0xdata
Big Data
Small Computers

Cliff Click
cliffc@0xdata.com
http://cliffc.org/blog
http://0xdata.com

agenda

- Marketing / Biz
- RAIN & H2O
- JMM & Parallel Computation
- Fork / Join
- Goals
- Reality Check

Marketing / Biz Model

- Now: Tb-sized single drives, Pb-sized farms
  - Tons of cheap disks
  - Capture **all** data, including bulky low-value
  - Not worth putting in Oracle DB
- Hadoop / HDFS – free version of GFS!
  - Manages Tbs to Pbs of low-value data
- But can we get value out of this data?

Marketing / Biz Model

- Big Data – needs Big Math
- The Stats guys Own the Math
  - Linear Regression? LogReg? SVM? RandomForest? (sub)Sampling?
  - Need their expertise to grok the data
- Stats guys – use R, SAS
  - No Java, no "classic programming"
  - R, SAS do not distribute, scale

Marketing / Biz Model

- Follow Google again into Map/Reduce
  - Simple programming style
  - Robust at scale, very parallel, very big
- Downside(s):
  - Hard to program M/R & complex to manage
    - Harder to find M/R programmers
  - Slow / batch: nothing in 4hrs, all in 5hrs

Marketing / Biz Model

- Market is ripe for a Better Mousetrap!
  - Follow the Data:
    - Must Read/write HDFS!
  - But better programming model than M/R
  - Easier to program, wider app domains
    - Incremental; small syntax for small jobs
  - Easy to setup & manage
    - One-jar no-args install
    - Auto-discovery

What are the Resources?

- Cheap, large disks
- Attached to modern multi-core X86's!
    - Typically with 8-32gig ram
    - Racked & stacked: fast networks
- Java + Linux everywhere
- Cheap X86 moboards
    - They fail occasionally or
    - Java GC pauses' or ....

RAIN

- Redundant Array of Inexpensive Nodes
  - Can we make a RAID-like structure for computation?
  - "Single-System Image"
  - Silent recovery from failed Node
  - Variable amount of parallelism over time
  - Rules on communication, recovery
- Programming Model?

H2O

- Distributed Non-Blocking HashMap
  - Semi-classic DHT
    - 100% peer-to-peer, no master
    - Keys spread around the cloud
  - Hash function for distribution
    - Can find Key/Value in 1 Node probe
    - (plus some # of replicas)
  - Nodes drop in & out ==>
    - Hash function changes to match

H2O

- Distributed HashTable
  - All Keys can be cached **locally**
  - **But** might only appear on the replicas
  - All Nodes know all Keys' replicas
  - If Key misses locally:
    - can query replica with 1 UDP packet
  - Any Key can be written to any Node
    - And it will "repair" to the replicas

Memory Model

- **Most** apps will need bulk-append (fast)
  - With replication & repair & rebalance
  - And fast random-read (for queries)
  - Write-only data!
    - No changing old key/values!
- But some crucial apps need more:
  - e.g. controlling complex code flow

Java Memory Model

- What are the rules when K/V's change?
  - It's a Memory Model!
  - The JMM is well understood
    - Millions of Java programmers
  - "volatile" K/V reads & writes
- Invoking the JMM is always an option
  - And might be slower than plain K/V ops
  - Invokes hardware-cache-coherence

Coherence

- “normal” Key/Value Put/Get are “weak”
  - Ordered relative to issuing Node
  - Unordered for others
  - Racing Puts will agree on a final Value
- Atomic R/M/W (without ordering)
  - e.g. Counters
  - Really: single-Key transactions
- “volatile” Put/Get available, slower

Reliable Computation

- What computation can be reliable?
  - If we ***expect*** Nodes to fail?
- Checkpointing & re-execute
  - Tried & true
  - But user has to define checkpoints
- Also need massively parallel computation
- Enter Distributed Fork/Join

Distributed Fork Join

- What is Fork/Join?
  - A parallel programming paradigm
  - Recursive-descent divide & conquer
    - With good load-balance properties
    - And typically good cache behavior
  - Break a Big Task up into multiple little jobs
  - Or run little jobs directly
- DFJ: Same thing, distributed

FJ Tasks + Futures

- A FJ Task is really a continuation
  - Of a logarithmic-sized program chunk
  - Execution produces a result
  - The join step merges 2 1/2-sized results
  - And either can store results in the Future
- i.e. It's a perfect checkpoint
- And easy to pass around between JVMs
  - Treat as a system Key/Value

DFJ – Reality Check

- DFJ: Distributed Divide & Conquer
- Very expressive, very flexible
    - Big data, parallel arrays, graph programs
- Good/free load-balance & cache re-use
- Constant log-sized checkpoints
- Nice model for experts to use
- But maybe too low-level for general use

Analytics & General Usage

- Easier model: Run General Java
    - Except at Scale, with multiple users
- So not really “general Java”
    - Make static globals “user-local”
    - Hook System.out,.err,.exit
    - "Syntax error" for JNI calls
    - Allocations are thread-local, etc, etc
- Basically make it resource-safe

