Matrix Multiplication in MapReduce

Overview

Matrix Multiplication:

- **is extremely important in computing.** It is critical to a large number of tasks from graphics and cryptography to graph algorithms and machine learning.
- **it computationally intensive.** A naïve sequential matrix multiplication algorithm has complexity of $O(n^3)$. Algorithms with lower computational complexity exist, but they are not always faster in practice.
- **is a good candidate for distributed processing.** Every matrix cell is computed using a separate, independent from other cells computation. The computation consumes $O(n)$ input (one matrix row and one matrix column).

As such, matrix multiplication is a good candidate for being expressed as a MapReduce computation.

Matrix Multiplication

For the sake of completeness we briefly define matrix multiplication operation here.

Let $A = (a_{ij}$ be an $n \times m$ matrix, and $B = (b_{jk})$ be an $m \times s$ matrix:

$$A = \begin{pmatrix} a_{11} & \dots & a_{1m} \\ & \dots & \\ a_{n1} & \dots & a_{nm} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & \dots & a_{1s} \\ & \dots & \\ b_{m1} & \dots & a_{ms} \end{pmatrix}$$

An $n \times s$ matrix $C = (c_{ik})$:

$$C = \begin{pmatrix} c_{11} & \dots & a_{1s} \\ & \dots & \\ c_{n1} & \dots & a_{ns} \end{pmatrix}$$

is the **product** of A and B:

$$AB = C$$

if

$$c_{ik} = \sum_{j=1}^{m} a_{ij} \cdot b_{jk}$$

Basic Solution

The last formula defines $n \cdot s$ computations, each independent of others.

Our goal is to create $n \cdot s$ reducers: **one for each pair** (i, k) of rows from matrix A and columns from matrix B.

A pair of indexes (i, k), $i \in [1, \dots, n], k \in [1, \dots, s]$ can serve as the key for the reducer function.

Straightforward: the input value for the reducer function can be a list consisting of two vectors: $\mathbf{a_i} = (a_{i1}, \dots, a_{im})$ and $\mathbf{b_k} = (b_{1k}, \dots, b_{mk})$.

reduce(). Based on this, a simple `reduce()` function for computing a single cell of the matrix can look as follows (in pseudocode):

```
function reduce(key, <list> value) {

   vector1 := value[0];     // extract one vector
   vector2 := value[1];     // extract the second vector
   m := size(vector1);

   cell := 0;
   for i=0 to m-1 do        // compute dot product of the vectors
     cell := cell+ vector1[i]*vector2[i];
   end for

   emit(key, cell);         // output the result indexed by the key
}
```

To create a matching `map()` function, we need to agree on the format of the input. Here, we select the simplest possible input for the mapper. In section below, we discuss variants of the input.

Let us assume that the input matrices come in two separate files, which have the following formats:

1. The file containing matrix A stores this matrix in the row-wide form, i.e., each line of the input file corresponds *to one row of matrix* A.

2. The file containing matrix B stores this matrix in the column-wide format, i.e., each line of the input file corresponds to *one column of matrix* B. In other words, this file actually stores in row-wide format matrix B^T.

3. Each row starts with a pair of numbers representing the row of the matrix that is being considered. (For input file for matrix A this is a row of A, for input file for matrix B this is a row of B^T, i.e., a column of B). We assume that `map()` functions treats that first value as the key.

Under these assumptions, we can use a simple *reduce-side join* pattern and create two independent `map()` functions to feed input into the `reduce()` function above.

```
function mapA(key, value) {   // mapper for managing matrix A input file

    Vector v = string2vector(value);  // convert input value from string to vector
    for j = 1 to m do                 // distribute the vector everywhere it is needed
      newKey = (key, j);
      emit(newKey, v);
    end for
}

function mapB(key, value) {   // mapper for managing matrix B input file

    Vector v = string2vector(value);  // convert input value from string to vector
    for i = 1 to s do                 // distribute the vector everywhere it is needed
       newKey = (i,key);
       emit(key, v);
    end for
}
```

Variants

The basic MapReduce solution assumes very specific input format: two matrices in separate files, second matrix transposed.

Below, we show how to change our matrix multiplication procedure for two (actually three) other common input formats.

Alternative Input Format 1: Two matrices in same file. Often, both matrices come in the same input file. In the most simple case, the input file first contains the rows of matrix A, followed by the rows of matrix B^T.

If both matrices are stored in the same file, the rows of A and the rows of B^T must be diffirentiated in the input file.

This means that the key to each row is a pair $(Origin, Index)$ where $Origin$ is a matrix label (`A` or `B`), and $Index$ is an integer row number.

