Matrix Layout Problem in MapReduce

Yunfei Shen and Luyao Li

Electrical & Computer Engineering, Duke University

Abstract

MapReduce is an excellent platform for massively parallel data processing. It works well for many relational-style queries consisting of operations like group-by, aggregation, join, etc. However, it is not optimized for matrix-form data and linear algebra kernels which are building blocks of many statistical/numerical methods used in deep analysis. [1] Among different matrix applications, matrix multiplication is a critical and fundamental operation.

In this project, the data access patterns in matrix files is studied and proposed a new concentric data layout solution to facilitate matrix data access and analysis in MapReduce framework. Concentric data is first written to Hadoop Distributed File System (HDFS) in the form of the dimensional parameters and value of each element in matrix. As for the matrix method, a block-based matrix multiplication method is proposed and studied. Three different plans to distribute sub-matrix blocks are designed and simulations are conducted to compare the performance of these three plans using the large data sets already transferred into HDFS. Thus, an optimal matrix multiplication method is found and the capability of the matrix layout is proved.

1 Introduction

Before starting to solve this problem, note that there are more than one method to do matrix multiplication. Suppose there are two input matrices A and B, with sizes *s* x *t* and *t* x *u* (s, t and u are very large), and want to compute C = AB.

Suppose A=$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, B=$\begin{bmatrix} e & f \\ g & h \end{bmatrix}$,

C=AB=$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+df & cf+dh \end{bmatrix}$

The following is a non-exhaustive list of such algorithms. There are other algorithms better suited for distributed computing: [1]

1. View each matrix as a table and each matrix element as a tuple (i; j; v) where i; j denote the row and column index of the element, and v denotes its actual value. With such a representation, C can be computed by the following relational query:
   SELECT A.I,B.J,SUM(A.V*B.V) FROM A,B
   WHERE A.J=B.I GROUP BY A.I,B.J
2. Take row i from A and column j from B; the dot product of them is Cij. Vary i from 1 to p and j from 1 to q, and we get C.

```
for i=1...p {
    Crow <- matrix(0,1,q)
    read Arow <- A[i,:] from disk
    for j=1...q {
        read Bcol <- B[:,j] from disk
        Crow[1,j] <- Arow _ Bcol
    }
    write Crow as C[i,:] to disk
}
```

3. Divide A into s _ t submatrices (blocks) and B into t _ u ones. Take the i-th row of blocks from A and the j-th column of blocks from B. These two multiplied together give the i; j-th block of C.

```
for i=1...(p/s)
    for j=1...(q/u) {
        Csub <- matrix(0,s,u)
        for k=1...(r/t) {
            read Asub <- A[(i*s-s+1):(i*s),(k*t-t+1):(k*t)] from disk
            read Bsub <- B[(k*t-t+1):(k*t),(j*u-u+1):(j*u)] from disk
            Csub <- Csub + Asub Bsub
        }
        write Csub as C[(i*s-s+1):(i*s),(j*u-u+1):(j*u)] to disk
    }
```

The blocking factor parameters s; t; u actually determine the I/O cost ofthe algorithm. It has been shown that the optimal I/O cost in a non-distributed setting is achieved when s = t = u [2].

In a non-distributed setting, these algorithms have different requirements on the layout of data in order to achieve optimal I/O performance. For example, a block-based layout may work much better for the third algorithm than a simple row-major layout does (more sequential accesses). [1]

2 Matrix Layout

In this project, the matrix multiplication is executed by blocking multiplication. In blocking, matrix multiplication of the original arrays is transformed into matrix multiplication of blocks. For example,

C_block(1,1)=A_block(1,1)*B_block(1,1) + A_block(1,2)*B_block(2,1)

This is a well-known Cache Blocking technique for efficient memory accesses.

Then, the Memory Blocking technique is continued to be considered for efficient disk accesses and parallelism on the Hadoop. Contrary to store the whole matrix into Hadoop file system, the dimensional information and value of each element in the matrix is stored. That is, each line in the file is dimension (row, column) and value. In this way, when there is a extreme large matrix which exceed a block size (need to be split when parse into Mapper), because of the dimensional information (row and column number for each element) referring to each element (cell) in the matrix, it will be easy to distribute the discrete elements (when reading into mapper from different splits) into the exact block (sub-matrix) for next step.

2.1 Matrix Layout Format

As mentioned in above paragraph, the matrix format is illustrated as dimension and value. Since in MapReduce, the default key is the offset of line number. Here, the SequenceFileInput and SequenceFileOutput format in MapReduce is applied to rewrite the default key value to the desired key, which is Key1 in this project. Key1 has two elements <index1, index2>, representing the dimensional information of each cell in matrix. The value referred by Key1 is Value in this project, which is the concrete value of this cell. In this way, a matrix is first transferred as <Key1, Value> pairs stored in HDFS for further use. Another advantage is that if one matrix will be used more than one time, e.g. in calculating AB+AC, once stored in HDFS, extra transfer is not required, thus saving the I/O cost.

