CPS216 Project Report
Matrix Multiplication in MapReduce

Botong Huang and You Wu (Will)

Department of Computer Science
Duke University
{bhuang, wuyou}@cs.duke.edu

1 INTRODUCTION

The MapReduce framework proposed by Google has emerged to be a popular model for processing massive amount of data in parallel. On the other hand, matrix multiplication, which is a common and important algebraic operation in many applications, involves large amounts of computation and it has the potential to be parallelized. In this project, we study the matrix multiplication problem in MapReduce. We focus on designing, implementing and analyzing algorithms based on the conventional approach which runs in $\Theta(n^3)$ time in the non-parallel setting.

In a typical MapReduce job, each map task is responsible for processing one piece of the input file(s) called a split. A file split, by default in Hadoop, contains a piece of data from a single input file. This indicates that if the two input matrices are stored in separate HDFS files, one map task would not be able to access the two input matrices at the same time. We found two approaches in previous works that deals with this problem.

The first one comes from an excise project by John Norstad from Northwestern University [1]. The problem was solved by starting the multiplication at the reduce side. In this way, two MapReduce jobs are needed to finish the whole task. The map phase of the first job reads in the two matrices to be multiplied and distributes them to the reducers as needed. The reduce phase does block-level multiplication, and flushes the result into temporary HDFS file. Then in the second job, identity mappers are used to read the blocks back from the temporary files and pass them on to the reducers, where blocks are added up to produce the final result.

In this approach, map phases in both jobs are used to distribute data properly for the reducers. The block multiplications and additions are done in the reduce phases. As we can see, data are copied repeatedly throughout this process, both in the two shuffle phases and during HDFS I/O to/from the temporary files between the two jobs.

The other approach was used in the Hadoop HAMA project [2]. They come up with a preprocessing phase to reorganize the input file according to some strategy so that the matrix multiplication can finish in one MapReduce job. More specifically, one input file containing replicated block data from both matrices are generated in an interleaving manner, so that each file split contains the exact data a map task needs for the block-level multiplication. Then the mappers do block multiplications and the reducers are responsible for adding the blocks up to obtain the output matrix. In this approach, the preprocessing phase, which is basically copying data and make replicates of blocks of the two input matrices, incurs large amount of extra file system I/O.

Having evaluated the pros and cons of both approaches, we modified the Hadoop MapReduce input format and enabled a mapper to get the proper block data from both input matrices without the preprocessing phase. As a result, we are able to multiply two matrices in one MapReduce job, and at the same time, avoid the preprocessing cost. Details about our modification on input format can be found in Section 3.2.

Using this technique, we came up with three different strategies to implement matrix multiplication, which will be explained in Section 2.1. We also implemented a strategy for sparse matrix multiplication using the same technique, which will be described in Section 2.2.

Experiments have been conducted for both dense and sparse matrix multiplication. Results will be shown and evaluated in Section 4. We conclude our work in Section 5. Future work will also be discussed.

2 STRATEGIES

In this section, we describe three strategies for multiplying two dense matrices and one strategy for multiplying sparse matrices in MapReduce conceptually. For simplicity, we assume that both of the input matrices are square matrices.

To facilitate quantitative analysis on different strategies, we denote by M the size of the input matrices, n the number of blocks partitioned along each side of the input matrix, and N the number of physical map/reduce slots limited by the number of nodes in the cluster.

As opposed to existing works that shuffle all data from mappers to reducers and let reducers do all the block multiplications, we are able to do the block multiplications, which dominates the total time of computation, at the map phase, using the techniques described in Section 3, thus speeding up our program in practice.

2.1 Dense Matrix Multiplication

Strategy 1. As shown in Figure 1, we partition each of the input matrices into $n \times n$ small square blocks of equal size. The size of each block would be $\frac{M}{n} \times \frac{M}{n}$. The output matrix would consists of $n \times n$ blocks, each resulting from the addition of n block matrix multiplications. We let each map task handle one block matrix multiplication. So there would be totally n^3 map tasks.

