Implementation of a Worst-Case Optimal Parallel Algorithms for Sparse Matrix Multiplication and Join-Aggregate Queries (CS848)

Allen Zhang[1]

Abstract

Sparse Matrix Multiplication has been an interesting problem for many years. A way to calculate Sparse Matrix Multiplication is by converting the calculation to an aggregate join query. Since these matrices are often large, the input for the aggregate join query is usually large too. Using a distributed system can allow mass computations that both allow less total time and larger input sizes. One such distributed system can be modeled by a massively parallel computation (MPC) model. Hu and Yi proposed a worst-case optimal aggregate join algorithm that can compute sparse matrix multiplication by assigning resources so that the load across servers is even during the computation. This project focuses on implementing their algorithm in Spark.

1. Introduction

A matrix with only a small percentage of nonzero elements is sparse. A practical definition would be an $n \times n$ matrix is sparse if its number of nonzero elements is in $O(n)$ (spm, 1973). Since this means the number of nonzero elements for sparse matrices is a lot less than $O(n^2)$, the standard $O(n^3)$-time algorithm may no longer be worst-case optimal under the semiring model.

A way to calculate sparse matrix multiplication is to first map nonzero entries in the two matrices to two relations R_1 and R_2. A join query can then find the multiplication result (Hu & Yi, 2020). This mapping can be defined as follows. Suppose we are given two arbitrary sparse matrix M_1 and M_2. For each nonzero entry with value w_1 at location (i, j) in M_1, we have a tuple (i, j, w_1) in $R_1(A, B, w)$. Similarly, for each nonzero entry with value w_2 at location (j, k) in M_2, we have a tuple (j, k, w_2) in $R_2(B, C, w)$. Then, we join $R_1(A, B, w)$ and $R_2(A, C, w)$ on column B to get $R_3(A, C, w)$. A caveat is on column w, where for each (i, j, w_1) joined with (j, k, w_2), we multiply the weights to get the tuple $(i, k, w_1 * w_2)$ in join results. Then we would group by the column A and C and aggregate over the column w in R_3 to get the desired result. An example is shown in Figure 1.

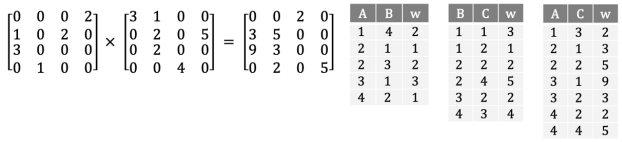

Figure 1. An example of using join to compute matrix multiplication

So a matrix multiplication can be seen as a join with N_1 tuples in R_1 and N_2 tuples in R_2, where N_1 and N_2 are nonzero entries in M_1 and M_2 respectively.

In the paper "Parallel Algorithms for Sparse Matrix Multiplication and Join-Aggregate Queries", Hu and Yi extended on this idea and proposed a worst-case optimal algorithm for sparse matrix multiplication in semiring with the Massively Parallel Computation (MPC) model. The MPC model usually consists of p servers, and the data are distributed across the servers evenly where each server holds N/p tuples. The computation proceeds in terms of rounds, where in each round, each server first receives messages from other servers and performs some local computation, then sends messages to other servers (Hu & Yi, 2020). The load is defined as the maximum message size received by any server in any round. Using this model, Hu and Yi proposed a worst-case optimal algorithm for any sparse matrix multiplication with $N_1, N_2 \geq 2$ tuples that can compute the aggregate join result in $O(1)$ rounds with load:

$$O(\frac{N_1 + N_2}{p} + \sqrt{\frac{N_1 N_2}{p}})$$

.

[1]University of Waterloo, Waterloo, Ontario, Canada. Correspondence to: Allen Zhang <allen.zhang@uwaterloo.ca>.

Submitted as part of the final project for CS848 in Spring term of 2024. Do not copy or distribute.

