Building Data-parallel Pipelines in Erlang

Pero Subasic

Erlang Factory, San Francisco

March 29, 2012

Outline

- An introduction to data-parallel
- Requirements
- Map-reduce Flows Recap
- Architecture Overview
- Flow Specification Language
- Iterative Flows - Concept Rank Flow
- Results
- Conclusion and Future

Requirements

Computation models
- map-reduce
- iterative and incremental

Processing models
- in stream, stateful
- in stream, stateless
- batch

Computation platform
- Cloud
- Virtualized, general
- Bare metal

Data Parallel Trend

Scientific Computing Tools

- R -> snow, multicore, parallel, RHIPE, R+Hadoop, etc.
- Mathematica -> gridMathematica
- MatLab -> Parallel Toolbox

Internet, Big Data

- Yahoo: S4
- Google: FlumeJava
- Cloudera: Crunch, Hadoop MR2

Parallelism granularity

- GPGPU
- Multi-CPU/Multicore
- Cluster

4

The concept is not really new. It is easy to find online a paper “Data Parallel Algorithms” by W. Daniel Hillis and Guy L. Steele, Jr. from December 1986 issue of Communications of ACM where they talk about data parallel algorithms where “their parallelism comes from simultaneous operations across large sets of data rather than from multiple threads of control.” Here it applies mostly to machines with hundreds or thousands of processors.

Erlang has data parallelism.

Since functions are first-class objects, they can be dispatched wherever we want them in the distributed system, on-demand, and with the data thereby enabling the concept of bringing computation to the data.

Data Parallel

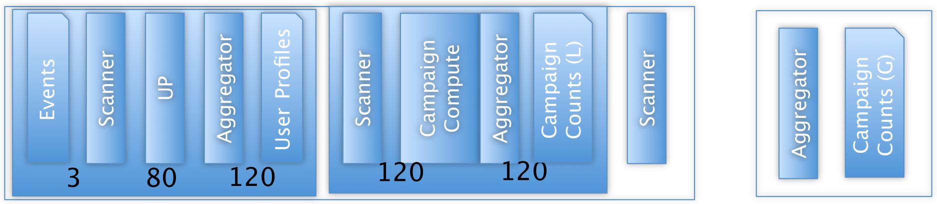
Given an integer vector x

$$x : x[i], i = 1, n$$

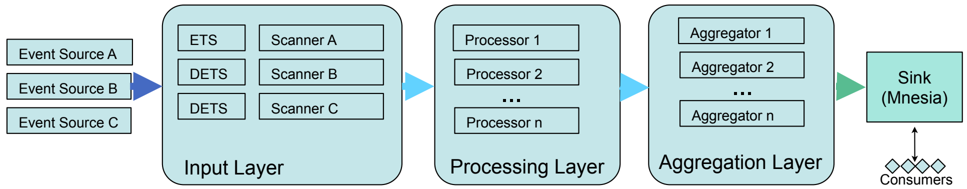
apply a function to vector elements *in parallel*

$$\| \, f(x)$$

Map Reduce*3 Flow



An Example Flow

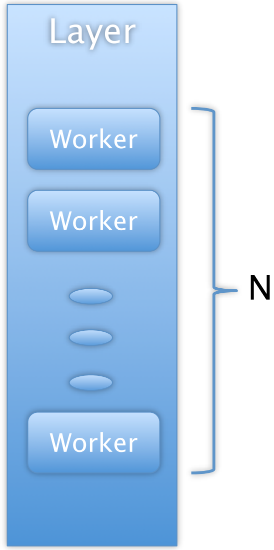


Architecture

Architecture

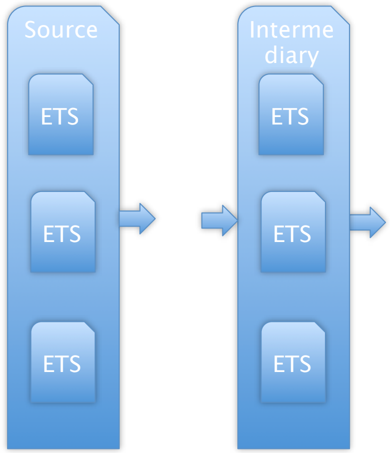
- Flow supervisor/monitor – layer – worker hierarchy
- ETS/DETS/Mnesia/TCP/UDP tables for sources, sinks and intermediaries
- Synchronous or asynchronous message passing between layers
- Plugins
- Example layer parameters:
  - Layer size
  - Layer identification
  - Layer input, output data/format and connectivity with adjacent layers
  - Mapping functions between layers: partitioning