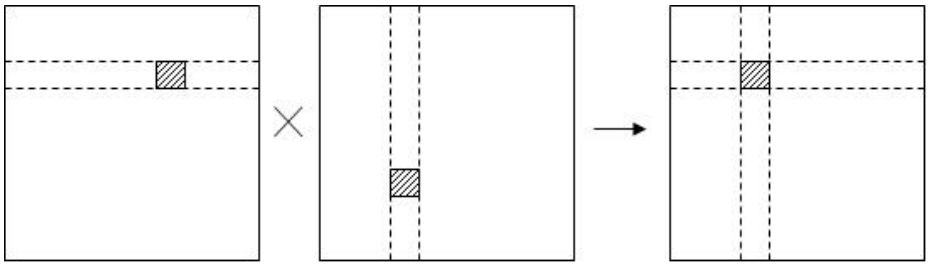


Fig. 1: Strategy 1 for dense matrices. Block by block per map task

Strategy 2. As shown in Figure 2, the first input matrix is partitioned into n row strips, while the second input matrix is partitioned input n column strips. Each map task reads one row strip from the first matrix and one column strip from the second matrix, and produces one $\frac{M}{n}$-by-$\frac{M}{n}$ block in the output matrix. The total number of map tasks for this strategy is n^2.

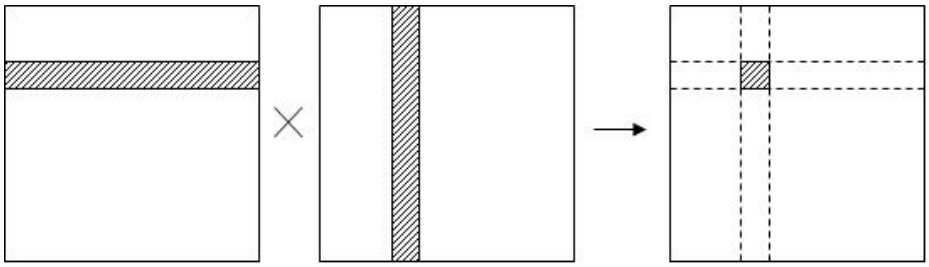


Fig. 2: Strategy 2 for dense matrices. Row strip by column strip per map task

Strategy 3. In this strategy, we partition the first matrix into $n \times n$ square blocks, and the second one into n row strips. A map task multiplies a block from the first matrix and a row strip from the second, and produces partial results for a row strip of the output matrix (Figure 3). This strategy also requires n^2 map tasks.

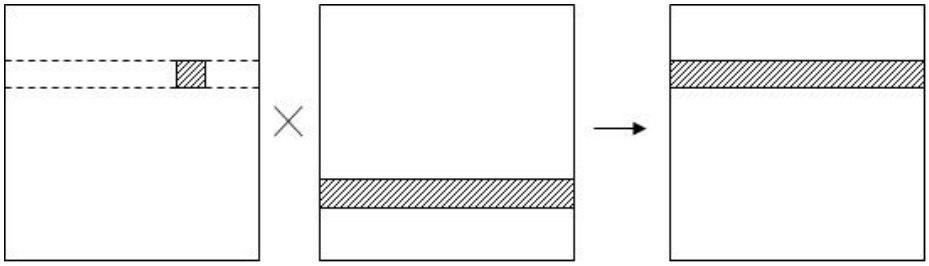


Fig. 3: Strategy 3 for dense matrices. Block by row strip per map task

Analysis. Using the notations M and n defined at the beginning of this section, we compare the three strategies in term of the following aspects: the total input traffic on the map side, average traffic into each map task, the amount of shuffle traffic from the mappers to the reducers, the amount of computation done by each map task (in terms of number of block multiplications), memory needed for each map task, and the total number of mappers. The comparison are summarized in Table 1.

