Translation of Array-Based Loops to Distributed Data-Parallel Programs

Leonidas Fegaras and Md Hasanuzzaman Noor

University of Texas at Arlington

Motivation

Arrays are important for scientific computing and ML

Most of these programs are written in an imperative programming language

- they access elements randomly (one at a time) using loops and array indexing
- they may call special array operations from libraries (R, NumPy, MATLAB, LAPACK)
- they can be accelerated with specialized hardware, such as GPUs, TPUs, SIMD

They utilize resources better when *scaled up* to a single high-end computer,
rather than *scaled out* to multiple computers

To process larger amounts of data, these sequential programs
must be rewritten to work on Big Data systems

Goal

Translate loop-based programs to efficient distributed data-parallel programs

Matrix Multiplication

```
for i = 0, n−1 do
    for j = 0, n−1 do {
        R[i,j] := 0;
        for k = 0, m−1 do
            R[i,j] += M[i,k]*N[k,j]
    }
```

Spark RDD Code

```
R = M.map{ case (i,j,m) => (j,(i,m)) }
      .join( N.map{ case (i,j,n) => (i,(j,n)) } )
      .map{ case (k,((i,m),(j,n))) => (i,j,m*n) }
      .reduceByKey(_+_)
```

Goal

Translate loop-based programs to efficient distributed data-parallel programs

Matrix Multiplication

```
for i = 0, n−1 do
    for j = 0, n−1 do {
        R[i,j] := 0;
        for k = 0, m−1 do
            R[i,j] += M[i,k]*N[k,j]
    }
```

Spark RDD Code

```
R = M.map{ case (i,j,m) => (j,(i,m)) }
      .join( N.map{ case (i,j,n) => (i,(j,n)) } )
      .map{ case (k,((i,m),(j,n))) => (i,j,m*n) }
      .reduceByKey(_+_)
```

Why bother?

- allows data analysts to parallelize their programs automatically without having to learn Big Data programming
- gives an alternative and more conventional way of developing distributed applications

Related work

- Automated parallelization in HPC – applied to shared-memory architectures
  - *Recurrence*: when an array is updated in one loop step and used in the next steps
  - Recurrences complicate parallelization
  - DOALL parallelizes loops without recurrences
- Many systems support distributed arrays:

  TensorFlow, MLlib, SciDB, TileDB, ArrayStore, SystemML, SciHadoop, SciMATE, SimSQL
- Translating imperative code to SQL [Emani et al, Sigmod'16], [Luo et al, ICDE'17]
- Closest related work:
  - MOLD [Radoi et al, OOPSLA'14] is template-based
  - Casper [Ahmad & Cheung, Sigmod'18] uses a program synthesizer

Our Approach

DIABLO: a Data-Intensive Array-Based Loop Optimizer

A compositional approach:

- Use a small set of meaning preserving rules to transform programs piece-wise:
  - every syntactic element (an AST node) in a loop-based program is translated to a query (a comprehension) that calculates the cumulative effects of this element across the loops
  - the resulting layered comprehensions are fused using standard normalization transformations
  - they are also simplified using general optimizations
- No need to search for program templates to match (as in MOLD)
  or use a program synthesizer (as in Casper)

Sparse Array Representation

- Arrays are represented as distributed collections of tuples
- Each tuple contains the indexes and value of a single array element

  Matrix: RDD[(Long, Long, Double)]
- Equivalent to the relational schema Matrix (I, J, V)
- Can be extended to support tiled arrays (this is work in progress)

  just one more layer to fuse

Matrix Multiplication: $R_{ij} = \sum_k M_{ik} * N_{kj}$

```
for i = 0, n−1 do
    for j = 0, n−1 do {
        R[i,j] := 0;
        for k = 0, m−1 do
            R[i,j] += M[i,k]*N[k,j]    }
```

Matrix Multiplication: $R_{ij} = \sum_k M_{ik} * N_{kj}$

```
for i = 0, n−1 do
    for j = 0, n−1 do {
        R[i,j] := 0;
        for k = 0, m−1 do
            R[i,j] += M[i,k]*N[k,j]    }
```

The R values can be calculated using SQL:

```
insert into R
  select M.I, N.J, sum(M.V*N.V) as V
  from M join N on M.J=N.I
  group by M.I, N.J
```