Layer



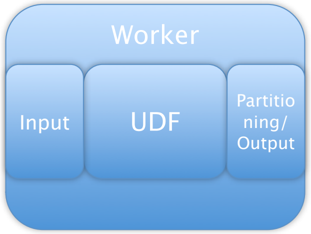
- Process
- Spawns and monitors workers
- Elastic (number of workers)
- Workers perform uniform functions
- Connects to other layers, sources, sinks, intermediaries, resources
- Maps to physical nodes

Sources, Sinks and Intermediaries



- Storage for crucial data sets
- Staging used for input for one or more flows
- Intermediary: output from previous layer, input to the next one; implementing barrier concept
- Both can be used for staging input for multiple downstream layers/flows

Workers



- Input layer: predefined for ETS scan
- UDF assumes intermediary/staging table record format
- UDF defined externally, still Erlang
- Partitioning/load balancing function: predefined or custom
- Output to intermediary or another layer
- All functionality local to a node

Flow Language

Flow Language: Configuration Hierarchy

Application
Platform
Infrastructure

- Infrastructure
  - Cloud/VM
  - Hardware
- Platform/Framework
  - Erlang node configuration
  - Code repository: framework, plugins, global libraries (Erlang, C/C++, CUDA)
- Applications
  - application libraries
  - **flow**
    - **flow infrastructure (TCP, UDP, ETS, DETS, Mnesia)**
    - **flow structure: flow graph (nodes, communication)**
    - **flow replication**
  - **optimization**
  - **monitoring**
  - scheduling

Flow Language

```
% [illegible]ers, workers
{[illegible]"}]},
        {[illegible]/conf/flow1.cfg"},...]
        {[illegible]1T}]},
        {[illegible]ner},
         {[illegible]},
[illegible] {disk, 1T}]}
         {[illegible]/data"},
         {[illegible]},...}
[illegible]
  }      ] {communication, {concurrent, asynchronous}}
     }]},
     {lay[illegible]anner, [
         {type, [illegible]scanner},
         {predece[illegible]_events}, % source specs for the scanner
         {s[illegible]parallel_scan}, % scan records in order, in
parallel fa[illegible]ator, aggregator_fun, []}}, % aggregates all fields with a
common ACID/T[illegible] Mnesia table
         {c[illegible], [adtech]}}, % user-defined
partitioning[illegible]adcom event records
         {s[illegible]synchronous}}
     ]]},...
  } ]},
```