Table 1: Summary 1 on the three three strategies

	Strategy 1	Strategy 2	Strategy 3
Map input traffic (total)	$2M^2n$	$2M^2n$	M^2n
Map input traffic (average)	$2M^2/n^2$	$2M^2/n$	M^2/n
Shuffle traffic	M^2n	M^2	M^2n
Computation per map task	1	n	n
Memory per map task	$3M^2/n^2$	$2M^2/n$	$2M^2/n$
Number of map tasks	n^3	n^2	n^2

Since the degree of parallelism is limited by the number of physical map/reduce slots, let us assume that for each strategy the number of map tasks matches the number of physical map slots N. We substitute $N^{1/3}$ or $N^{1/2}$ for all n's in Table 1 and obtain Table 2.

One thing worth mentioning is that if the number of map tasks is no less than the number of map slots, the parallelism is fully exploited. Since the total amount of multiplication is fixed, when the number of map slots is also fixed, for different strategies, the amount of multiplication done in each map task should be the same across different strategies. However, the "Computation per map task" of Strategy 1 shown in Table 2 is less than that of Strategies 2 and 3. This is because computation per map task counts the number of block multiplications done by each map task. When we fix the number of map slots and set the number of map tasks equal to the number of map slots, the block size for Strategy 1 is large than that of Strategy 2 and 3. It is easy to verify that taking block sizes into account, the actual amount of computation per map task is the same for all three strategies.

Table 2: Summary 2 on the three three strategies

	Strategy 1	Strategy 2	Strategy 3
n	$N^{1/3}$	$N^{1/2}$	$N^{1/2}$
Map input traffic (total)	$2M^2N^{1/3}$	$2M^2N^{1/2}$	$M^2N^{1/2}$
Map input traffic (average)	$2M^2N^{-2/3}$	$2M^2N^{-1/2}$	$M^2N^{-1/2}$
Shuffle traffic	$M^2N^{1/3}$	M^2	$M^2N^{1/2}$
Computation per map task	1	$N^{1/2}$	$N^{1/2}$
Memory per map task	$3M^2N^{-2/3}$	$2M^2N^{-1/2}$	$2M^2N^{-1/2}$

In Section 4, we will show experiment results for all of the three strategies described in this subsection. We will analyze the results and see if they agree with our analysis in Table 2.

2.2 Sparse Matrix Multiplication

For sparse matrices, we use a simple analogy to strategy 3 described in Section 2.1. Figure 4 shows how a row in result matrix C is obtained by multiplying the corresponding row in matrix A with the whole matrix B. In our approach, we use one line in matrix A as the unit to partition the workload between mappers. Basically, we give each map tasks some consecutive lines from matrix A such that the number of non-zero values in those lines is roughly the same for different mappers. Suppose the distribution of non-zero values in B is independent of that of A, the expected workload for each mapper should be approximately the same.

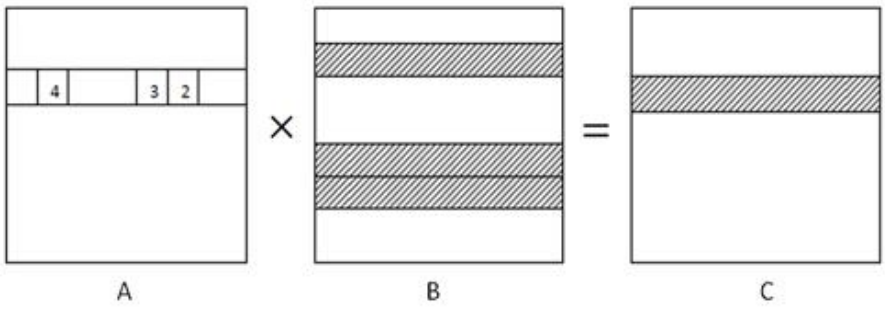


Fig. 4: Strategy for sparse matrices. Row of A by B

3 IMPLEMENTATION

In this section, we introduce how the input matrices are laid out are HDFS, how we modified the Hadoop code to distribute data properly before the map phase, and how our programs can be configured.

