Parallel Matrix Multiplication using Sparse Data Representation

Team: Mean Partitions

Sahil Gandhi
sahilgandhi
@ccs.neu.edu

Mustafa Kapadia
mustafa8895
@ccs.neu.edu

Sarthak Kothari
sarthakkothari
@ccs.neu.edu

Siddhant Benadikar
siddhantb
@ccs.neu.edu

1. Introduction

Matrix multiplication in a distributed environment is a tricky concept. There are various approaches to solve the Matrix Multiplication problem. We can solve it using Row-By-Column partitioning or by Column-By-Row partitioning. Both approaches involve either duplicating data or having additional jobs to produce the final output. This proves to be a hindrance for big data, as the machines may not be able to store the partitioned data on the heap. In addition to the heap issue, the efficiency of both algorithms depends on the amount of data transferred between the mappers and the reducers, and especially the volume of data taken as input for the reduce phase, and the volume of data produced as output by the reduce phase. This depends entirely on the data distribution and the task at hand [3]. Research suggests that such problems can be solved using smaller partitions at the cost of data shuffle, however, the data-shuffle/duplication can be reduced using intelligent techniques. One such approach to solve Matrix Multiplication is the Block-Block partitioning technique. However, since there are a lot of duplicate tuples generated as intermediate data, the encoding of the data makes a huge difference in the feasibility of the problem. It strongly depends on the data properties, in particular, the sparseness of the input matrices and distribution of non-zero values over their cells. The aim of the project is to benchmark two widely used Block-Block partition approaches, Simple-Block-Block algorithm and Cannon's algorithm, that complement each other's shortcomings.

2. Dataset

Due to the limitation of AWS and to thoroughly understand the benefits and shortcomings of both algorithms, we did not use any publicly available dataset. We generated our own data using a simple pseudo-random Python script. Our preliminary experiments and research suggested that density of the the matrix (and not the dimension) affects the running time and resources used for the algorithm. Building on this finding, we kept the dimension of the matrix constant while changing the density for each.

We created 2 sets of 2 matrices (matrices to be multiplied), each of 10,000 x 10,000 dimension, with the following densities:

- 10% density with 10 million non-zero values per matrix
- 5% density with 5 million non-zero values per matrix

Since we work with sparse matrices, we represent the data as a Dictionary-Of-Key (DOC) data structure. DOK consists of a dictionary that maps (row, column)-pairs to the value of the elements. Elements that are missing from the dictionary are taken to be zero [5]. The format is good for incrementally constructing a sparse matrix in random order, but poor for iterating over non-zero values in lexicographical order. An example of such a data representation is shown below:

Original Matrix:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

DOK representation (*assuming all values, A, B, C & D, are non-zero*)
0,0,A
0,1,B
1,0,C
1,1,D

3. Methodology

We have picked two widely used algorithms for sparse matrix multiplication and compared their performance in terms of time taken and data transferred for varied data and partition sizes. Both these algorithms have their own benefits and downfalls.

3.1. Cannon's Algorithm

One of the most commonly and widely used algorithms for parallel Matrix multiplication is Cannon's algorithm. It is extremely suitable for square matrices. The main advantage of this algorithm is that its storage requirements remain constant and are independent of the number of processors [2]. The two NxN matrices A and B are divided into square submatrices of size $\frac{a}{\sqrt{p}}$ and $\frac{b}{\sqrt{p}}$ among the p processors. The sub-blocks of A and B residing with the processor (i, j) are denoted by A^{ij} and B^{ij} respectively, where $0 \leq i < \sqrt{p}$ and $0 \leq j < \sqrt{p}$. In the first phase of the execution of the algorithm, the data in the two input matrices are aligned in such a manner that the corresponding square submatrices at each processor can be multiplied together locally. This is done by sending the block A^{ij} to the processor (i, ((j - i) + $\sqrt{p}$) mod $\sqrt{p}$), and the block B^{ij} to the processor (((i - j) + $\sqrt{p}$) mod $\sqrt{p}$, j). The copied sub-blocks are then multiplied together. Now the A sub-blacks are rolled one step to the left and the B sub-blocks are rolled one step upward and the newly copied sub-blocks are multiplied and the results added to the partial results in the C sub-blocks. The multiplication of A and B is complete after $\sqrt{p}$ such steps.

