High Performance Big-Data Analytics

Kunle Olukotun, Hector Garcia-Molina, Pat Hanrahan, Jure Leskovec

AHPCRC Technical Review Meeting

November 21, 2013
Disk-to-disk map-reduce data processing
Next Gen Big Data Analytics: Improved Decision Making

- Higher performance ⇒ faster decisions
  - Bigger data sizes ⇒ better decisions
  - Low latency big data processing ⇒ interactive decisions
  - Processing on live data streams ⇒ real time decisions

- Higher productivity ⇒ easier decisions
  - More intuitive than map-reduce with key-value pairs
  - Simple programming for complex tasks
    - Data transformation
    - Graph analysis
    - Predictive analysis using machine learning
Next Gen Big Data Analytics Must Embrace Heterogeneous Parallelism

Fine grained parallelism is the only way to get high performance and performance/watt
Heterogeneous Parallel Programming

- Pthreads
- OpenMP
- Multicore
- CUDA
- OpenCL
- GPU
- MPI
- PGAS
- Cluster
- Verilog
- VHDL
- FPGA
Big-Data Analytics Programming Challenge

Data Analytics Application

- Data Prep
- Data Transform
- Network Analysis
- Predictive Analytics

Pthreads
OpenMP

CUDA
OpenCL

MPI
PGAS

Verilog
VHDL

Multicore
GPU
Cluster
FPGA
Big-Data Analytics Programming Challenge

Data Analytics Application

- Data Prep
- Data Transform
- Network Analysis
- Predictive Analytics

Domain Specific Languages

- Pthreads
- OpenMP
- Multicore
- CUDA
- OpenCL
- GPU
- MPI
- PGAS
- Cluster
- Verilog
- VHDL
- FPGA
Domain Specific Languages (DSLs)

- Definition: A language or library with restrictive expressiveness that exploits domain knowledge for productivity and efficiency
- High-level, usually declarative, and deterministic
Benefits of Using DSLs for High Performance

Productivity
• Shield most programmers from the difficulty of parallel programming
• Focus on developing algorithms and applications and not on low level implementation details

Performance
• Match high level domain abstraction to generic parallel execution patterns
• Restrict expressiveness to more easily and fully extract available parallelism
• Use domain knowledge for static/dynamic optimizations

Portability and forward scalability
• DSL & Runtime can be evolved to take advantage of latest hardware features
• Applications remain unchanged
• Allows innovative HW without worrying about application portability
Our Approach: Data Analytics DSLs

Applications
- Data Transform
- Data Wrangling
- Social network Analysis
- Predictive Analytics

Domain Specific Languages (Scala)
- Data Prep: OptiWrangle
- Data Query: OptiQL
- Graph Alg.: OptiGraph
- Machine Learning: OptiML
- Convex Opt.: OptiCVX

Heterogeneous Hardware

New Arch.
Problem 1: abstraction penalty
- Staging: remove abstraction programmatically using partial evaluation

Problem 2: compiler lacks semantic knowledge
- Extend compiler with high-level knowledge
  - E.g. Teach compiler linear algebra

Problem 3: compiler lacks parallelism knowledge
- Extend the compiler with parallelism and locality knowledge

Solving any of the problems alone will not result in high performance
Delite Overview

Key elements

- DSLs embedded in Scala
- IR created using staging
- Domain specific optimization
- General parallelism and locality optimizations
- Mapping to HW targets

Opti{CVX, Graph, ML, QL, Wrangle}

Domain specific analyses & transformations

Generic analyses & transformations

Code generators
- Scala
- C++
- CUDA
- OpenCL
- MPI
- Verilog
Big Data Analytics Systems

Berkeley in memory framework for interactive queries and iterative computations

- Hadoop
- Spark
- Delite

HDFS

Mesos

Processing

Storage management

Cluster resource management
// lineItems: Table[LineItem]
val q = lineItems
    Where(_.l_shipdate <=
        Date("1998-12-01"))
  GroupBy(l => l.l_linestatus).
  Select(g => new Result {
      val linestatus = g.key
      val sumQty = g.Sum(_.l_quantity)
      val sumDiscountedPrice =
        g.Sum(l => l.l_extendedPrice*
              (1.0-l.l_discount))
      val avgPrice =
        g.Average(_.l_extendedPrice)
      val countOrder = g.Count
  })
  OrderBy(_.returnFlag)
  ThenBy(_.lineStatus)