3.1 Matrix Layout

For dense matrices, as want our program to be flexible in both matrix size and block size, we use plain row major order to store the matrices on HDFS. For a matrix of size M, entries are stored in the following order: $(0,0)(0,1)...(0,M-1)(1,0)(1,1)...(1,M-1)...(M-1,0)(M-1,1)...(M-1,M-1)$.

For sparse matrices, we use the following row-major-like strategy (Figure 5).

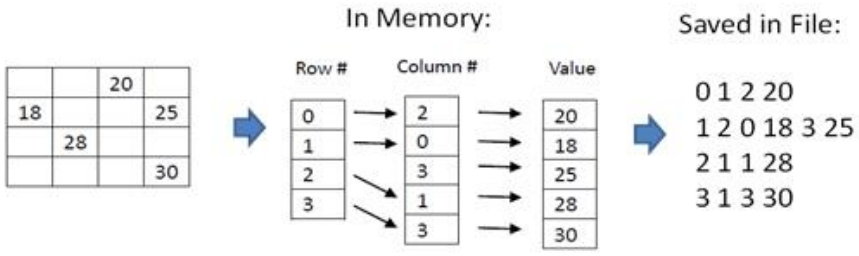


Fig. 5: Sparse matrix layout on HDFS

In this way, a mapper can obtain the rows and columns it needs by only scanning through a consecutive part of the file.

3.2 Input Format for MapReduce

The Hadoop MapReduce framework has provided many different types of data input formats. But almost all of them concentrate on the data type within the input file rather than the input file structure. Here is part of the code for class ***FileSplit*** used by all input formats. It contains four variables:

```
private Path file;
private long start;
private long length;
private String[] hosts;
```

The data corresponding to this split start from offset *start* to offset *start+length*

within file $file$. $hosts$ is the physical location of this file split, which is later used to choose a data-local map task. As we can see, this file split structure limits the mapper from reading data from multiple file. So we change its structure by extending the class ***FileSplit*** as follows.

```
Path fileA;
Path fileB;
long startA[], startB[];
int x, y, z;
int M, n;
```

In our file split, we include more information. Firstly, we have two file paths for the two input matrices. Secondly, as we store the input matrices in row major order, a matrix block required by a mapper is not consecutive in the input file. So two arrays $startA[]$ and $startB[]$ are needed to record all the starting offsets for the blocks. M and n stand for the matrix size and the number of blocks per row/column as defined in Section 2. x, y, and z stand for the block level indices of the block, letting the mapper know the location of the block in the input matrix.

With this new file split structure, we implemented our own ***FileInputFormat*** which generates the file split in our desired fashion, and ***RecordReader*** which takes our file split as input. Then all data are distributed properly to the mappers so that the computation can be carried out in the map phase.

3.3 Configurations

For dense matrices, we leave the number of blocks n as a configurable parameter. Then the user can control the number of map tasks by tuning the value of n.

For sparse matrices, we give user the right to set the number of mappers. As we will see in Section 4, in order to minimize the total running time we tend to have the number of mappers slightly smaller than the number of physical map slots determined by the size of the cluster. In this way, we can get the most parallelism out of the cluster, while avoiding the overheads of starting multiple map waves.

4 EXPERIMENTS

We tested our matrix multiplication strategies on the Amazon EC2 platform, on clusters of 4, 8, and 16 slave nodes. Each slave node has capacity of 3 map slots and 3 reduce slots, i.e. the maximum number of mappers that be run simultaneously is 12, 24, and 48, respectively.

4.1 Results for Dense Matrices

We use randomly generated square matrices as input.

Map Startup Overhead. We start with examining the impact of map startup overhead caused multiple map waves.