3.1.1. Pseudo-Code

```
// make initial alignment
For i,j := 0 to $\sqrt{p}$-1 do
        send block $A_{i,j}$ to process $(i, (j - i + \sqrt{P}) \ mod \sqrt{P})$ and block $B_{i,j}$ to process $((i - j + \sqrt{P}) \ mod \sqrt{P}, \ j)$
Endfor;
Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

// compute-and-shift. A sequence of one-step shifts pairs up $A_{i,k}$ and $B_{k,j}$
// on process $P_{i,j}$ : $C_{i,j} = C_{i,j} + A_{i,k}B_{k,j}$
For step := 1 to $\sqrt{P} - 1$ do
    Shift $A_{i,j}$ one step to the left (with wraparound) and $B_{i,j}$ one step up (with wraparound)
    Process $P_{i,j}$ : multiply received submatrices together and add the result to $C_{i,j}$
```

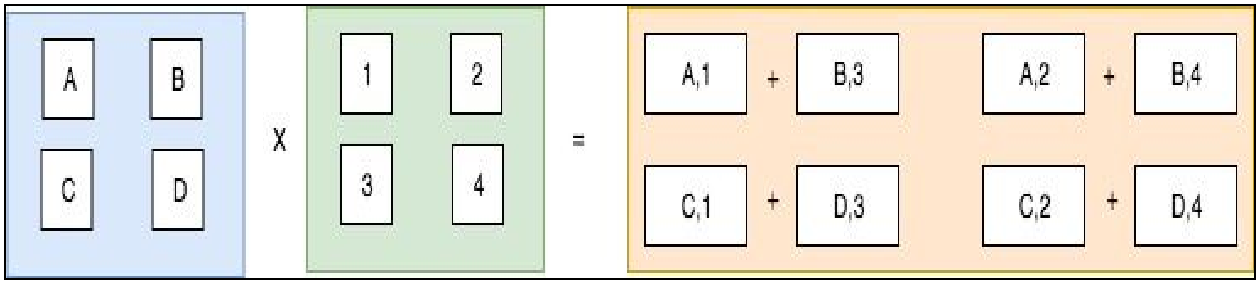
```
Endfor;

Remark: In the initial alignment, the send operation is to: shift $A_{i,j}$ to the left (with wraparound) by $i$ steps, and shift $B_{i,j}$ to the up (with wraparound) by $j$ steps.
```

3.1.2. Algorithm Analysis & Program Analysis

- **Total number of iterations:** $\sqrt{P}$
    - Cannon's algorithm requires multiple iterations as it aims to reduce the data duplication and spread the partial result submatrix computation across iterations
    - In our implementation, we pushed the aggregation of the partial result submatrices to a separate aggregation job which gives the final output matrix C
- **Data Duplication:**
    - There is no data duplication within an iteration in Cannon's algorithm
- **Matrix multiplication in Reduce stage:**
    - Cannon's algorithm follows a row-by-column approach for matrix multiplication, hence, the submatrix multiplication in the reduce phase mimics an optimized version of the sequential 3-loop matrix multiplication logic.
- **Limitations:**
    - Cannon's algorithm works with an inherent assumption that the submatrices are square matrices. This further puts a restriction that the rows and columns of the input matrices be divisible by $\sqrt{P}$ (where P defines the number of blocks per matrix).
    - One can easily overcome this limitation by choosing a value of the rows and columns that is divisible by $\sqrt{P}$, where the added values are assumed to be zero. This does increase the data size, but in a sparse representation has no effect.

3.2. Simple Block-Block (SBB) Algorithm



Another widely used technique for Matrix Multiplication is Block Block Partition where we divide the matrices into sub-blocks, send the multiplicative blocks together to the same reduce task and aggregate to get the results. In contrast to the Cannon's algorithm, we use a column-by-row approach to multiply the submatrices.

3.2.1. Pseudo Code

```
// duplicate and align all row-blocks of A matrix to the corresponding column of C matrix
// duplicate and align all column-blocks of B matrix to the corresponding row of C matrix
For i,j := 0 in blocks of result matrix $C$, do
```

```
    Duplicate and send block $A_{i,j}$ to processor $P_{k,j} \forall k \in$ all blocks in the column of C
    Duplicate and send block $B_{i,j}$ to processor $P_{i,k} \forall k \in$ all blocks in the row of C

// multiply the submatrices in each processor using the column-by-row algorithm

// aggregate all partial results from submatrices multiplication
```

3.3.2. Algorithm and Program Analysis

- **Total number of iterations:**
  - A single iteration of 2 jobs
  - Job 1 computes the partial results
  - Job 2 aggregates the partial results
- **Data Duplication:**
  - A large amount of data duplication as blocks from both the input matrices are all-to-all broadcasted
  - This results in very large intermediate data and heavy shuffling between the map and reduce phases
- **Matrix multiplication in Reduce stage:**
  - The algorithm uses the V-H matrix multiplication technique
- **Limitations:**
  - Increasing the number of partitions leads to better load balancing and smaller matrices for multiplication in the reduce phase, but for B-B it also causes an increase in duplication and data shuffled. Hence, as data size increases and a finer partitioning is needed, increasing partitioning beyond a limit degrades the performance

4. Experiments

Note: All experiments were run on 10K X 10K Matrices of varying densities on a 10 machine cluster of m4.large ec2 instances using multiple partition values.

