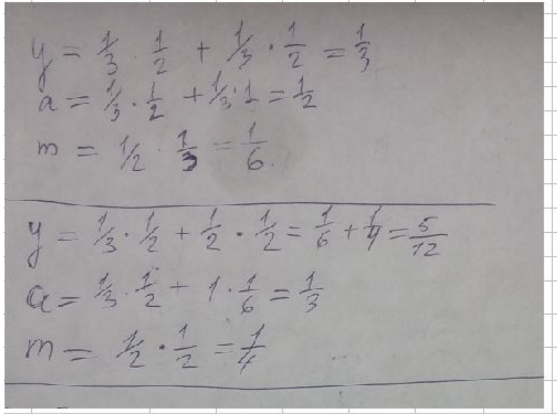
PageRank by matrix multiplication

	y	a	m
y	0.5	0.5	0
a	0.5	0	1
m	0	0.5	0

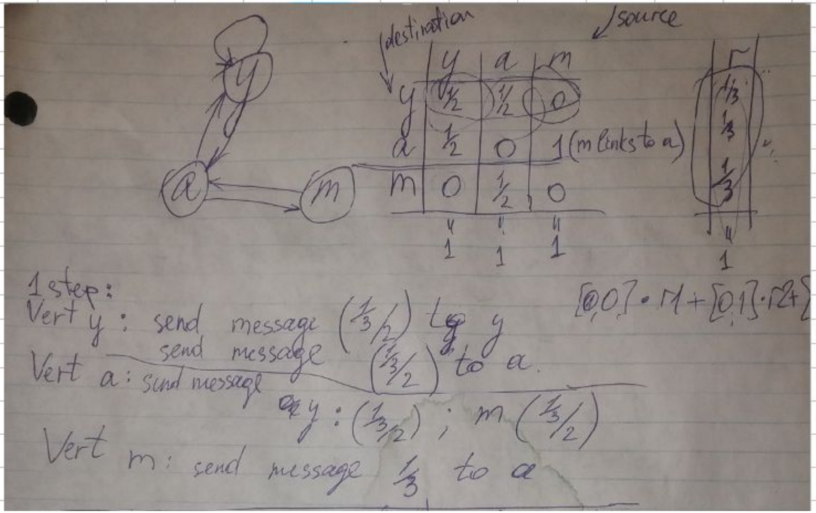
r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15
0.333	0.333	0.417	0.375	0.417	0.385	0.411	0.391	0.408	0.394	0.405	0.396	0.403	0.397	0.402
0.333	0.500	0.333	0.458	0.354	0.438	0.370	0.424	0.380	0.416	0.387	0.410	0.392	0.407	0.394
0.333	0.167	0.250	0.167	0.229	0.177	0.219	0.185	0.212	0.190	0.208	0.194	0.205	0.196	0.203

PageRank calculation by matrix multiplication:

Y_next_iteration_value: r2[0]=[0,0]*r1[0]+[0,1]*r1[1]+[0,2]*r1[2]
A_next_iteration_value: r2[1]=[1,0]*r1[0]+[1,1]*r1[1]+[1,2]*r1[2]
M_next_iteration_value: r2[2]=[2,0]*r1[0]+[2,1]*r1[1]+[2,2]*r1[2]



PageRank weights assignment using Pregel vertex-centric model:



UNIVERSITY OF WATERLOO

TCP network communication (instead of UDP)

bloom-lang / **bud**

Watch 45 | Star 483 | Fork 42

Code | Issues 97 | Pull requests 1 | Projects 0 | Wiki | Pulse | Graphs

Support TCP channels + TCP semantics #100

New issue

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

neilconway commented on Mar 12, 2011 Member

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?

Projects

None yet

Labels

research

wishlist

Milestone

No milestone

Assignees

No one assigned

2 participants

UNIVERSITY OF WATERLOO