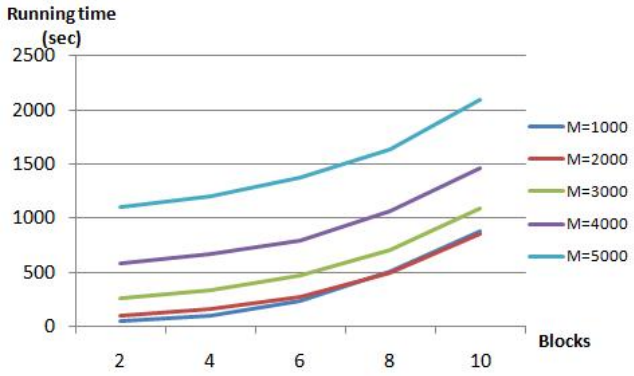
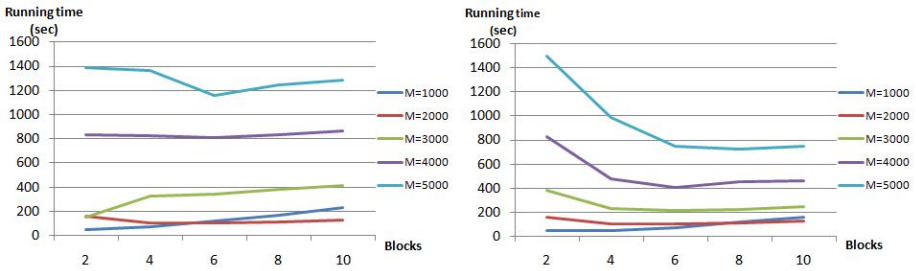


Fig. 6: Strategy 1. 4 nodes. Running time vs. number of blocks n

For strategy 1, the number of map tasks increase cubically with the increase of number of blocks partitioned along a row/column (n). With a 4-node cluster, we have 12 map slots. The number of map tasks exceeds the number of map slots starting from 3 blocks along each side of a matrix. After that point, the number of map waves increases linearly with the number map tasks, as the number of map slots is fixed, i.e. cubical with the number of blocks. In Figure 6, we observe a non-linear growth in running time when n increases linearly. We conjectured that this non-linear growth should be attributed to map startup overheads, and continued to test on the second strategy.

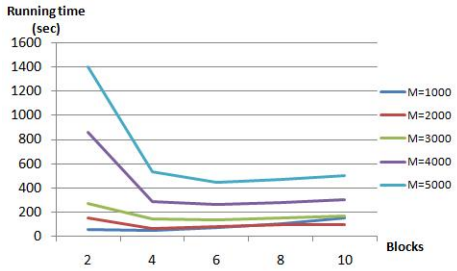
For strategy 2, when n is small, the number of mappers is much smaller than the number of map slots, which means the parallelism provided by the cluster is not fully utilized. This is why we observe a sharp decrease in running time in both Figure 7(b) and (c), when n is increased from 2 to 4. Based on our observation and conjecture in Figure 6, we expect to minimize the running time at $n \approx N^{1/2}$. The trend is not obvious in Figure 7 (a) due to the small size of the cluster. In Figures 7(b) and (c), the minimum running times are achieved when n is either 4 or 6.

Another factor that should not be overlooked is the amount of input traffic to the mappers as n increases. Because our input matrices are stored in entirety on HDFS, the rate of reading data from input files is bottlenecked. According to our analysis in Section 2.1, for all three strategies, the total amount of input traffic increases linearly with n. Taking this into account, the number of blocks n at which the running time is minimized would be shifted a little to the left of $N^{1/2}$, as we see in Figure 7(c). This effect is not as visible in Figure 7(b).



(a) 4 nodes

(b) 8 nodes



(c) 16 nodes

Fig. 7: Strategy 2. Running time vs. number of blocks n

Number of Nodes. We also examine the impact of the size of the cluster on the running time. For this experiment, we use strategy 3. For each size of the input matrices, we plot the minimal running time achieved by adjusting n. We start with a 4-node cluster and doubled the cluster size twice. Results are shown in Figure 8.