2. Method

We will now describe the implementation of this algorithm. We use Spark as our MPC framework. The algorithm by Hu and Yi has four distinct steps: compute data statistics, heavy-heavy, heavy-light, and light-light. For each of the steps, we will describe our implementation in Spark. The spark application is ran locally via spark-submit. We imitate the p servers with p partitions and p executors with 1 core each, so each partition will be sent to a single executor at each round of the computation. An initial balanced state between p servers is achieved via salting technique so that each partition will begin with N_1/p tuples from R_1 and N_2/p tuples from R_2. We start with partitioned RDDs $R_1(A, B, w_1)$ and $R_2(B, C, w_2)$, but since Spark requires the data to be key-value pairs in join operations, and since we are mostly dealing with using A and C as key values, we map the two RDDs to $R'_1(A, (B, w_1))$ and $R'_2(C, (B, w_2))$. Also since we are doing a lot of the same operations on both $R'_1(A, (B, w_1))$ and $R'_2(C, (B, w_2))$, we will refer them to $R'_i(X, (B, w_i))$ where $(X, i) \in \{(A, 1), (C, 2)\}$.

2.1. Step 1 - Compute Data Statistics

To compute the degree of x for each $x \in dom(X)$ of $R_i(X, (B, w_i))$, we use the RDD operation countByKey() on R_i, and this returns a map in the master server that contains $x \in X$ as keys and the count of tuples with x being the key in R_i. Then, based on $L = \sqrt{\frac{N_1 N_2}{p}}$, we can then use filter() on this map to get the x values that belong to the X^{heavy} set and X^{light} set. The map can be split into X^{heavy} count map and X^{light} count map in this process which only contains $x \in X^{\text{heavy}}$ and $x \in X^{\text{light}}$ with their count respectively. Then we use this information to further filter() on the RDD $R'_i(X, (B, w_i))$ to $R_i^{\text{heavy}\prime}(X, (B, w_i))$ and $R_i^{\text{light}\prime}(X, (B, w_i))$.

2.2. Step 2 - Heavy-Heavy

We can calculate the number of servers we need to allocate to each (a, c) pair with $a \in A^{\text{heavy}}$ and $c \in C^{\text{heavy}}$ as follows. Looping through the A^{heavy} count map, for each $a \in A^{\text{heavy}}$, we loop through each $c \in C^{\text{heavy}}$ count map, and based on the counts a_{degree} and c_{degree} we can calculate the server allocation as $p_{a,c} = \left\lceil \frac{a_{\text{degree}} + c_{\text{degree}}}{L} \right\rceil$. Then, we hash the join value B to get an exact partition number as a key for each tuple with a and c. The new RDD would be $R_i^{\text{heavyheavy}\prime\prime}(z, (X, (B, w_i)))$ where z is a partition number that may exceed p. Each $(x, (B, w_i))$ may be repeated for different z since there can be multiple (a, c) pairs. In addition, for each of (a, c) pair a set of unique partition numbers are reserved, and no other (a, c) pair can use those partition numbers. We keep track of the total number of partitions used and will only assign larger partition numbers in Step 3.

2.3. Step 3 - Heavy-Light

We can calculate the number of servers we need to allocate to each for each $x \in X^{\text{heavy}}$ given Y^{light} where $(X, Y) \in \{(A, C), (C, A)\}$ as follows. Looping through the X^{heavy} count map, for each $x \in X^{\text{heavy}}$, we use x_{degree} and $|Y^{\text{light}}|$ to calculate the server allocation as $p_x = \left\lceil \frac{|x_{\text{degree}}| + |Y^{\text{light}}|}{L} \right\rceil$. Then, we then use a sorting technique to find the exact partition number as a key for each tuple with x. This is invoked by sorting all tuples in X^{heavy} with $X = x$ and Y^{light} based on their B value and then finding $p - 1$ splitters that are broadcasted to all servers to assign the partition number for tuple with value x (Hu et al., 2019). The new RDD would be $R_i^{\text{heavylight}\prime\prime}(z, (X, (B, w_i)))$ and $R_i^{\text{lightheavy}\prime\prime}(z, (Y, (B, w_i)))$ where z is a partition number that may exceed p. The partition numbers would start from a partition number greater than the total partition number in Step 2, and also for each x, a set of unique partition numbers are reserved, and no other x value can use those partition numbers.

