Research Article

Ying Wang* and Korhan Cengiz

Implementation of the Spark technique in a matrix distributed computing algorithm

https://doi.org/10.1515/jisys-2022-0051
received October 29, 2021; accepted February 07, 2022

Abstract: Two analyzes of Spark engine performance strategies to implement the Spark technique in a matrix distributed computational algorithm, the multiplication of a sparse multiplication operational test model. The dimensions of the two input sparse matrices have been fixed to 30,000 × 30,000, and the density of the input matrix have been changed. The experimental results show that when the density reaches about 0.3, the original dense matrix multiplication performance can outperform the sparse-sparse matrix multiplication, which is basically consistent with the relationship between the sparse matrix multiplication implementation in the single-machine sparse matrix test and the computational performance of the local native library. When the density of the fixed sparse matrix is 0.01, the distributed density-sparse matrix multiplication outperforms the same sparsity but uses the density matrix storage, and the acceleration ratio increases from 1.88× to 5.71× with the increase in dimension. The overall performance of distributed operations is improved.

Keywords: spark technology, distributed, matrix operation, sparse matrix, dense matrix

1 Introduction

In recent years, with the development of communication technology, especially the rapid development of mobile, internet, video, voice, image, and other data are growing rapidly [1]. In order to dig deeper into the value contained in the massive data, Click-Through-Rate estimation, recommendation algorithm, image recognition technology, speech recognition technology, etc., are widely used. With the widespread application of these algorithms on large-scale data, the computing power of the existing single machine is limited by the bandwidth and the limited computing power of a single CPU, and hence, it can no longer meet the needs of massive data processing. Matrix operations are a basic mathematical tool, and are the foundation of many machine learning and data mining algorithms, commonly used matrix operations have high algorithm complexity [2]: for example, matrix multiplication commonly used in deep learning, matrix factorization commonly used in natural language processing and recommendation systems, etc. Massive data promote the rapid development of various parallel algorithms, especially the wide application of Hadoop ecosystem, parallel algorithms based on the Hadoop ecosystem are constantly emerging [3]. The use of traditional methods on general computers cannot solve large-scale numerical calculation problems, massive data promote the rapid development of various parallel algorithms [4]. Initially, parallel computing required a dedicated computer system, such as parallel vector processing machines, massively parallel processor, and distributed shared memory processor. These dedicated computer systems are

* **Corresponding author: Ying Wang,** Department of Information Engineering, Tianjin Maritime College, Tianjin, 300350, China, e-mail: YingWang5@126.com
Korhan Cengiz: College of Information Technology, University of Fujairah, Fujairah, United Arab Emirates; Department of Electrical – Electronics Engineering, Trakya University, 22030, Edirne, Turkey, e-mail: korhancengiz@uof.ac.ae

expensive and not suitable for large-scale use. With the continuous improvement in workstation performance and the declining price, there are also high-speed and cheap networks constantly appearing, and the use of ordinary workstations to form a cheap parallel computing system appeared [5]. This system can make full use of the resources of each workstation, unified scheduling processor, and realize efficient parallel computing. In this kind of workstation cluster, users need to explicitly send and receive messages to achieve data exchange between processors, call it messaging. Based on this, parallel programming environment, MPI4, appeared in the 1990s, it is a standard for messaging interface used to develop parallel computing programs based on message passing, its purpose is to provide users with a portable, practically usable, and efficient messaging interface library. Xiong et al. proposed a new generation of parallel computing framework, Spark. As shown in Figure 1, the framework is dedicated to proposing a unified programming model, providing a better programming interface, at the same time, supporting batch jobs, interactive jobs, iterative jobs, and stream processing jobs. While providing performance comparable to proprietary systems, it can also reduce learning costs and maintenance costs [6].

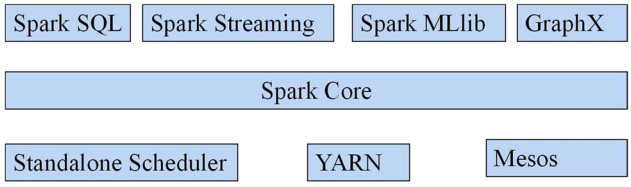

Figure 1: The Spark system component diagram.

2 Literature review

Heidari and others believe that MapReduce implements Shuffle in order to reduce memory usage, extensive use of external sorting to achieve the aggregation of Value of the same Key, and there will be a lot of disk IO operations [7]. Wang and others proposed a new data abstraction resilient distributed dataset (RDD), which reduced disk IO and brought a huge performance improvement. Spark can also make full use of the various resources already in the Hadoop ecosystem, no technology stack migration is required. RDD is an immutable, partitioned, distributed, and abstract distributed data collection [8].