2.2 Example

Consider a simple example, how to transfer a matrix A= $\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$

Suppose this matrix is written to File A, then in A file, the format is like following:

0, 0	1
0, 1	2
0, 2	3
1, 0	4
1, 1	5
1, 2	6

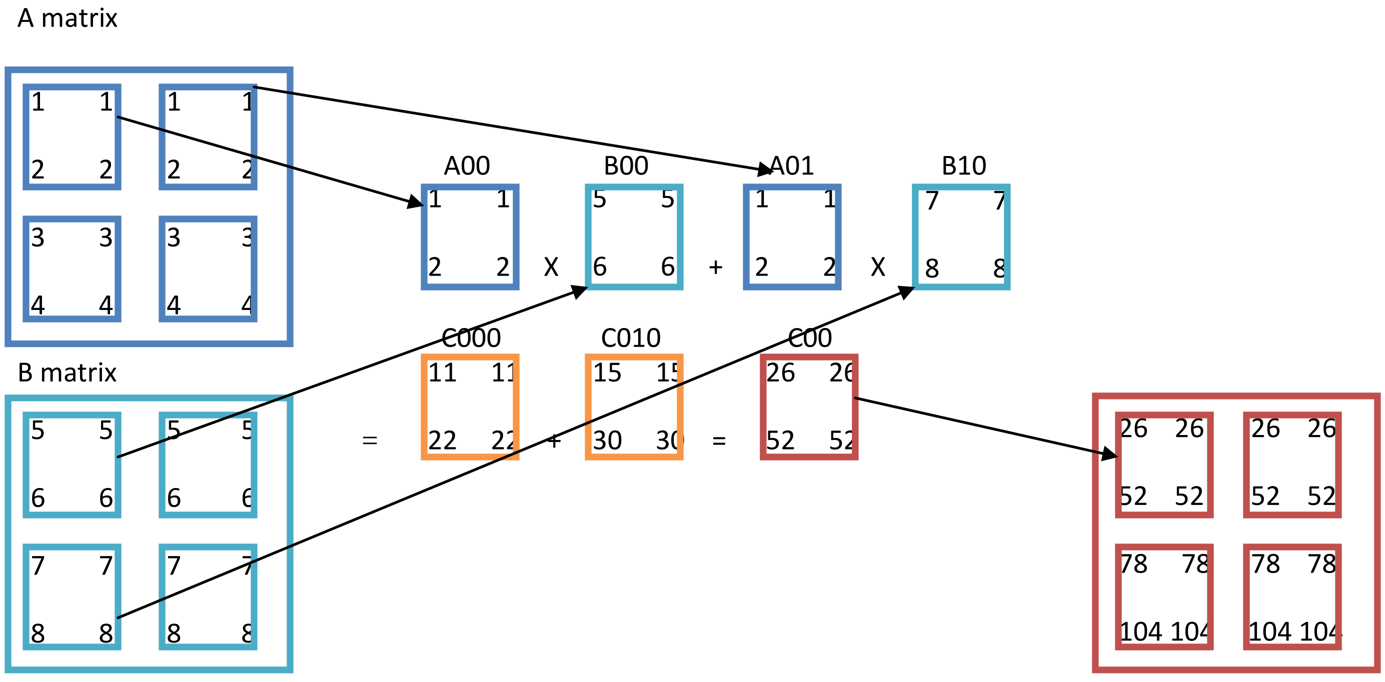
2, 0	7
2, 1	8
2, 2	9

The java code to use SequenceFile Writer to write the matrix into file system is given as follow:

```
public static void write(int[][] matrix, int rowDim, int colDim, String
pathStr, Configuration conf, FileSystem fs) throws Exception{
	Path path = new Path(pathStr);
	SequenceFile.Writer writer = SequenceFile.createWriter(fs, conf, path,
	Key1.class, IntWritable.class, SequenceFile.CompressionType.NONE);
	Key1 key1 = new Key1();
	IntWritable el = new IntWritable();
	for (int i = 0; i < rowDim; i++) {
		for (int j = 0; j < colDim; j++) {
			int v = matrix[i][j];
			if (v != 0) {
				key1.index1 = i;
				key1.index2 = j;
				el.set(v);
				writer.append(key1, el);
			}
		}
	}
	writer.close();
}
```

3 Implementation

The first procedure for block-based Matrix Multiplication is to determine how to split block which is also closely related with how to conduct block matrix multiplication. The basic theory of sub-matrix multiplication can be illustrated by the following diagram:



Assume there are two 4x4 matrices, then use a 2x2 sub-matrix to split the matrix. Here,

C_{000}=A_{00} x B_{00}

C_{010}=A_{01} x B_{10}

C_{00}=C_{000}+C010=A_{00} x B_{00} + A_{01} x B_{10}

Thus, $Csu = \sum_{t=0}^{T} Cstu = \sum_{t=0}^{T} Ast\ Btu$

Since the matrix may be unbalanced, which means that s and t or t and u are not necessarily equal, the special situation for the last block of each low or each column must be especially considered and handled in the program.

3.1 Notations and Definitions

In the problem, some notations are used and also the puedo-code will use the same notations to illustrate the different block distribution plans discussed in the following chapter.