Example. Consider for example the following two matrices:

$$A = \begin{pmatrix} 2 & 5 & 8 \\ 3 & 9 & 1 \\ 6 & 4 & 10 \end{pmatrix}$$

$$B = \begin{pmatrix} 5 & 8 \\ 2 & 1 \\ 7 & 9 \end{pmatrix}$$

The input file representing these two matrices can look as follows[1]:

```
A 1,2 5 8
A 2,3 9 1
A 3,6 4 10
B 1,5 2 7
B 2,8 1 9
```

[1] Without loss of generality we use commas to separate keys from values in each row, and we use spaces to separate individual matrix cell values from each other.

For input like this, we can keep the same `reduce()` function, but we do need to merge the two `map()` functions into one:

```
function map(key, value){

  v := String2Vector(value);
  matrixId := key.matrixId;

  if matrixId == "A" then
     row := key.rowId;
     for i = 1 to m do
        newKey = (row, i);
        emit(newKey, v);
     end for
  else
    if matrixId == "B" then
       col := key.rowId;
       for j = 1 to m do
         newkey=(j, col);
         emit(newKey, v);
       end for
   end if
}
```

Alternative Input Format 2: Matrix B is NOT transposed. The first truly challenging input format arises in situations when the second input matrix, B, is supplied directly, NOT in transposed format. For example, the input file supplying both matrices A and B as-is can look as follows:

```
A 1,2 5 8
A 2,3 9 1
A 3,6 4 10
B 1,5 8
B 2,2 1
B 3,7 9
```

Note: In this case, the mapper must do different things for A and B matrix input lines. Specifically, *we no longer can emit the full row of matrix B^T (i.e., full column of B) to the reducer.*

This means that we must change both the `map()` and the `reduce()` functions.

```
function map(key, value) {


  v := String2Vector(value);
  matrixId := key.matrixId;

  if matrixId == "A" then   // for matrix A we do the same thing
     row := key.rowId;
     for i = 1 to m do
        newKey = (row, i);
        newVal = ("A", v);  // except we add "A" to emitted value
        emit(newKey, newVal);
     end for
  else
    if matrixId == "B" then   // for matrix B we emit one value at a time
       row := key.rowId;
       for k = 1 to s do
```

```
            newVal := v[k-1];  // extract the next element row v

            for j = 1 to m do  // emit this element where it is needed
              newkey :=(row, k);
              newValue := ("B", j, newVal); // we emit the value, its position in vector
              emit(newKey, newValue);       // and matrix flag
        end for
    end if
 }
```

reduce() function must now untangle its input properly.

```
function reduce(key, <list> values) {

  vectorB := new vector(m, 0); // initialize vectorB to be an m-tuple of zeros

  for v in values do
     if v.matrixId == "A" then  // for matrix A simply extract the vector
        vectorA := v.vector;
     else
       if v.matrixId == "A" then   // for matrix B extract one element at a time
          index := v.index;
          val := v.val;
          vectorB[index-1] := val;
       end if
  end for

  // now, compute the dot product of two vectors

  sum := 0;
  for i=1 to m do
    sum := sum + vectorA[i-1]*vectorB[i-1];
  end for

  emit(key, sum);
}
```

Alternative input format 3: Sparse Matrices. Large matrices are often sparse and the dense row-wide formats for representing them in input are bulky. For these cases, a format that presents one value per line is used. The input file has the format:

```
MatrixId  Row  Column, Value
```

For example, the matrices from the example above would be represented as

```
A 1 1, 2
A 1 2, 5
A 1 3, 8
A 2 1, 3
A 2 2, 9
A 2 3, 1
A 3 1, 6
A 3 2, 4
A 3 3, 10
B 1 1, 5
B 1 2, 8
B 2 1, 2
B 2 2, 1
B 3 1, 7
B 3 2, 9
```

(note: B is this case is not transposed)

In this case, `map()` must emit one value at a time for both A and B values. This value must be keyed by the row and column of the result matrix, but must also contain the matrix of origin information and location in the vector.

```
function map(key, value) {

 matrixId := key.matrixId;  // deconstruct key
 row := key.row;
 col := key.col;

 if matrixId == "A" then
   for i = 1 to s do
      newKey := (row,i);
      newValue := ("A", col, value);
      emit(newKey, newValue);
   end for
 else
    if matrixId == "B" then
      for i = 1 to n do
        newKey := (i,col);
        newValue := ("B", row, value);
        emit(newKey, newValue);
   end for
}
```

The matching `reduce()` function reconstructs **both** vectors from A and B and computes the dot product.

```
function reduce(key, <list> values) {

  vectorA := new vector(m, 0); // initialize vectorA to be an m-tuple of zeros
  vectorB := new vector(m, 0); // initialize vectorB to be an m-tuple of zeros

  for v in values do
     if v.matrixId == "A" then
          index := v.index;
          val := v.val;
          vectorA[index-1] := val;
     else
       if v.matrixId == "A" then
          index := v.index;
          val := v.val;
          vectorB[index-1] := val;
       end if
  end for

  // now, compute the dot product of two vectors

  sum := 0;
  for i=1 to m do
    sum := sum + vectorA[i-1]*vectorB[i-1];
  end for

  emit(key, sum);
}
```