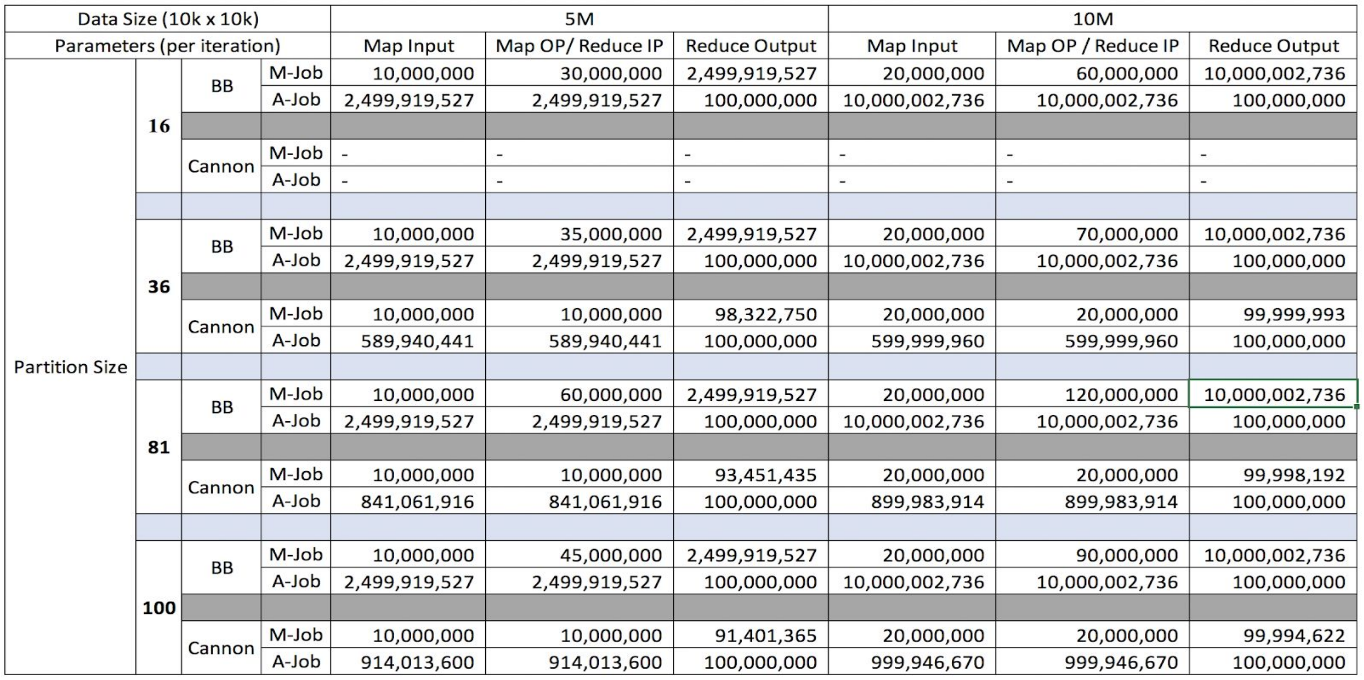
Keeping the goal of comparing the 2 approaches, the same datasets were used for both Simple B-B and Cannon's algorithm and the results were observed for blocks of different sizes. The 10K X 10K matrices which were multiplied were generated with 10% density (10M non-zero cells) and 5% density (5M non-zero cells). The partitioning choices were governed by Cannon's restrictive options. The number of blocks chosen was 16, 36, 81, and 100 to allocate Cannon's need for each submatrix to be a square matrix.The table below shows the findings with respect to the running times for the 2 algorithms.