Suppose:

Matrix A has dimension maxS x maxT with elements a(s, t) for 0 <= s < maxS and 0 <= t < maxT

Matrix B has dimension maxT x maxU with elements b(t, u) for 0 <= t < maxT and 0 <= u < maxU

Then:

Matrix C = A*B has dimension maxS x maxU with elements c(s, u) defined as:

c(s, u) = sum over 0 <= t < maxT of a(s, t)*b(t, u)

Let:

S = Number of rows per A block and C block.
T = Number of columns per A block and rows per B block.
U = Number of columns per B block and C block.

numS = number of A row and C row partitions = (maxS-1)/S+1
numT = number of A column and B row partitions = (maxT-1)/T+1
numU = number of B column and C column partitions = (maxU-1)/U+1

Use the following notation for the blocks. For all:

0 <= sb < numS
0 <= tb < numT
0 <= ub < numU

Define:

A[sb, tb] = The block of A consisting of rows S*sb through min(S*(sb+1),maxS)-1 (consider the last block may be unbalanced) columns T*tb through min(T*(tb+1),maxT)-1

B[tb, ub] = The block of B consisting of rows T*tb through min(T*(tb+1),maxT)-1 columns U*ub through min(U*(ub+1),maxU)-1

C[sb,ub] = The block of C consisting of rows S*sb through min(S*(sb+1),maxS)-1 columns U*ub through min(U*(ub+1),maxU)-1

C[sb,tb,ub] = A[sb,tb] * B[tb,ub]

With this notation,

C[sb,ub] = sum over 0 <= tb < numT of A[sb,tb]*B[tb,ub]

= sum over 0 <= tb < numT of C[sb,tb,ub]

Note that:

A blocks have dimension SxT.
B blocks have dimension TxU.
C blocks have dimension SxU.

Except for the last blocks at the bottom and right edges of A, B and C, which might have smaller dimension,

A has numS*numT blocks.
B has numT*numU blocks.
C has numS*numU blocks.

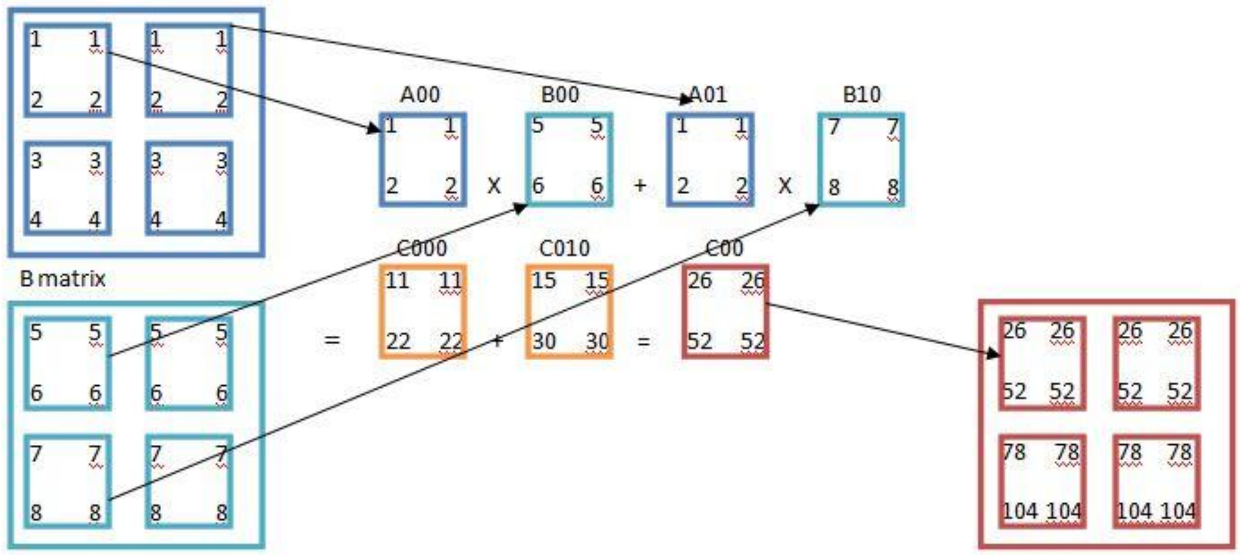
3.2 Algorithm Overview

Then, consider how Mapper and Reducer operate throughout the whole process.

For Mapper, as discussed above, the input Key/Value pairs are Key1 class, which has two fields index1 and index2 indicating row and column position for each cell respectively, and IntWritable class indicating the value in each cell. The diagram illustrated above has actually already implied the job of mapper, which is to transfer the cell-based dimension parameters as the block-based dimension parameters, that is to describe the position of cell in the scope of a sub-matrix it belongs to instead of in the scope of whole matrix. This is the output key class of Mapper. Meanwhile, in order to get the accurate position of each cell in C matrix (result matrix), the original position (cell-based) of cells in A and B matrices are also needed. Thus, for the output Value class, not only the value of each cell but also the original position (cell-based) will be stored.

Key2 has three fields (or four fields will be discussed below), they are index1, index2, index3, recording the block position. Value also has three fields, they are index1, index2, which are the same as Key1 for the cell, and value which is the same as the IntWritable.

For example,



For A block,

Input: {Key1, IntWritable}

…

{0, 0}, 1 (A00 block)

{0, 1}, 1 (A00 block)

{0, 2}, 1 (A01 block)

…

Output: {Key2, Iterator<Value>}

…

{{0,0,0}, <{0,0,1}, {0,1,1}>} enter into C000 block (reducer), to calculate C000

{{0,1,0},< {0,2,1}>} enter into C010 block (reducer), to calculate C010

…

Thus, the input key class is Key2, which indicate the Cstu block index as analyzed above. Reducers will do the multiplication between A blocks and corresponding B blocks, Ast*Btu, and then sum the Cstu blocks into Csu block. The output class will then be written as Key1 and IntWritable again. Here, Key1 represents the cell dimension in C matrix, IntWritable stores the result of each C cell.

