Matrix Factorization with Spark

HingOn Miu
hmiu@andrew.cmu.edu
Carnegie Mellon University

1. Introduction

This project uses Spark to parallelize an iterative machine learning algorithm – matrix factorization. Spark is very fast as it is designed to store all intermediate data in memory for later stages of computation. It is often programmers' responsibility to design a parallel Spark algorithm that minimizes network I/O and utilizes all resources to their fullest potentials. Therefore, I have one vital parallelization goal in this project: maximize computation time and minimize network I/O latency for each executor core.

Parallelization gets noticeably harder when dealing with larger datasets since there is often harsher constraints on resources. Now, let's start discussing the possible parallelization of each code segment in matrix factorization.

2. Parallelization of Code Segments

2.1. blockify_data

In the iterative version, blockify_data looks for the maximum userid and movieid sequentially (one core does all the work), while my code first reads the dataset into #-of-cores partitions and computes the maximum with Spark max. As evident by Spark Event Timeline, all cores in my code participate in computing maximum in local partitions and merge the result. Also, I notice that the ratings dataset is read-only throughout the entire matrix factorization computation, and the dataset is read frequently to factorize W rows and H rows. So, I believe that Spark broadcast variable is more efficient than RDD in this scenario since the entire ratings dataset broadcast sits in memory of each executor node instead of only a subset of the dataset (local partitions). Therefore, the entire ratings dataset is only needed to be transferred once in the network, and this eliminates network I/O of shuffling partitions of the ratings dataset in each factorization computation if RDD were used to store the ratings dataset.

I experimented my theory on 1M ratings dataset and observed at least 2x speedup. I used lookup to fetch each block of the dataset matrix, and each RDD lookup requires huge network I/O of broadcasting the lookup result. On the other hand, there is no network I/O observed at all for using broadcast variable to store the dataset during the entire iteration of matrix factorization.

2.2. initialize_factor_matrices

With the same reasoning of minimizing network I/O in shuffling partitions, I decide to make two copies each of W rows and H rows such that one copy is RDD for computation and one copy is broadcast variable for quick lookups. Hence, the two copies are cached in memory before each iteration and unpersisted at the end of each iteration of matrix factorization. I again experimented my theory on 1M ratings dataset and observed at least 2x speedup. I used lookup to fetch each block of the W/H rows, and each RDD lookup requires huge network I/O of broadcasting the lookup result. On the other hand, there is no network I/O observed at all for using broadcast variable to store the W/H rows during the entire iteration of matrix factorization. However, there is a huge burst of network I/O at the end of the iteration when the new broadcast variables of W rows and H rows are generated.

2.3. sgd_block_by_block

In the iterative version, each block in matrix is traversed sequentially, and this is obviously a waste of resources because only the executors that hold the corresponding blocks of W rows and H rows are doing the computation. As evident by Spark Event Timeline, all other executors remain in idle.

To fix this issue, I decide to parallelize the matrix traversal. I notice that sgd_on_one_block updates the corresponding blocks of W rows and H rows at the same time. This greatly limits the potential parallelism because if my code were to factorize an entire row of blocks in the matrix, my code would have to somehow atomically update the single block of W rows to avoid race condition. Therefore, I eventually split sgd_on_one_block into two functions: one function updates the block of W rows and one function updates the block of H rows. Then, my code computes the matrix row by row and updates an entire row of blocks of H rows in parallel, then my code computes the matrix column by column and updates an entire column of

blocks of W rows in parallel. Since there are #-of-cores partitions each in W rows and H rows, Spark Event Timeline indeed shows that there is no idle core during matrix traversal. As mentioned in previous section, since I have a copy of W rows, H rows and dataset as broadcast variables, I manage to eliminate the network I/O of throughout the entire matrix traversal.

2.4. evaluate_block_by_block

Similar to sgd_block_by_block, the iterative version traverses each block in the matrix sequentially. Moreover, I notice that there is no update to any block of W rows and H rows. So, my code simply gets the Cartesian product of W rows and H rows, and joins with the rating dataset such that compute all blocks in the matrix can be computed in parallel. I experimented my theory on 1M ratings dataset and observed at least 2x speedup. Similar to sgd_block_by_block, I observe no network I/O at all from Spark History Server. There is also no new broadcast variable creation, and so the entire function runs locally in each executor core.

2.5. saveAsTextFile

It is required that the ultimate outputs of W rows and H rows are stored in separate single HDFS files, my code has to sort them in correct order and group the blocks into one single partition. I attempted to use sortByKey to generate one single partition with sorted blocks, but I noticed from Spark History Server that there is many network I/O involved in sorting the partitions globally. Therefore, I avoided sorting by using partition index. Since the default ordering of partitions in each RDD accords to their partition index, I created the RDDs of W rows and H rows with sorted blocks mapped to partition index. Without sortByKey, I use coalesce to generate one single partition in the end, and I notice there is much less network I/O compared to sortByKey on 1M ratings dataset from Spark History Server.

3. Scalability

3.1. num_partitions

As mentioned above, the number of partitions is set to be the number of logical cores of all executor cores, and so each executor core can work independently on its local partitions. So, my code has a global variable num_partitions that controls the partitioning of all RDDs, and my code can adapt to any number of executor cores by updating num_partitions.

3.2. Memory

As the executor instance type changes, my code sets spark.driver.memory, spark.executor.mem-ory, and spark.python.worker.memory to replace the default configuration to fully utilize all resources. My code also sets spark.akka.frameSize such that the maximum memory each task uses can be adjusted according to the input rating datasets and number of partitions.

3.3. Network

To further reduce network I/O, I increase the value of spark.network.timeout and spark.akka.heart-beat.interval so that there is less flood of heartbeat messages in the network.

4. Conclusion

In other words, I have managed to minimize network I/O as little as possible such that each executor core can work on is local partitions most of the time. For next steps, I would want to study more of the Spark configuration since there are so many parameters to tune. Also, I would love to try to run my code on different instance types and cluster sizes to determine the best cluster configuration of computing matrix factorization with minimum amount of resources.

As shown in Spark History Server, there are only three bottlenecks in my program that incur huge network I/O. The first bottleneck is when generating the broadcast variable of the ratings dataset, and it is proved that this is a good tradeoff since it greatly reduces the network I/O of ratings lookup in later stages. The second bottleneck is when generating new broadcast variable for W rows and H rows after each iteration of matrix factorization. Again, with the same reasoning, this is a good tradeoff since it greatly reduces the network I/O of W/H rows lookup in later stages. The last bottleneck is the coalescing all partitions into one single partition to output W rows and H rows into HDFS files. Ideally, since my code already has the broadcast variables of the final version of W rows and H rows in the end, they already sit in memory of every executor node and I should be able to easily store them into a single HDFS file. However, after some extensive research, there is no easy way to do that without using other libraries. I believe I may still overlook some shortcuts that can write an in-memory string into a single HDFS file such that I can eliminate the network I/O completely in coalesce.