Analytics & General Usage

- Auto-Parallelize the embarrassingly parallel
  - Simple outer loops or loop nests
  - Insert DFJ calls as needed
  - Syntax Error if cannot parallelize
    - No parallel means running 100x slower
    - (escape hatch as needed)
- Similar to Vectorizing Compiler experience
  - Train programmers via compiler

Other Ease of Use

- REST API
  - Submit jobs, view progress & results via browser (or curl or Excel!)
- Most “plain Java” just runs
- Replace collections with dist & par versions
  - (by the compiler, under the hood)
  - Distributed collections a must!
  - Allows for results >>> single disk

Some Parallel Java

- List<String> listOfStrings = … // Very Large
  for(String s : listOfStrings)
    if(s.indexOf("wonky") != -1) // filter &
      cnt++; // atomic-count
- for(int i=0; i<N; i++) { // 1st pass of a LR
    sumx += X[i];
    sumy += Y[i];
    sumx2 += X[i]*X[i];
  }

Column Store

- Data is fixed-count of columns, in rows
  - Variable-length format: log files, csv, text
  - Dozens to 1000's of columns
  - ***Billions*** of rows
- Want: fixed-length rows for array access
  - Dense linear scans, and random access
- Dense: more data in RAM, in cache, cpu
  - Dense is faster at all levels

Column Store

- Goals: dense & fast-access
- Columns are usually restricted-range
  - e.g. age from 0 to 127
  - boolean: sex, has_cancer, owns_home
  - Or highly correlated
- Easy to do compression
  - Find column range, or count of uniques
  - Scale, shift, offset

Column Store

- Column is 1,2,4 or 8 bytes, or bitvector
  - With scale/size/base/off factors
  - Typically 2x to 10x compression
- Data is unpacked in hot inner loops
  - Ex: add all of column 5:

```
for( int i=0; i<N; i++ )
  sum += A[i].col(5);
```

  - Ex: col 5 is boolean “homeowners”

```
for( int i=0; i<N; i++ )
  sum += (A_bits[i]>>5)&1;
```

HDFS & Map / Reduce

- We support HDFS directly
  - HDFS files are Keys already
  - Can pass in to bulk / parallel ops directly
- Support Map / Reduce directly
  - Trivial translation from M/R to DFJ
  - Except we use zillions of tiny maps
  - And power-of-2 reduces

Reality Check

- Missing, in progress:
  - Distributed F/J: no cross-node stealing yet
  - No compiler
    - We're hand-doing most compiler ops now
    - Some simple rewriting is working
    - We do have compiler experts on team
  - Scale to 1000; right now tested to "only" 20

What Is H2O?

- Distributed Key/Value Store
  - (but so was Linda)
- Auto replication & repair & persistence
  - (like Hadoop)
- Easy management
  - One-jar install & deploy
  - Distributed Dashboard

Reality Check

- Fast, simple **coherent** model
  - Cache reads & writes everywhere
- UDP for control plane, TCP for bulk xfers
  - Reverse of the usual, but UDP is fast
  - ...and highly reliable in racks
- Basic D/F/J solid, working

What Is H2O?

- Distributed coherent K/V store
- Distributed Fork/Join framework
- Cloud dashboard with profiling
- A dense column-store
- A compiler for auto-distributing code
- A collection of Math / Analytics utilities

Summary

- Programming Model for the Rack
  - Bulk / parallel by default
  - Replicate & repair **computation**
  - "JMM like" consistency
  - Runs "almost plain Java"
- Directly read / write HDFS
- Targeted at Large Scale Analytics
  - For the *non-expert* programmer

Summary?

- Yes, I am having a blast
  - Yes, I am hitting about 2K LOC / week
  - Stuff's coming together at an amazing rate
- We're well funded and looking for top talent

Team:
Cliff Click
Kevin Normoyle
Matt Fowles
Michal Malohlava
Petr Maj
Srisatish Ambati
Tomas Nykodym

Advisors:
Anand Babu
Doug Lea
Dhruba Borthakur
Jan Vitek
Niall Dalton

Questions?

Cliff Click
cliffc@0xdata.com
http://cliffc.org/blog
http://0xdata.com

H2O

- Replicate Keys based on Hash
  - Every Key is on 3 (or N) nodes, but
  - Which set of 3 nodes varies per key
- Auto-Repair
  - Copy keys from other Nodes to replica
    - Lazily, in spare cycles
  - Copies Value as well: load-balance
    - Same mechanism for hot-data re-balance

DFJ – Repair Computation

- Task “3” split into tasks “1” and “2”
  - And Keys for tasks 1 & 2 replicated
- Node A starts task “1”
- Node B starts task “2”...
  - And dies 1/2-way thru
- Node A finishes “1” (result is replicated)
  - And finds unfinished task "2" on a replica
  - Node A can now do task "2"

DFJ – Repair Computation

- Task “3” split into tasks “1” and “2”
  - And Keys for tasks 1 & 2 replicated
- Node A starts task “1”
- Node B starts task “2”...
  - And dies 1/2-way thru
- Node A finishes “1” (result is replicated)
  - And finds unfinished task "2" on a replica
  - Node A can now do task "2"