Also, take the following example,

In C00, cells will be represented as

{Key1, IntWritable}

{{0,0}, 26}

{{0,1}, 26}

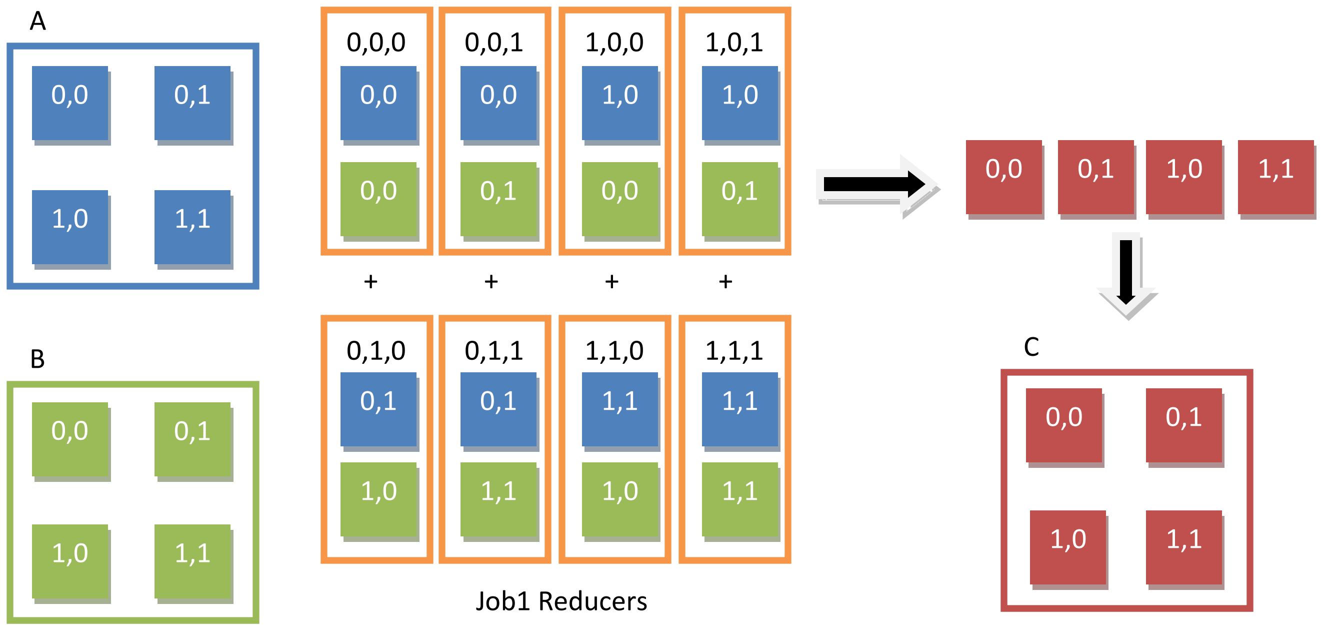
{{1,0}, 52}

{{1,1}, 52}

Then, the distribution models of the reducers (partition) will play a critical role in affecting the performance of matrix multiplication.

3.2.1 Job Structure 1

The simplest strategy is to have each reducer do just one of the block multiplications. Reducer R[sb, tb, ub] is responsible for multiplying A[sb, tb] times B[tb, ub] to produce C[sb, tb, ub]. There have a maximum of numS*numT*numU reducers multiplying blocks in parallel.



The Mappers must route a copy of each A[sb, tb] block to all of the R[sb, tb, ub] reducers, for each 0 <= ub < numU. This is numU copies of A and a total of numU*maxS*maxT key/value pairs, for the worst case where A is dense with no zero elements. Similarly, the Mappers must route a copy of each B[tb, ub] block to all the R[sb, tb, ub] reducers, for each 0 <= sb < numS.

This is numS copies of B for a worst-case total of numS*maxT*maxU key/value pairs. Thus with block-based strategy, if transferring a worst-case of total maxT*(numU*maxS + numS*maxU) key/value pairs over the network during the sort & shuffle phase.

In this job structure, two jobs are needed. The Mapper in Job1 is used to transfer each input record (Key1, IntWritable) into the format of (Key2, Value) outputs and distribute each A block, A[sb, tb], and corresponding B block, B[tb, ub] to the correct Reducer R[sb, tb, ub] according to Key2. Here, the Key2 contains four field, index1, index2, index3 representing sb, tb, ub and another field m (=0 or 1) to represent from which file this record comes from, that is from Matrix A file if m=0 or Matrix B file if m=1. Each Job1 Reducer will do the multiplication between A[sb, tb] and B[tb, ub] and parse the intermediate result C[sb, tb, ub] to Job2 Mapper with the format of Key1, representing each C block cell. Job2 Mapper uses a simple identity Mapper to sort and shuffle records and Job2 Reducer also uses a built-in IntSumReducer to sum the records with the same Key1.

This strategy uses lots of reducers, which makes good use of parallelization, but it also requires a large amount of network traffic. We might prefer alternatives which use less network traffic, possibly at the expense of fewer reducers with a lower level of parallelization.