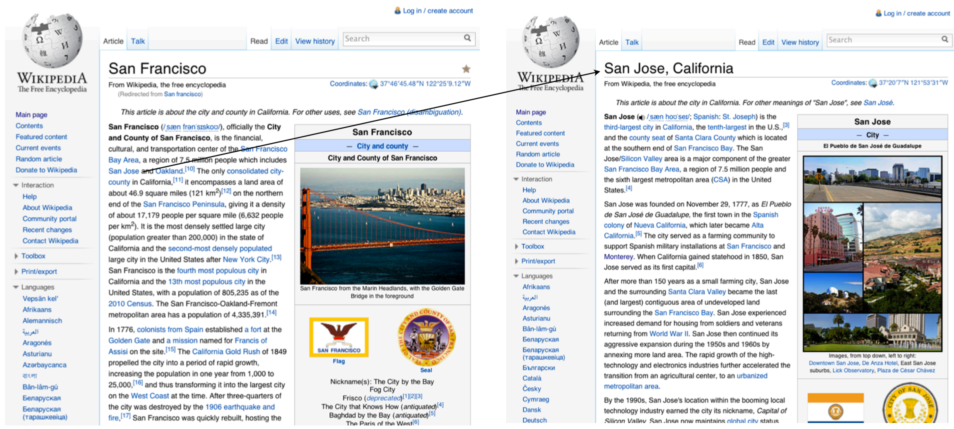
Iterative Flow – Concept Rank

Wikipedia Concept Rank Problem Space

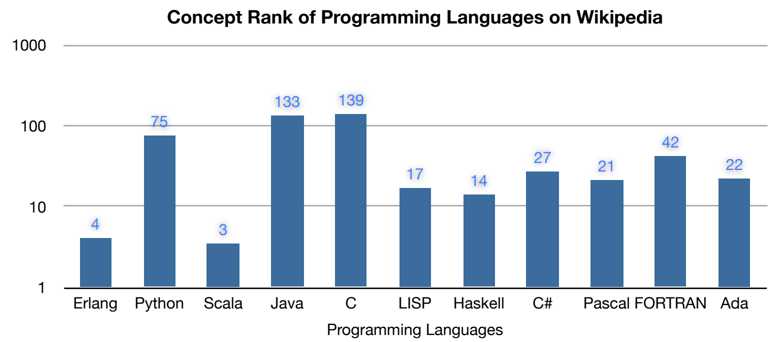
3,091,923 concepts

42,205,936 links

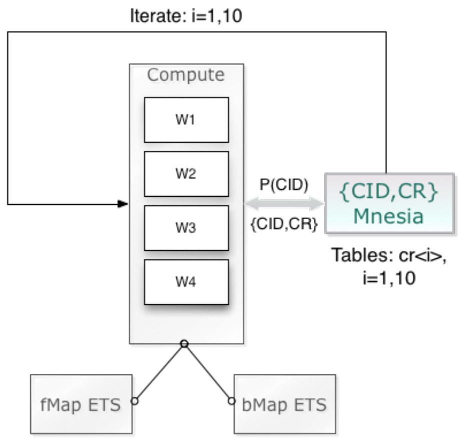
$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$



Concept Rank Results

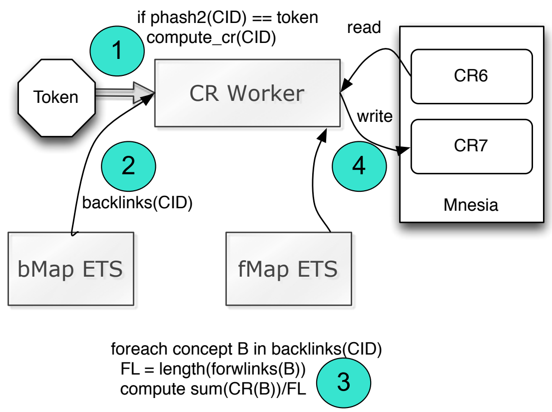


CR Flow



Iterate: i=1,10
Compute
W1
W2
W3
W4
P(CID)
{CID,CR}
{CID,CR}
Mnesia
Tables: cr<i>,
i=1,10
fMap ETS
bMap ETS

CR Compute Worker



Computation Metrics and Results

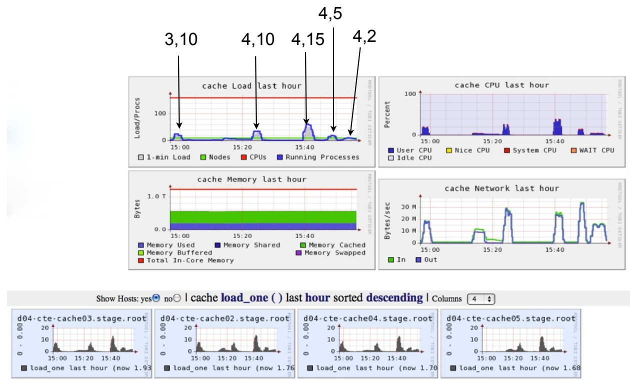
- memory use: 420m RAM, 63m resident memory
- Mnesia is distributed across nodes.
- fMap and bMap are loaded on each node in ETS tables
- fMap 126M, 3,091,923 records
- bMap 121M, 2,479,969 records
- conceptrankX tables in Mnesia -> 3,091,923 records, 31,374,390 words of memory each

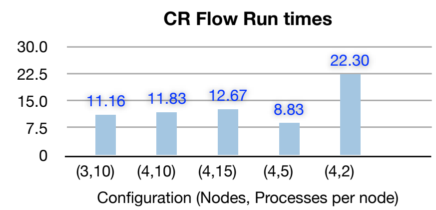
Runtimes, 10 iterations:

* 1 worker process, 1 node: 50 minutes (time for 10 iterations)
* 10, 1: 18.36 min
* 20, 1: 23.48 min
* 5, 1: 19.8 min
* 2 nodes, 10 proc/node: 14.83 min
* 3 nodes, 10 proc/node: 11.16 min low network traffic 10MB/sec; nice low traffic and fast response with not so high CPU usage
* 4 nodes, 10 proc/node: 11.83 min med-high to high 30MB/sec
* 4 nodes, 15 proc/node: 12.67 min med-high network traffic 25MB/sec
* 4 nodes, 5 proc/node: 8.83 min high network traffic 35MB/sec *
* 4 nodes, 2 proc/node: 22.3 min medium network traffic 15MB/sec across all four nodes *

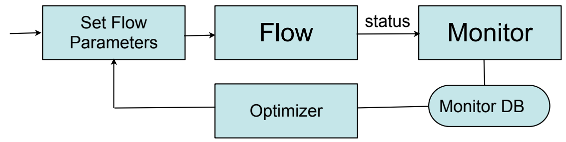
- finding compromise point between network traffic due to Mnesia table sync and local computation requirements on each node - seems like 5 proc/node minimizes response time at the expense of high network traffic. Reducing number of processes per node to 2 reduces network traffic, but impacts computing capacity (CPU utilization is lowest of all aproaches). So, in that case, system spends most time computing the ranks.