- In-memory data querying
- LINQ, SQL like
- Key operations are query operators on the Table data structure
  - User-defined schema
- Optimizations:
  - Fusion eliminates temporary allocations
  - Eliminate fields not used in query
TPC-H Query 1 on 20 x 4 cores
Using OptiQL for Data Transformations

- Data on disk isn’t always in expected format for analytics DSL
  - Typically pre-processed using a Python script

- OptiQL is also capable of transforming data for the next stage (DSL) in the data analytics pipeline
  - High performance
  - Output can be passed directly to other Delite DSLs
case class TraceRoute(..., trace: String) extends Record
case class Hop(id: Int, ip: String, latency: Float) extends Record
case class Edge(src: String, dst: String) extends Record

val data = Table.fromFile[TraceRoute]("/path/to/file", colSep="",")
val allEdges = data SelectMany { r =>
  //parse trace field string into a Table
  val hops = Table.fromString[Hop](r.trace, rowSep="\|\", colSep="":"")
  Table.range(0, hops.size-1) Select { i =>
    Edge(hops(i).ip, hops(i+1).ip) //edge between this ip and next ip
  }
}
//eliminate duplicate edges and edges that appear infrequently
val edges = allEdges GroupBy(e => e) Select(g => new Record {
  val edge = g.key
  val count = g.Count
}) Where(_.count > threshhold) Select(_.edge)
Graph Transformation Performance

20GB Akamai Traceroute “es_cl” dataset

Execution Time (sec)

- Scala Lib: 1 hr
- Delite 1: 15m
- Delite 4: 5m
- Delite 16: 2m
- Delite 64: 1m
OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning, ICML 2011

- Designed for iterative statistical inference
  - e.g. SVMs, logistic regression, neural networks, etc.
  - Dense/sparse vectors and matrices, message-passing graphs, training/test sets

- Mostly functional
  - Data manipulation with classic functional operators (map, filter)
  - ML-specific ones (sum, vector constructor, untilconverged)
  - Math with MATLAB-like syntax (a*b, chol(..), exp(..))
  - Mutation is explicit (.mutable) and last resort

- Runs anywhere
  - Single source to multicore CPUs, GPUs, and clusters (via Delite)
until converged (centroids, tol) {
  centroid =>

  newCentroids
}

OptiML: $k$-means Clustering
until converged(centroids, tol){
  centroid =>
    // assign each sample to the closest centroid
    val clusters = samples.groupRowsBy { sample =>
      // calculate distances to current centroids
      val allDistances = centroids mapRows { centroid =>
        dist(sample, centroid)
      }
      allDistances.minIndex
    }

    // move each cluster centroid to the mean of the points assigned to it
    val newCentroids = clusters.map(e => e.sum / e.length)
    newCentroids
}

- No explicit map-reduce
- No key value pairs
- Efficient multicore, GPU and cluster implementations
Markov State Models (MSMs)
MSMs are a powerful means of modeling the structure and dynamics of molecular systems, like proteins.
Machine Learning on 20 x 4 cores

- **k-means**
  - 1.7 GB: Spark 75, Delite 275
  - 17G: Spark 13, Delite 200

- **Logistic Regression**
  - 3.4GB: Spark 100, Delite 450
  - 17G: Spark 75, Delite 300
Machine Learning
4 x 12 cores and 4 x GPU

- Spark
- Delite CPU
- Delite GPU

Speedup (over Spark)

- k-means
- Logistic Regression
Green-Marl: A DSL for Graph Analysis

- Classic graphs; New “Big Data” applications
  - Artificial Intelligence, Computational Biology, ...
  - SNS apps: Linkedin, Facebook, ...