Puedo-code for Job Structure 1,

```
map (key, value)
  if from matrix A with key=(s,t) and value=a(s,t)
    for 0 <= ub < numU
      emit (s/S, t/T, numU, 0), (s mod S, t mod T, a(s,t))
  if from matrix B with key=(t,u) and value=b(t,u)
    for 0 <= sb < numS
      emit (sb, t/T, u/U, 1), (t mod T, u mod U, b(t,u))

reduce (key, valueList)
  if key is (sb, tb, ub, 0)
    // Save the A block.
    ms = sb
    mt = tb
    Zero matrix A
    for each value = (s, t, v) in valueList A(s,t) = v
  if key is (sb, tb, ub, 1)
    if sb != ms or tb != mt return   // A[sb,tb] must be zero!
    // Build the B block.
    Zero matrix B
    for each value = (t, u, v) in valueList B(t,u) = v
    // Multiply the blocks and emit the result.
    sbase = sb*S
    ubase = ub*U
    for 0 <= s < row dimension of A
      for 0 <= u < column dimension of B
        sum = 0
```

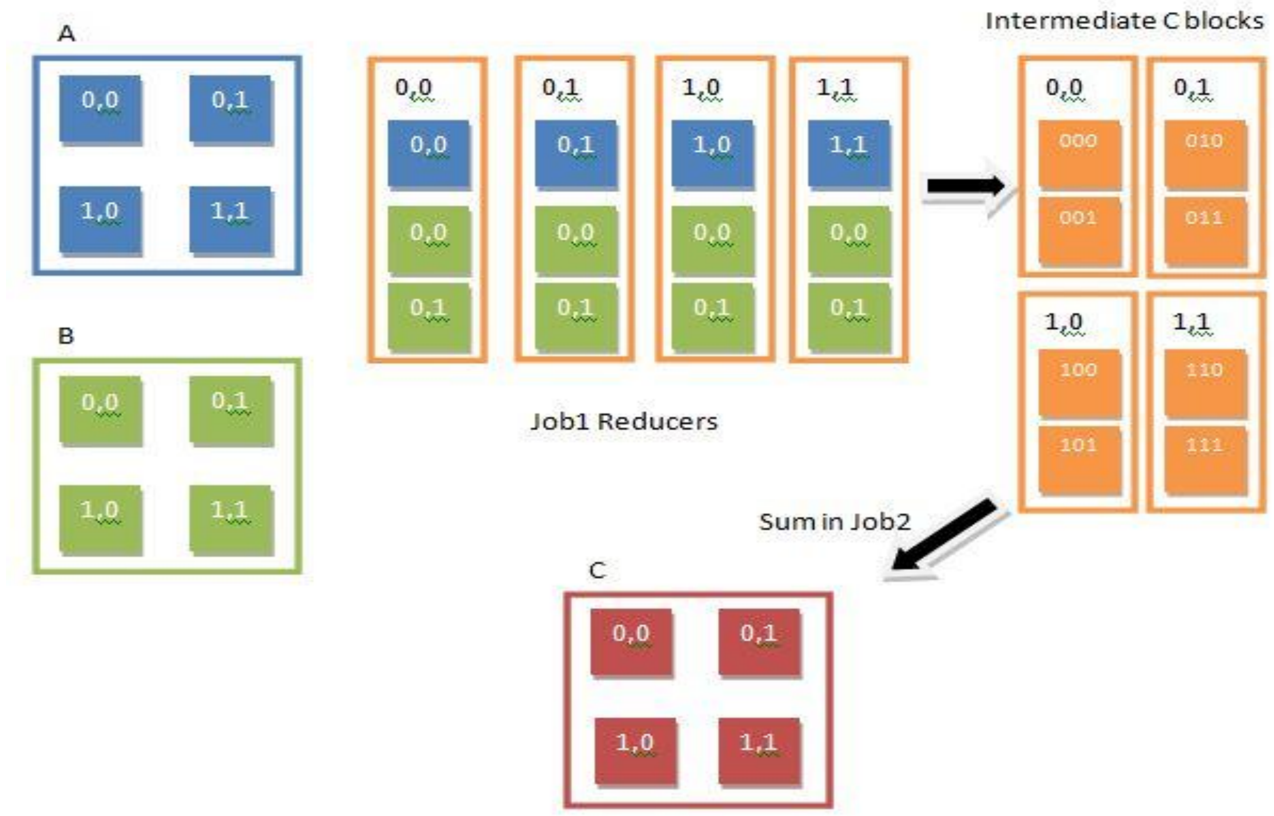
```
for 0 <= t < column dimension of A = row dimension of B
  sum += A(s,t)*B(t,u)
        if sum != 0 emit (sbase+s, ubase+u), sum
```

3.2.2 Job Structure 2

In this strategy, a single Reducer is used to multiply a single A block times a whole row of B blocks. That is, for each A block A[sb,tb], use a single Reducer R[sb,tb] that is responsible for multiplying the A block times all the B blocks B[tb,ub] for 0 <= ub < numU. This involves a maximum of numS*numT Reducers. With this stategy, the data for an A block A[sb,tb] only has to be routed to the single Reducer R[sb,tb], and the data for a B block B[tb,ub] has to be routed to the numS Reducers R[sb,tb] for 0 <= sb < numS.

The worst-case number of intermediate key/value pairs transferred over the network is maxS*maxT + numS*maxT*maxU = maxT*(maxS+numS*maxU). This is a considerable improvement over strategy 1 in terms of network traffic, at the cost of fewer Reduers each of which has to do more work, resulting in a lower level of parallelization.