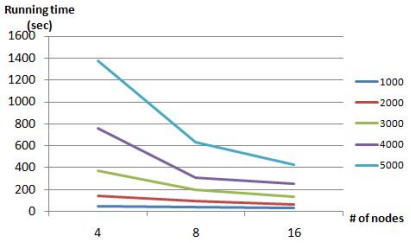


Fig. 8: Strategy 3. Running time vs. cluster size

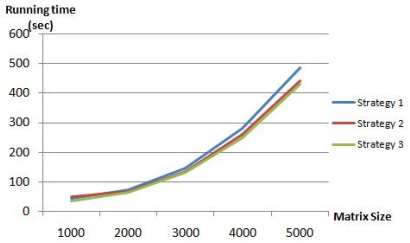


Fig. 9: Comparison between the three strategies

Ideally, when we double the cluster size, the running time should be halved. However, there are overheads such as HDFS I/O cost, shuffle cost, map startup cost, and etc. As we can see in Figure 8, when the number of nodes is increased

from 4 to 8, the running time is approximately halved because the overheads are not visible compared with the long running time. When the number of nodes is doubled again, from 8 to 16, the new running time is visibly larger than half of the original running time.

Comparing the Three Strategies. Like what we did for the previous experiment, for each combination of problem size and strategy, we adjust n to minimize the running time. The results are plotted in Figure 9. According to our analysis on the relationship between block size and map startup overhead, the running time is minimized when parallelism is fully exploited and as few map waves are involved as possible. Hence the time spent on computation should be roughly the same for all three strategies.

We have two observations from the figure. Firstly, strategy 3 slightly outperforms strategy 4 in all matrix sizes. There are two possible reasons for this small difference, both related to reading input from HDFS. First reason is that the amount of input traffic for strategy 3 is half of that of strategy 2. Second reason is that the data each mapper gets from the second input matrix is consecutive for strategy 3, but not for strategy 2.

The second observation brings memory limit to our attention. We can see from the figure that as the problem size increases, the running time of strategy 1 increases faster than strategy 2 and 3. This is because when matrix size increases to 4000/5000, none of the minimal running time was achieved at the theoretically optimal block size due to the large amount of memory required. As a result, we had to partition the matrix into smaller pieces, which leads to more map waves. As the number of mappers for strategy 1 grows faster than strategies 2 and 3 (cubic vs. quadratic) more map startup overheads are incurred. We discuss possible solution to the problem of memory limit in Section 5.

Others. We compared our strategy 1 with Norstad's analogy [1] on a 16-node cluster, on 5000-by-5000 matrices. We adjusted parameters to minimize the running time for both programs. Our program finished in 491 seconds while theirs finished in 2365 seconds. We also scaled up the input size to 10000-by-10000. It took our program 3916 seconds to finish, which is approximately $2^3 = 8$ times the running time to multiply two 5000-by-5000 matrices.

4.2 Results for Sparse Matrices

We generate our sparse matrix randomly. For a $m \times m$ sparse matrix, we let each value be non-zero with probability $\ln m/m$ independently, so that the expected number of non-zero values per line is $\ln m$, and $m \ln m$ for the whole matrix.

Using this method, we generated matrices with size from 100000 (100K) to 400000 (400K) and run the sparse matrix multiplication on the same cluster

above with 4, 8 and 16 nodes, which has 12, 24 and 48 map slots. The results are shown in Figure 10.

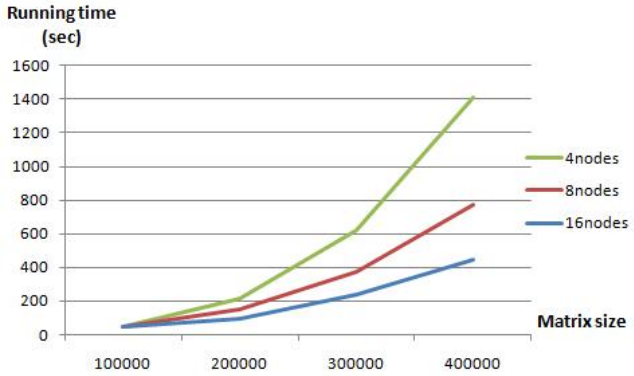


Fig. 10: Sparse Matrix Result

By setting the number of mapper to be slightly smaller than the number of map slot, the map phase could finish in one wave. So there won't be too much overhead on starting new mappers.