Data Size	Partition Size							
	16 partitions		36 partitions		81 partitions		100 partitions	
	SBB	Cannon	SBB	Cannon	SBB	Cannon	SBB	Cannon
5M	59m	-	59m	59m	1hr	1hr 8m	1hr 2m	1hr 17m
10M	4hr 3m	-	3hr 52m	2hr 28m	3hr 58m	2hr 7m	4hr	1hr 59m

** Cannon Algorithm for 16 partitions was not feasible and the runtime exceeded our hard-limit of 3hrs. This is because, in our implementation of Cannon, the reduce phase is a sequential multiplication of submatrices. For 16 partitions, the size of the submatrices is in the order of 2,500 x 2,500.*

The performance times for both the algorithms heavily depends on the number of partitions chosen, as seen in the results. Increasing the number of partitions results in better load balancing and simpler reduce tasks, at the cost of increased duplication for Simple B-B, and an increased number of jobs for Cannon.
The performance times are more or less similar for smaller data samples, and the best for a given data sample requires fine-tuning the block sizes. The results also show that as the data size increases, Cannon's algorithm can be tuned by increasing the number of partitions to ensure competitive if not better performance times, whereas, for Simple Block-Block, the time reduces gradually before reaching a constant and does not reduce further. The most noteworthy observation however, is that with increasing data size (density in our experiments) as seen when the data sample is doubled from 5M to 10M records, Cannon's algorithm clearly outperforms Simple Block-Block.

4.1. Data Comparisons



Data Size (10k x 10k)				5M			10M		
Parameters (per iteration)				Map Input	Map OP/ Reduce IP	Reduce Output	Map Input	Map OP / Reduce IP	Reduce Output
Partition Size	16	BB	M-Job	10,000,000	30,000,000	2,499,919,527	20,000,000	60,000,000	10,000,002,736
			A-Job	2,499,919,527	2,499,919,527	100,000,000	10,000,002,736	10,000,002,736	100,000,000
		Cannon	M-Job	-	-	-	-	-	-
			A-Job	-	-	-	-	-	-
	36	BB	M-Job	10,000,000	35,000,000	2,499,919,527	20,000,000	70,000,000	10,000,002,736
			A-Job	2,499,919,527	2,499,919,527	100,000,000	10,000,002,736	10,000,002,736	100,000,000
		Cannon	M-Job	10,000,000	10,000,000	98,322,750	20,000,000	20,000,000	99,999,993
			A-Job	589,940,441	589,940,441	100,000,000	599,999,960	599,999,960	100,000,000
	81	BB	M-Job	10,000,000	60,000,000	2,499,919,527	20,000,000	120,000,000	10,000,002,736
			A-Job	2,499,919,527	2,499,919,527	100,000,000	10,000,002,736	10,000,002,736	100,000,000
		Cannon	M-Job	10,000,000	10,000,000	93,451,435	20,000,000	20,000,000	99,998,192
			A-Job	841,061,916	841,061,916	100,000,000	899,983,914	899,983,914	100,000,000
	100	BB	M-Job	10,000,000	45,000,000	2,499,919,527	20,000,000	90,000,000	10,000,002,736
			A-Job	2,499,919,527	2,499,919,527	100,000,000	10,000,002,736	10,000,002,736	100,000,000
		Cannon	M-Job	10,000,000	10,000,000	91,401,365	20,000,000	20,000,000	99,994,622
			A-Job	914,013,600	914,013,600	100,000,000	999,946,670	999,946,670	100,000,000

** Cannon Algorithm for 16 partitions was not feasible and the runtime exceeded our hard-limit of 3hrs. This is because, in our implementation of Cannon, the reduce phase is a sequential multiplication of submatrices. For 16 partitions, the size of the submatrices is in the order of 2,500 x 2,500.*

The primary difference between the two algorithms is seen in the table above. Simple B-B duplicates the incoming records based on the partitioning scheme used, reaching a high of 6 times the data input, as seen in the 81 partitions row of the table. This not only affects the first job but also the aggregation phase which aggregates approximately 100 times the number of records aggregated using Cannon's algorithm. These numbers back the experiments as it was seen that the aggregation phase for B-B takes the majority of the processing time. Cannon's as seen above does not duplicate records within an iteration and simply passes on the records to their respective blocks. The number of records for the aggregation phase increase with an increase in the number of iterations which in turn depends on the chosen number of partitions.

4.2. Speed-up Comparisons

For speed-up comparisons, we ran one execution for each algorithm on a P value of 100. The table below shows the running time and the speedup for each experiment:

Algorithm	**Data Size**	**Runtime on 10 machine cluster**	**Runtime on 20 machine cluster**	**Speedup**
SBB (P = 100)	**5 M**	1 hr 2 min	30 min	2.067
Cannon (P = 100)	**5M**	1 hr 17 min	42 min	1.833
Cannon (P = 100)	**10M**	1 hr 59 min	1 hr 5 min	1.831

4.3. Result Sample

The result data format also follows the same DOK format (as defined for the input datasets) for both algorithms.