This Strategy also needs two jobs. The process is similar as Job Structure 1. Job 1 Reducer do the multiplication and Job2 Reducer do the addition with IntSumReducer.



Peudo-code for Job Structure 2,

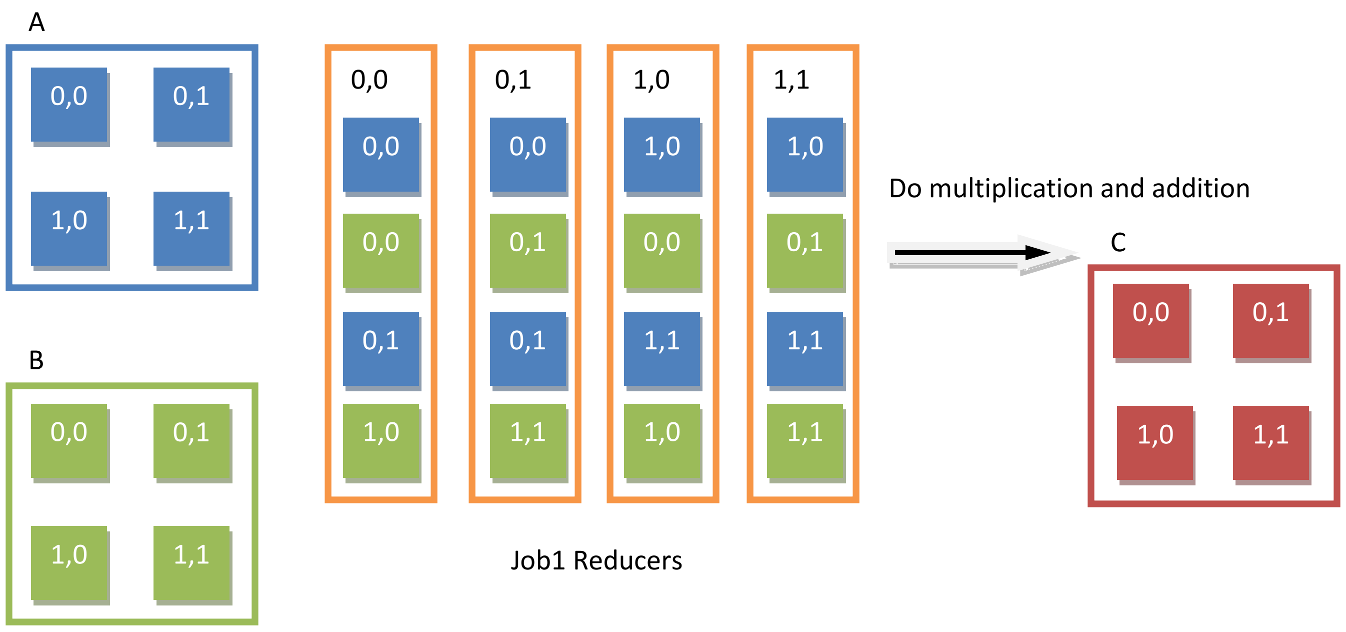
```
map (key, value)
  if from matrix A with key=(s,t) and value=a(s,t)
     emit (s/S, t/T, -1), (s mod S, t mod T, a(s,t))
  if from matrix B with key=(t,u) and value=b(t,u)
    for 0 <= sb < NIB
      emit (sb, t/T, u/U), (t mod T, u mod U, b(t,u))

reduce (key, valueList)
  if key is (sb, tb, -1)
    // Save the A block.
    ms = sb
    mt = tb
    Zero matrix A
    for each value = (s, t, v) in valueList A(s,t) = v
  if key is (sb, tb, ub) with ub >= 0
    if sb != ms or tb != mt return   // A[sb,tb] must be zero!
    // Build the B block.
    Zero matrix B
    for each value = (t, u, v) in valueList B(t,u) = v
    // Multiply the blocks and emit the result.
    sbase = sb*S
    ubase = ub*U
    for 0 <= s < row dimension of A
      for 0 <= u < column dimension of B
        sum = 0
        for 0 <= t < column dimension of A = row dimension of B
          sum += A(s,t)*B(t,u)
        if sum != 0 emit (sbase+s, ubase+u), sum
```

3.2.3 Job Structure 3

In the first two strategies presented above, each Reducer emits one or more C[sb,tb,ub] blocks, and thus a second MapReduce job have to be used to sum up over tb to produce the final C[sb,ub] blocks.

In this strategy, only a single Reducer R[sb,ub] to compute the final C[sb,ub] block, and there's no need for a second MapReduce job. The Reducer receives from the Mappers all the A[sb,tb] and B[tb,ub] blocks for 0 <= tb < numT, interleaved in the following order:

A[sb,0] B[0,ub] A[sb,1] B[1,ub] ... A[sb,numT-1] B[numT-1, ub]



The reducer multiplies the A and B block pairs and adds up the results. That is, it computes and emits the sum over 0 <= tb < numT of A[sb,tb]*B[tb,ub].

The maximum number of reducers with this strategy is numS*numU.