Given the same number of nodes, as the matrix size grows, the input traffic for each mapper grows linearly. The amount of computing operations for matrix with $m \ln m$ non-zero values are $m \ln^2 m$. Overall, we see in figure 10 the curves a growth in running time larger than linear.

Given the same input matrix size, the computation workload are the same. As the cluster grows from 4 nodes to 8 and 16 nodes, the running time are almost cut down by half twice. This is the result of the low overhead involved in our implementation.

5 CONCLUSIONS AND FUTURE WORK

In the work, we have studied previous works [1, 2] on parallel matrix multiplications in MapReduce. We have modified the Hadoop input format to allow a mapper to get non-consecutive data from multiple files. By doing this, we are able to complete a matrix multiplication in one MapReduce job, and at the same time, avoid the preprocessing overhead in the approach proposed by [2]. We applied this technique to three dense matrix multiplication strategies and one sparse matrix multiplication strategy, and conducted experiments on our programs.

Experiment results provided valuable information for adjusting parameters to

optimize performance. We have shown by experiment how to choose an appropriate block size for each dense matrix multiplication strategy, and how to choose a fair amount of mappers for sparse matrix multiplication. The goal is to match the number of mappers with the number of physical map slots so that the parallelism is fully utilized, and little overhead is incurred in starting up the mappers.

Problems have also been revealed in experiments. Firstly, in our implementation, the input files are stored in entirety on HDFS. Suppose there are a few copies of them on several nodes of the cluster. Mappers from all nodes would have to read data from these few nodes. Then the rate of reading input is bottlenecked by the bandwidth of these nodes. Since different mappers need different parts of the input matrices, what we could do is to partition the input files and put the partitions on different nodes of clusters. It is worth studying how many partitions should we have, and which nodes should hold which partitions, based on the communication cost between each pair of nodes. This partitioning process is closely related to the block size in the three dense strategies. The results in the previous paragraph can provide a useful guideline.

Another problem with the current approaches is the memory limit for dense matrix multiplication. Due to the memory limit of a mapper, sometimes the theoretically optimal block size is to large to be achieved for large matrices. Inspired by Norstad's work [1], we could partition the input matrices into smaller blocks and let each mapper handle multiple block multiplications. In this way, each mapper only require memory that is sufficient for one block multiplication, and we could avoid the overhead of starting multiple map waves.

For sparse matrices, we generated the inputs in such a way that for each of the input matrix, each value is non-zero independently with a fixed probability. Then each row/column has the same expected number of non-zero entries, and the two input matrices are not correlated. The question remains how to partition the input so as to balance the workload of mappers when the input matrices are correlated.

To sum up, we have improved the performance of dense matrix multiplication in MapReduce by slightly modifying the infrastructure of Hadoop, and packing all work into one MapReduce job. We have shown how to configure the number of blocks to optimize the performance through experiments, and suggested possible solutions to limitations in the current approach, such as bottleneck in data transfer bandwidth and memory limit of mappers. We will also continue to implement these ideas, and also examine the impact of correlation between the input matrices on performance for sparse matrices.

References

1. J. Norstad, "A MapReduce Algorithm for Matrix Multiplication", 2009

2. S. Seo, E.J. Yoon, J. Kim, S. Jin, J.S. Kim, and S. Maeng, “HAMA: An Efficient Matrix Computation with the MapReduce Framework”, 2010.
3. Y. Zhang, H. Herodotou, and J. Yang, “RIOT: I/O-efficient numerical computing without SQL”, in *CIDR*, 2009.