2.4. Step 4 - Light-Light

For light-light join, we first use the count maps of A^{light} and C^{light} to assign the groups evenly into $k = \lceil \frac{N_1}{L} \rceil$ groups and $l = \lceil \frac{N_2}{L} \rceil$ groups respectively. This can be done by sorting the count maps by count and assign a group number to the lowest total count group greedily. Then given the groups $A_1, A_2, ..., A_k$ and $C_1, C_2, ..., C_l$, we send a copy of the tuples in each group in a partition as follows: for each A_i with $i \in [k]$, for each $j \in [l]$, we assign it $i + k \times j$ partition (add the partition number to tuple). Then for each C_j with $j \in [l]$, for each $i \in [k]$, we assign it to the $i + k \times j$ partition (add the partition number to tuple). This allows each $A_i \bowtie C_j$ to be computed at a partition. We would get two new RDDs $R_i^{\text{lightlight}\prime\prime}(z, (X, (B, w_i)))$. In this step, the partition numbers start from 0 again.

2.5. Join Process

We first gather the result where for each $(X, i) \in (A, 1), (C, 2)$, we put all of $R_i^{\text{heavyheavy}\prime\prime}(z, (X, (B, w_i)))$, $R_i^{\text{heavylight}\prime\prime}(z, (X, (B, w_i)))$, and $R_i^{\text{lightheavy}\prime\prime}(z, (X, (B, w_i)))$ together to get $R_i^{\text{notlightlight}\prime\prime}(z, (X, (B, w_i)))$. We join $R_1^{\text{notlightlight}\prime\prime}(z, (A, (B, w_1)))$ with $R_2^{\text{notlightlight}\prime\prime}(z, (C, (B, w_2)))$ in Spark by joining by the partition number z that we added during the 4 steps, and then filter on B values from R_1 to R_2 to be equal to get the join result. This is due to Spark would hash the join key by default and shuffle them based on the hash, and this would allow the partition numbers to be respected. Also since $R_i^{\text{lightlight}\prime\prime}(z, (X, (B, w_i)))$ use non-unique partition numbers, we can join them in a separate join round just

like for the not-light-light case. Then, we union the two join results into a new RDD which would be of the form $R_3''(z, ((A, (B, w_1)), (C, (B, w_2))))$, and we can map it to $R_3'((A, C), w_1 * w_2))$. Then, we use reduceByKey to aggregate all $w_1 * w_2$ for each (a, c) in $R_3'(A, C)$ and finally get $R_3((A, C), \sum w_1 w_2)$ which gives the result of the matrix multiplication.

3. Testing

The testing of the algorithms are done with two custom simple data set and a few large random sampled datasets. Each dataset is made of two separate files including R1 and R2. Some of the other tests include comparing the light-light join results of the two and see if the results are the same. This helped figure out whether a problem in the implementation comes from light-light join or elsewhere. The default and worst-case optimal join algorithms without weights were tested with a simple dataset only. The first simple dataset only had a single heavy value for each A and C, while the second dataset had two heavy values for each A and C.

For default and worst-case optimal matrix multiplication algorithms, we attached each tuple in the two simple datasets with a random double between $[0, 1)$ to test both the matrix multiplication result locations and the aggregated values. More tests were done with randomly generated matrices. These matrices were generated in the following steps:

1. We first selected some sizes of N, namely $[1000, 10000, 100000, 1000000, 2000000, 4000000]$
2. For each N, we decided to use a $\frac{N}{4} \times \frac{N}{4}$ square matrix.
3. Then for each square matrix, we sample each coordinate of the nonzero entries with Normal distribution with mean and standard deviation equal to equals $\frac{N}{8}$ and $\frac{N}{16}$ respectively, and truncated at limits of the matrix size. This distribution is chosen since it covers all columns and rows, has peaks in the middle, but values are not so frequent that sampled coordinates will be the same as previous ones.
4. Finally, we make the coordinates distinct and attach a random double between $[0, 1)$ to the coordinates.