The Mappers must route a copy of each A[sb,tb] block to all of the R[sb,tb] reducers, for each 0 <= ub < numU. This is numU copies of A and a total of numU*maxS*maxT key/value pairs, for the worst case. Similarly, the Mappers must route a copy of each B[tb,ub] block to all the R[tb,ub] Reducers, for each 0 <= sb < numS. This is numS copies of B for a worst-case total of numS*maxT*maxU key/value pairs. Thus, as in Job Structure 1, a worst-case that total of maxT*(numU*maxS + numS*maxU) key/value pairs will be transferred over the network during the sort & shuffle phase.

Peudo-code for Job Structure 3,

```
map (key, value)
  if from matrix A with key=(s,t) and value=a(s,t)
    for 0 <= ub < NJB
      emit (s/S, ub, t/T, 0), (s mod S, t mod T, a(s,t))
  if from matrix B with key=(t,u) and value=b(t,u)
    for 0 <= sb < NIB
      emit (sb, u/U, t/T, 1), (t mod T, u mod U, b(t,u))

var ms = -1
var mu = -1
var mt = -1
reduce (key, valueList)
  // key is (sb, tb, ub, m)
  if sb != ms or ub != mu
    // Start a new (sb,ub) sequence.
    if ms != -1
```

```
      // Emit the C block for the previous sequence.
      sbase = ms*S
      ubase = mu*U
      for 0 <= s < row dimension of C
        for 0 <= u < column dimension of C
          v = C(s,u)
          if v != 0 emit (sbase+s, ubase+u), v
    ms = sb
    mu = ub
    mt = -1
    Zero matrix C
  if key is (sb, ub, tb, 0)
    // Save the A block.
    mt = tb
    Zero matrix A
    for each value = (s, t, v) in valueList A(s,t) = v
  if key is (sb, ub, tb, 1)
    if tb != mt return   // A[sb,tb] must be zero!
    // Multiply the A and B blocks and add them into C.
    for each value = (t, u, v) in valueList
      for 0 <= s < row dimension of A
        C(s,u) += A(s,t)*v
```

At the end of the reducer task we must emit the last C block.

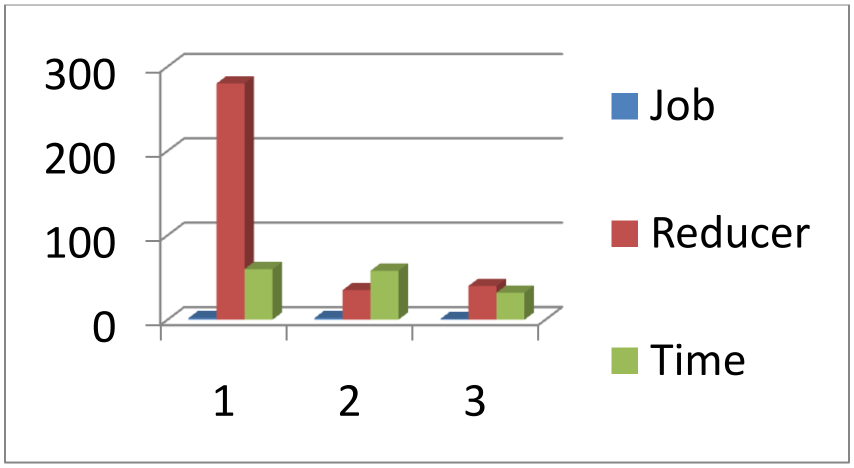
```
cleanup ()
  if ms != -1
    // Emit the last C block.
    sbase = ms*S
    ubase = mu*U
    for 0 <= s < row dimension of C
      for 0 <= u < column dimension of C
        v = C(s,u)
        if v != 0 emit (sbase+s, ubase+u), v
```

4 Analysis

To analyze the performance of these three strategies under the matrix layout format discussed, three groups of simulations with Duke Cluster are executed and simulation results are recorded in the following tables.

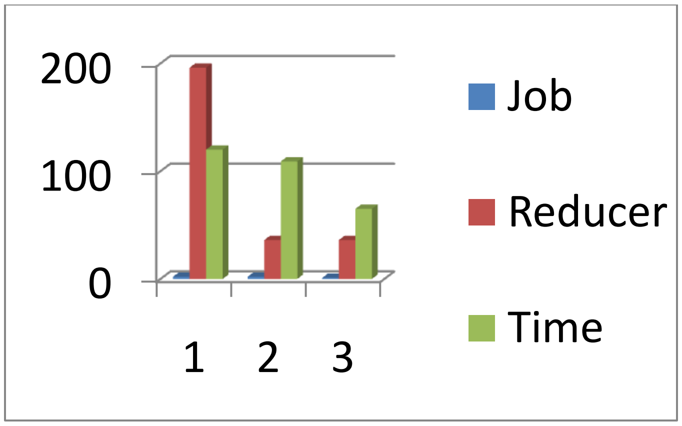
In first group of tests, the size of A is 50x68, B 68x72 is used as the input matrices. Run the matrix multiplication problem with three strategies respectively. The sub-matrix size is 10x10.

