Scalable Linear Algebra Programming for Big Data Analysis

Leonidas Fegaras

University of Texas at Arlington

Motivation

- Arrays are very important for scientific computing and ML
- How to scale array apps out on a Big Data system?

  partition large arrays into blocks: *distributed block arrays*
- Some linear algebra libraries support them (TensorFlow, MLlib, ScaLAPACK, ...), but ...
  - no ad-hoc array programming
  - limited choices for array storage
  - hard to extend with new operations and storage structures

Goal

Develop a query language for array computations that

- is declarative
- allows ad-hoc array programming
- captures a large class of array computations

Implement a compiler/optimizer that

- uses customized array storage structures
- translates these array computations to efficient distributed code

Data Model: Abstract Arrays

- Abstract arrays are represented in a coordinate format
- An array is an association list that uniquely maps array indices to values
- An array element $M_{ijk\ldots}$ is represented as an index-value pair:

$$\Big(\underbrace{(i, j, k, \ldots)}_{\text{array indices}}, \underbrace{v}_{\text{array value}} \Big)$$

Array Comprehensions: A Calculus for Linear Algebra

- Can access multiple arrays by traversing their elements one-by-one
- Can construct a new array in one shot by mapping array indices to values
- Matrix column summation $\sum_j M_{ij}$:

$$\overbrace{\Big[\ \big(\ i,\ \underbrace{+/m}_{\text{aggregation}}\ \big)}^{\text{new vector element}}\ \Big|\ \overbrace{\underbrace{((i,j),m)}_{\text{pattern}} \leftarrow \underbrace{M}_{\text{matrix}}}^{\text{generator}},\ \textbf{group by}\ i\ \Big]$$

- Array elements can be constructed in an arbitrary order

  Matrix row rotation:

  [(((i+1)%n, j), m) **|** ((i,j),m) ← M **]**

- They are just monoid comprehensions on association lists

Array Comprehensions (cont.)

Can capture many array computations:

- Matrix addition $M_{ij} + N_{ij}$:

  [((i,j),m+n) **|** ((i,j),m) ← M, ((ii,jj),n) ← N, ii == i, jj == j **]**

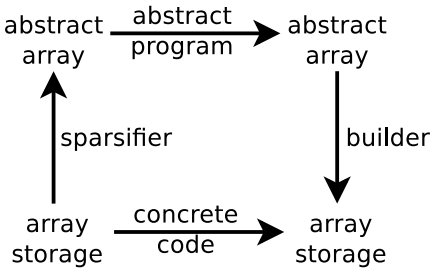
- Matrix multiplication $\sum_{ij} M_{ik} * N_{kj}$:

  [((i,j),+/v) **|** ((i,k),m) ← M, ((kk,j),n) ← N, kk == k, **let** v = m∗n, **group by** (i,j) **]**

- array slicing, concatenation, smoothing, ...

Separation of Specification from Implementation

- Abstract arrays are mapped to customized storage structures based on user-defined type mappings



- The sparsifier and builder are unfolded and fused with the abstract program

Storing a Matrix as a Distributed Block Array

- Storage is RDD[($\underbrace{\text{(Int,Int)}}_{\text{tile coordinates}}$, $\underbrace{\text{Matrix[Double]}}_{\text{a tile}}$)]
- The sparsifier converts the block array M to an abstract array:

$$\textbf{[}\ (\ (\ ii{*}N{+}i,\ jj{*}N{+}j\ ),\ A(i,j)\ )\ \textbf{|}\ (\ \underbrace{(ii,jj)}_{\text{coordinates}}\ ,\underbrace{A}_{\text{tile}}\ ) \leftarrow M,\ i \leftarrow 0\ \textbf{until}\ N,\ j \leftarrow 0\ \textbf{until}\ N\ \textbf{]}$$

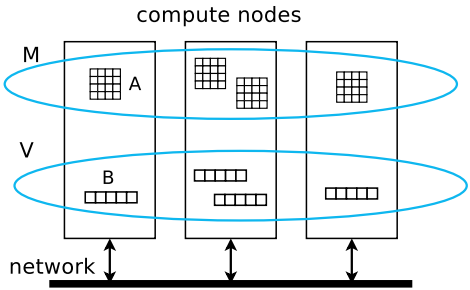
- The builder tiled(L) constructs a block matrix from the list L:

$$\underbrace{\text{rdd}}_{\text{RDD builder}}\ \textbf{[}\ (\ (i/N,\ j/N),\ \underbrace{\text{matrix(w)}}_{\text{matrix builder}}\ )\ \textbf{|}\ ((i,j),v) \leftarrow L,\ \textbf{let}\ w = \underbrace{(\ (\ i\%N,\ j\%N\ ),\ v\ )}_{\text{tile element}},\ \textbf{group by}\ (\ i/N,\ j/N\ )\ \textbf{]}$$

Code generation for $V_i = \sum_j M_{ij}$

Array Comprehension

```
V = [ (i,+/m) | ((i,j),m) ← M, group by i ]
```



Spark Code

$B_i = \sum_j A_{ij}$ ⟼

```
V = M.map { case ((I,J),A)
          ⇒ { val B = Array.fill(N)(0.0);
              for { i ← (0 until N).par;
                    j ← 0 until N }
                B(i) += A(i,j);
              (I,B) } }
    .reduceByKey { case (X,Y)
          ⇒ { val Z = Array.fill(N)(0.0);
              for { i ← (0 until N).par }
                Z(i) = X(i) + Y(i);
              Z } }
```

$Z_i = X_i + Y_i$ ⟼

M: RDD[((Int,Int),Matrix[Double])]
V: RDD[(Int,Vector[Double])]

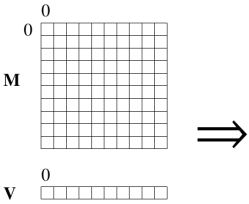
SAC: Scalable Array Comprehensions

- Implemented on top of Spark using Scala's compile-time reflection
- Translates array comprehensions to efficient Spark code on block arrays
- Used as a drop-in back-end replacement for DIABLO [Fegaras&Noor@VLDB'20]
  - translates Java-like loop-based programs to Spark programs on block arrays
- Up to 6 times faster than MLlib for matrix multiplication
- Up to 3 times faster than MLlib for matrix factorization
- Current work: extend SAC to work on compressed sparse column block arrays

Code generation for $V_i = \sum_j M_{ij}$ using DIABLO

Imperative Code

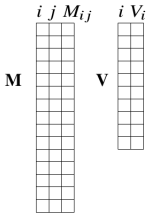
```
for ( int i = 0; i < n; i++ ) {
    V[i] = 0.0;
    for ( int j = 0; j < m; j++ )
        V[i] += M[i][j];  }
```



M: Matrix[Double]
V: Vector[Double]

Array Comprehension

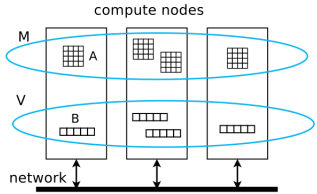
```
V = [ (i,+/m) | ((i,j),m) ← M,
                group by i ]
```



M: List[((Int,Int),Double)]
V: List[(Int,Double)]

Spark Code

```
V = M.map { case ((I,J),A)
                ⇒ { val B = Array.fill(N)(0.0);
                    for { i ← (0 until N).par;
                          j ← 0 until N }
                        B(i) += A(i,j);
                    (I,B)  }  }
    .reduceByKey( addVectors )
```



M: RDD[((Int,Int),Matrix[Double])]
V: RDD[(Int,Vector[Double])]