5. Conclusion

The results show that there isn't a clear cut winner between the 2 algorithms. Both come with their own benefits which play on each other's shortcomings. The choice for an application depends highly on the dataset. For massive datasets Cannon's algorithm emerges as the clear winner. It has no data duplication at the cost of increased iterations which results in a larger amount of HDFS access. Whereas, Simple Block-Block has just a single iteration but has a large amount of data duplication. For very large datasets this duplication becomes very costly for shuffling and aggregation as supported by the results. Another noteworthy observation is that Cannon algorithm's runtime follows a hyperbolic curve as the number of partitions are increased (High for low values, reaching an optimum at a specific number, and then increasing again). For Simple Block-Block the runtime remains more or less consistent if the number of partitions is increased for small data. In this case, it would be fruitful to keep the smallest number of possible partitions, keeping memory constraints in mind, which would minimize data duplication. However, as data size increases, optimizing the Cannon algorithm is a better approach.

6. Future Work

During our research, we stumbled across the Hyper Sparse Gemma algorithm for sparse matrix multiplication, a sequential algorithm for matrix multiplication which finds intersecting values between the 2 matrices to be multiplied, and avoids checking the zero values altogether. This algorithm can be used to optimize the reduce phase of the Cannon algorithm, which currently mimics the 3-loop sequential multiplication, a computationally expensive set of operations.

7. References

[1] Anshul Gupta, Vipin Kumar. Scalability of Parallel Algorithms for Matrix Multiplication https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4134256

[2] Cannon's Algorithm Wikipedia. https://en.wikipedia.org/wiki/Cannon%27s_algorithm

[3] Ortega, Patricia. Parallel Algorithm for Dense Matrix Multiplication. CSE633 Parallel Algorithms Fall 2012. https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Ortega-Fall-2012-CSE633.pdf

[4] Zhiliang Xu. Lecture 6: Parallel Matrix Algorithms (part 3) https://www3.nd.edu/~zxu2/acms60212-40212/Lec-07-3.pdf

[5] Sparse Matrices, Wikipedia, Dictionary of Keys. https://en.wikipedia.org/wiki/Sparse_matrix#Dictionary_of_keys_(DOK)

8. Appendix

8.1. Output Links:

Name	**Output links (AWS S3 links)** ***login required**
Cannon-10m-100p-20cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions/10k-10m/100-best_P/?region=us-east-1
Cannon-10m-100p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions/10k-10m/100/?region=us-east-1&tab=overview
Cannon-10m-81p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions/10k-10m/81/?region=us-east-1&tab=overview
Cannon-10m-36p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions/10k-10m/36/?region=us-east-1&tab=overview
Cannon-5M-36P-10Cluster	https://s3.console.aws.amazon.com/s3/buckets/mr-project8895/10k-5m/36/?region=us-east-1&tab=overview
Cannon-5M-81P-10Cluster	https://s3.console.aws.amazon.com/s3/buckets/mr-project8895/10k-5m/81/?region=us-east-1&tab=overview
Cannon-5M-100P-10Cluster	https://s3.console.aws.amazon.com/s3/buckets/mr-project8895/10k-5m/100/?region=us-east-1&tab=overview
Cannon-5M-100P-20Cluster	https://s3.console.aws.amazon.com/s3/buckets/mr-project8895/10k-5m/

	100-20machines/?region=us-east-1&tab=overview
SBB-5m-16P-10Cluster	https://s3.console.aws.amazon.com/s3/buckets/mr-project8895/output/5mil-16/?region=us-east-1&tab=overview
SBB-5m-36P-10Cluster	https://s3.console.aws.amazon.com/s3/buckets/mr-project8895/output/5mil-36/?region=us-east-1&tab=overview
SBB-5m-81p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/mean-partitions/output/5mil-81-5/?region=us-east-1&tab=overview
SBB-5m-100p-20cluster	https://s3.console.aws.amazon.com/s3/buckets/mean-partitions/output/5mil-speedup-100/?region=us-east-1&tab=overview
SBB-5m-100p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/mean-partitions/output/5mil-100-3/?region=us-east-1&tab=overview
SBB-10m-16p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions-bb-sk/output/10m-16-final/
SBB-10m-36p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions-bb-sk/output/10m-36-final/
SBB-10m-81p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions-bb-sk/output/10m-81-final/
SBB-10m-100p-10cluster	https://s3.console.aws.amazon.com/s3/buckets/meanpartitions-bb-sk/output/10m-100-final/