Plan	Job	Reducer	Running Time (s)
1	2	280	60
2	2	35	58
3	1	40	32



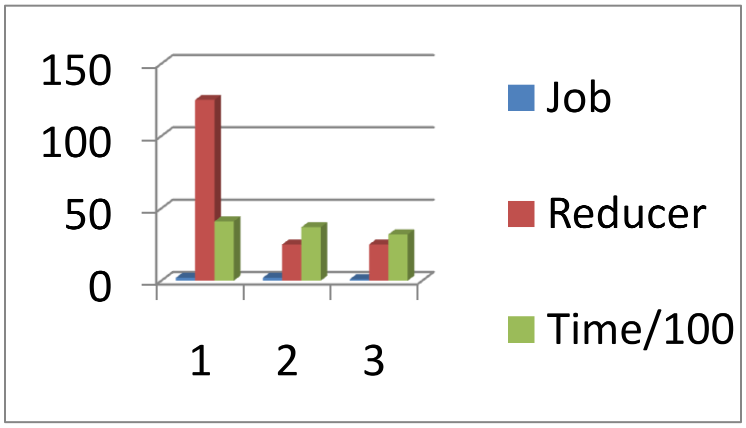
Second group, increase the size of input matrices, where A is 103x112, B 112x119. The sub-matrix size is 20x20.

Plan	Job	Reducer	Running Time(s)
1	2	196	120
2	2	36	109
3	1	36	65

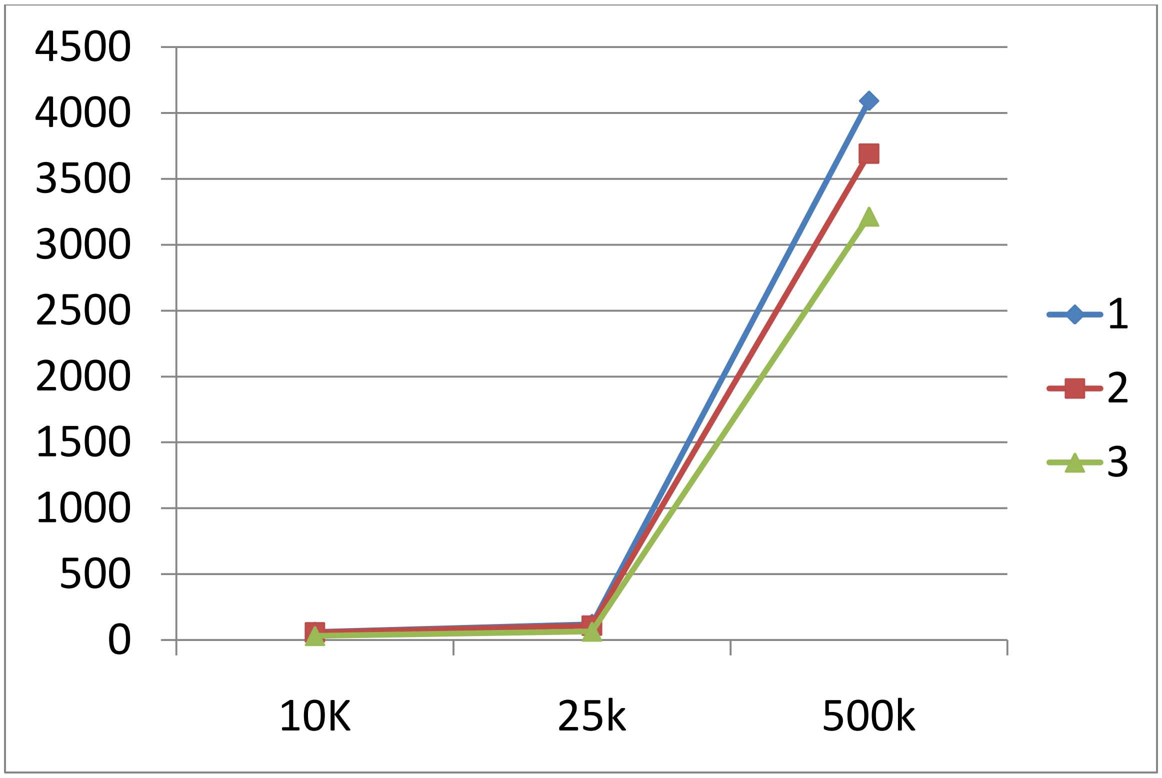


In the third group, further increasing the size of input matrices, where A is 504x523, B is 547. The sub-matrix size is 100x100.

Plan	**Job**	**Reducer**	**Running Time (s)**
1	2	125	4091
2	2	25	3692
3	1	25	3216



From the above three simulations, the conclusion can be drawn that Job Structure 2 and 3 use less reducers than Job Structure 1, thus causing less I/O costs. Moreover, compare the computation time, Job Structure 3 uses less time than the other two.



Plot the three group of simulations in one graph, it can be seen more clearly that, with the increase of input records amount, Job Structure 3 can achieve a better performance.

5 Conclusion

In this project, the data access patterns in matrix files are discussed and a new concentric data layout solution is proposed to facilitate matrix data access. From the analysis of some simulations conducted in MapReduce framework, several conclusions can be drawn. First, the data layout proposed maintains the dimensional property in large data sets. The data is stored in sub-matrix levels and then transferred from the same sub-matrix into one chunk. This matches well with the matrix like computation. By preprocessesing the data beforehand, and optimizes the afterward job structure of MapReduce application. Second, the experiments indicate that Job Structure 3 causes less I/O costs and uses least computation time, and thus is proved to be the optimal matrix multiplication method.

Reference

[1] Yi Zhang, Parallel Matrix Layout for MapReduce

[2] Yi Zhang, Herodotos Herodotou, and Jun Yang, RIOT: I/O-E_cient Numerical Computing without SQL. In CIDR, 2009.

[3] John Norstad, A MapReduce Algorithm for Matrix Multiplication