With these random sampled matrices, both the default and worst-case optimal algorithms were applied.

All join results are sorted, counted, and compared to ensure both algorithms output the same results and counts. It shows that the worst-case optimal algorithm implemented was correct to the extent of testing since all of the counts are the same, and the sorted results for simple cases were the same.

4. Limitations

The Spark library has been the most limiting for performance. It doesn't have in partition joins for local environments, and so we cannot create a partitioner to use before the join that allows us to directly join on the join column in each partition. Spark also doesn't have a feature where the data can be sent to multiple partitions for join; therefore, data need to be replicated for each partition before the join as well, making shuffling slower. This limitation may be addressed given an actual distributed environment, and maybe a master node that can send data in topics where different workers can subscribe to the different topics. The different workers will then sink the data to their own partition and perform join separately. This is more common in a real-time distributed environment as messages can be sent in real-time, with maybe an architecture with Kafka and Flink better suited so that the messages do not need to be replicated but rather each subscribed worker creates their own copy based on the messages they are listening to. Moreover, after receiving all messages, the workers can then perform the join with their own partition with Spark again.

Another limitation has been the time constraints. The experiments took a long time to run when the data was large. Rebuilding and testing the JAR files took a long time as well. We also did not figure out a good distribution to sample nonzero coordinates of matrices. Although the original paper (Hu & Yi, 2020) used Zipf distribution, it's difficult to achieve the same when we need to sample 2D coordinates that will be filled with nonzero entries since the coordinates can collide when both dimensions are using Zipf distribution, which would mean the joint distribution given that the coordinates are distinct are no longer Zipf. Normal distribution would be the best in this case, but finding the optimal variance is difficult. Ideally, an optimal variance would result in some A values and C values in the heavy set, but also not too dense where coordinates collision will happen very often. Unfortunately, we did not have time to gather enough results to make meaningful comparisons with the algorithm with the default join provided by Spark. We can use the code further to set up experiments to test the performances in the future.

Finally, the Windows environment was limiting too. Spark's scripts and setup favor the Linux system, and trying to get Spark running on Windows locally has been difficult. Some Spark monitoring applications could have helped run experiments and gather results, but we could not start them on Windows correctly.

5. Environments

The overall technical implementation is written in Scala. We mostly used the Spark library for partitioning and processing RDDs. Assisting libraries include Apache Log4j, Apache Commons Math3, and Apache Hadoop for logging, calculations, sampling distributions, and file writing. Hardware used to run the model is Intel i9-13900KF with 32 cores running on a Windows system with 64GB total system memory. The project code and running instructions are uploaded to the following GitHub repository: https://github.com/physics2001/Parallelized_Matrix_Multiplication

References

Chapter 1 preliminary considerations. In Tewarson, R. P. (ed.), *Sparse Matrices*, volume 99 of *Mathematics in Science and Engineering*, pp. 1–13. Elsevier, 1973. doi: https://doi.org/10.1016/S0076-5392(08)60527-5. URL https://www.sciencedirect.com/science/article/pii/S0076539208605275.

Hu, X. and Yi, K. Parallel algorithms for sparse matrix multiplication and join-aggregate queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'20, pp. 411–425, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371087. doi: 10.1145/3375395.3387657. URL https://doi.org/10.1145/3375395.3387657.

Hu, X., Yi, K., and Tao, Y. Output-optimal massively parallel algorithms for similarity joins. *ACM Trans. Database Syst.*, 44(2), apr 2019. ISSN 0362-5915. doi: 10.1145/3311967. URL https://doi.org/10.1145/3311967.