Matrix Multiplication: $R_{ij} = \sum_k M_{ik} * N_{kj}$

```
for i = 0, n−1 do
    for j = 0, n−1 do {
        R[i,j] := 0;
        for k = 0, m−1 do
            R[i,j] += M[i,k]*N[k,j]    }
```

The R values can be calculated using SQL:

```
insert into R
  select M.I, N.J, sum(M.V*N.V) as V
  from M join N on M.J=N.I
  group by M.I, N.J
```

DIABLO actually generates a comprehension:

$$R := \{\, (i,j,+/v) \mid (i,k,m) \leftarrow M,\ (k',j,n) \leftarrow N,\ k' = k,\ \textbf{let}\ v = m * n,\ \textbf{group by}\ (i,j) \,\}$$

Matrix Multiplication: $R_{ij} = \sum_k M_{ik} * N_{kj}$

$$R := \{\, (i, j, +/v) \mid (i, k, m) \leftarrow M,\ (k', j, n) \leftarrow N,\ k' = k,\ \textbf{let}\ v = m * n,\ \textbf{group by}\ (i, j) \,\}$$

Monolithic construction: the entire matrix R is calculated in one shot

Updates to R are performed in bulk by

- grouping the values $v = m * n$ across the loops by the result indices and
- summing up these values for each group

Easy to parallelize:
In Spark: it's a join between M and N followed by a reduceByKey

Restrictions on Parallelization

Simple syntactic restrictions:

- We cannot read and update an array in the same loop (DOALL recurrence restriction)

except ...

if it is in an incremental update of the form $V[e_1] \mathrel{+}= e_2$, for some commutative operation $+$ and some terms e_1 and e_2

access to V in $V[e_1] \mathrel{+}= e_2$ counts as a write to V, but not as a read from V

- The destination $V[e_1]$ of a regular update $V[e_1] := e_2$ must be unique at each loop step

the destination index e_1 must be affine (a linear combination of loop indexes)

Contributions

- DIABLO is built on top of DIQL
  - a query optimization framework for Big Data that translates queries to Java byte code at compile-time
- The input programs are written in a small loop-based array language
- DIABLO translates these programs to comprehensions and then compiles them to Scala code on Spark RDD API
- We have provided compositional transformations and correctness proof
- We have evaluated performance relative to hand-written Spark programs and Casper on a variety of linear algebra and ML programs
  - DIABLO programs have performance comparable to hand-written programs

Ongoing Work

- We are replacing the DIQL back-end with the Spark SQL engine (the Catalyst optimizer)
- We are working on using packed arrays (such as tiled matrices) instead of sparse arrays

The Transformation Process

DIABLO translates a loop-based program in pieces, in a bottom-up fashion:

$$M[i,j] \quad \longrightarrow \quad \{\, m \mid (I,J,m) \leftarrow M,\ I = i,\ J = j \,\}$$

Every operation is lifted to a comprehension:

$$A * B \quad \longrightarrow \quad \{\, a * b \mid a \leftarrow A,\ b \leftarrow B \,\}$$

which is normalized to:

$$\begin{aligned}
&M[i,k] * N[k,j] \\
&= \{\, a * b \mid a \leftarrow \{\, m \mid (I,J,m) \leftarrow M,\ I = i,\ J = k \,\}, \\
&\qquad\qquad b \leftarrow \{\, n \mid (I,J,n) \leftarrow N,\ I = k,\ J = j \,\} \,\} \\
&= \{\, m * n \mid (I,J,m) \leftarrow M,\ I = i,\ J = k, \\
&\qquad\qquad (I',J',n) \leftarrow N,\ I' = k,\ J' = j \,\},
\end{aligned}$$

which is a join between M and N

The Transformation Process (cont.)

for $i = 0,\ n-1$ do
  for $j = 0,\ n-1$ do
    for $k = 0,\ m-1$ do
      $R[i,j] \mathrel{+}= M[i,k] * N[k,j]$

$\longrightarrow$

$$R := \{\, (i, j, +/v) \mid i \leftarrow \text{range}(0, n-1),\ j \leftarrow \text{range}(0, n-1),\ k \leftarrow \text{range}(0, m-1),\ v \leftarrow M[i,k] * N[k,j],\ \textbf{group by}\ (i,j) \,\}$$

then replace $M[i,k] * N[k,j]$