Example > Movie Database

- "Is he a central figure in the movie network? How much?"
- "Are these actors work together more frequently than others?"
- "What would be the avg. hop-distance between any two (Australian) actors?"
**Betweenness Centrality**

- A measure that tells how ‘central’ a node is in the graph
- Used in social network analysis
- Definition
  - How many shortest paths are there between any two nodes going through this node.

\[
C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}
\]
betweenness centrality

Example: Betweenness Centrality

[Brandes 2001]

_init BC and Outer-loop

init BC and Outer-loop

(s)

Looks complex

Compute delta from children

Accumulate delta into BC

if \( w \neq s \) then \( C_B[w] := C_B[w] + \delta[w] \);

Compute sigma from parents

Parallel Assignment

Parallel Iteration

Parallel BFS

Reduction

Procedure comp_BC(G: Graph, BC: Node_Property<

\text{float}>(G))

\begin{aligned}
\text{G.BC} &= 0; \quad \text{// Initialize} \\
\text{foreach } (s: G.Nodes) \{ \\
\quad &\text{// temporary values per Node} \\
\quad \text{Node_Property<\text{float}}(G) \text{ sigma}; \\
\quad \text{Node_Property<\text{float}}(G) \text{ delta}; \\
\quad \text{G.sigma} = 0; \quad \text{// Initialize} \\
\quad \text{G.delta} = 0; \\
\quad \text{s.sigma} = 1; \\
\quad \text{// BFS order iteration from s} \\
\quad \text{InBFS(v: G.Nodes From s)} \{ \\
\quad \quad v.sigma = \quad \text{// Summing over BFS parents} \\
\quad \quad \text{Sum } (w:v.UpNbrs) \{ w.sigma \}; \\
\quad \} \\
\quad \text{// Reverse-BFS order iteration to s} \\
\quad \text{InRBFS(v:G.Nodes To s)(v!=s)} \{ \\
\quad \quad v.delta = \quad \text{// Summing over BFS children} \\
\quad \quad \text{Sum } (w:v.DownNbrs) \{ \\
\quad \quad \quad v.sigma / w.sigma * (1 + w.delta) \} ; \\
\quad \} \\
\quad v.BC += v.delta @ s; \quad \text{// accumulate BC} \\
\} 
\end{aligned}
A DSL for large-scale graph analysis based on Green-Marl
  - A DSL for Real-world Graph Analysis
  - Green-Marl: A DSL for Easy and Efficient Graph Analysis (Hong et. al.), ASPLOS ’12

Functional DSL
  - No mutation

Data structures
  - Graph (directed, undirected), node, edge,
  - Set of nodes, edges, neighbors, ...

Graph iteration
  - Normal parallel iteration, Breadth-first iteration, Topological Order, ...

Parallel reductions but no deferred assignment (Bulk synchronous consistency)
1. val bc = Sum(G.Nodes){ s => 
2.   val sigma = inBfs(s) { (v,prev_sigma) => if(v==s) 1 
3.     else Sum(v.UpNbrs){w=> prev_sigma(w)} 
4.   } 
5. val delta = inRevBfs(s){ (v,prev_delta) => 
6.   Sum(v.DownNbrs){w=> 
7.     sigma(v)/sigma(w)*(1+prev_delta(w)) 
8.   } 
9. } 
10. delta 
11. }
Real Applications Span Multiple Domains

Data cleansing and querying

Analytics

Visualization

Image sources: ThinkStock
http://technaverbascripta.wordpress.com/2012/10/19/text-network-analysis-2-meaning-circulation-in-lolita/
http://bl.ocks.org/mbostock/3943967
DSL Composition

- DSLs that require restricted semantics to enable domain-specific transformations

- We use a three step process:
  - Independently stage each DSL block using scopes
  - Lower each IR to the common IR (Delite IR) and combine together
  - Optimize and code generate the final composed IR

- Scopes
  - A Scala construct for type-safe isolated lexical scoping
Data analytic application with a Twitter dataset

Uses OptiQL, OptiGraph, and OptiML
Conclusions

- DSLs are the key to next generation big data analytics
  - High Productivity: higher level abstractions
  - High performance: fine-grained parallelism

- Sophisticated compilers needed to make sense of high-level, domain-specific abstractions

- Performance advantage of compiling DSLs is substantial