Concept Rank Flow: Tradeoffs





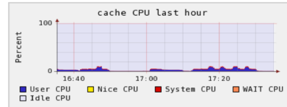
Optimization

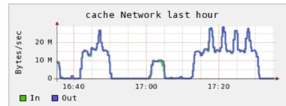


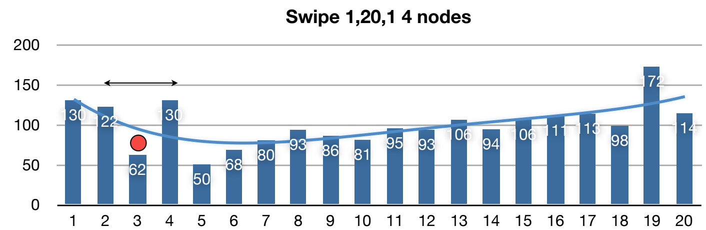
Minimize

- run time
- network traffic
- cpu utilization
- disk i/o
- cloud expense
- some combination of the above

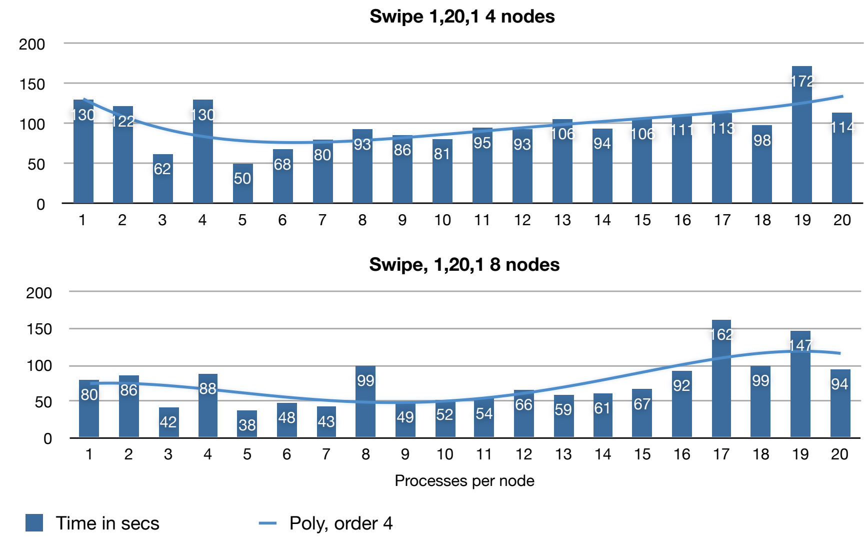
Run Time Local Minima



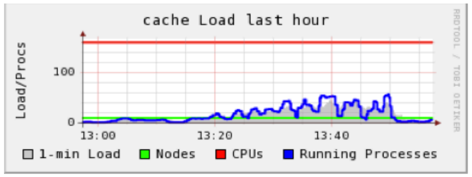


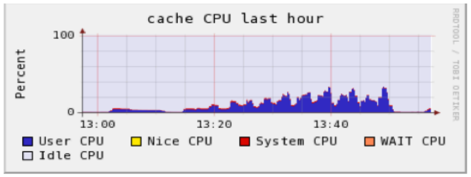


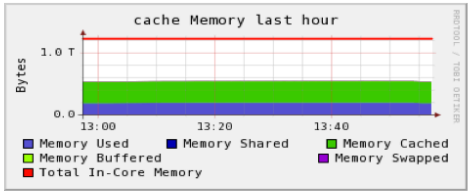
Swipe Results

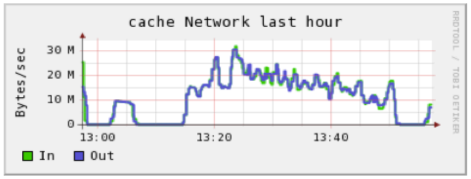


Swipe Ganglia Monitoring









Conclusion and Future

- Erlang is very convenient and appropriate language and platform for data-parallel flows
- Building languages and platforms makes sense to facilitate easy flow specification
- Small Erlang team can do wonders

Questions, Comments?