Guo et al. examined the performance bottleneck of MLlib (ML's official Spark package) in detail, focusing on its implementation of stochastic gradient descent (SGD) under the training of multiple ML models. We prove that the performance disadvantage of Spark is caused by implementation problems, and the Spark performance can be significantly improved by utilizing the famous "model averaging" (MA) technology in distributed ML. Further, the application of MA in training a potential Dirichlet allocation (LDA) model in Spark is demonstrated. Not intrusive, only a small amount of development work. The results of the experimental evaluation show that the MA-based SGD and LDA versions are several orders of magnitude faster than the similar versions without the MA [9].

SparkBLAST, a distributed parallel BLAST method, was designed based on the big data technology Spark, by Wang et al. Under the memory computing framework Spark, SparkBLAST identifies the sequence alignment task, divides the sequence dataset, and compares the sequence data. The Apache Hadoop YARN is used for task scheduling and resource allocation. Finally, the SparkBLAST was experimentally compared with an independent BLAST. It is shown that the SparkBLAST achieves an acceleration ratio of 3.95 without sacrificing accuracy. That is, SparkBLAST is computationally more efficient than independent BLAST. The results provide an efficient sequence alignment tool for bioinformatics researchers [10].

In cloud computing and big data systems, delayed detection and manual resolution of performance anomalies can result in performance violations and financial penalties. Based on this, Alnafessa et al. proposed an anomaly detection method based on artificial neural network, which is applicable to The Apache Spark memory processing platform. Apache Spark is widely used in the industry due to its high

speed and versatility. However, there is still no comprehensive performance exception detection method applicable to this platform. Alnafessa et al. proposed a method driven by artificial neural network to rapidly filter Spark log data and operating system monitoring indicators, and accurately detect and classify abnormal behaviors based on the characteristics of Spark elastic distributed dataset. The method is evaluated using three popular machine learning algorithms, decision trees, nearest neighbors, and support vector machines, as well as four variables that consider different monitoring datasets. The results show that this method is superior to other methods, usually achieving 98–99% F scores, and provides higher accuracy than other techniques in detecting the period and type of anomaly occurrence [11].

Compared with traditional statistical methods, machine learning algorithms have been widely used in load forecasting to obtain better accuracy. However, with the huge growth of data scale, complex models need to be established, which requires a big data platform. By maximizing the effective utilization of cluster nodes, available computing resources can be optimized and utilized efficiently. In the process of smart grid big data, parallel computing is needed to realize the optimal utilization of resources. Zainab et al. carried out experiments on load prediction in multi-AMI environment by using master-slave parallel computing mode. A parallel job scheduling algorithm based on Apache Spark in multi-energy data source environment is proposed. An efficient Spark job submission resource utilization strategy is proposed to reduce job completion time. The optimal value of the cluster clusters the data to reduce the computation time. Multiple tree-based machine learning algorithms were tested for parallel computation to evaluate performance with tunable parameters on real datasets. Three years of real data from 1,000 distribution transformers in Spain were used to demonstrate the trade-off between accuracy and processing time of the method [12].

Data reduction or summarization techniques allow for a reduced representation of a dataset that has a much smaller collective volume but closely preserves the integrity of the original data. Clustering hierarchical clustering algorithm has little advantage in summarizing data. It can be as simple as generating a specific level of summary (in the form of a clustering pattern) with a simple tweak, or it can be paralyzed. Apache Spark is a data processing framework that can quickly perform processing tasks on very large datasets. It is a standard tool for analyzing big data. In terms of data processing, Spark can distribute data processing tasks across multiple machines. Spark runs on the DISTRIBUTED File System (HDFS) and YARN (Resource Management) of Hadoop to access HDFS files and efficiently utilize network resources. In order to achieve high performance and scalable data analysis technology in the Spark environment, the cost of phase conversion, narrow conversion, and wide conversion, I/O, and network must be considered. Moertini and Ariel developed a data summarization technique using clustering algorithms on Spark. To avoid bias in the results, records in a given big data are randomly divided into bags of datasets stored in elastic RDD partitions on the work machine. To reduce network and I/O costs, an extensive transformation is adopted that involves data transformation across the network. In addition, RDD partitions are then processed locally to generate cluster patterns from work tasks. Functions with complex calculations are designed as Spark parallel tasks. We ran a series of experiments on a Spark cluster by varying data sizes (5–20 Gb), machine kernel usage (10–50), and application variables (data split and Max object/tree). The results showed that the method is scalable and effective. Execution time is largely determined by parallel tasks running locally on the worker [13].

Clustering is one of the most important unsupervised machine learning tasks. It is widely used in intrusion detection, text analysis, image segmentation, and other problems. Subspace clustering is the most important method in high dimensional data clustering. In order to solve the clustering problem of parallel subspaces of high-dimensional big data, Xiao and Hu proposed a parallel subspace clustering (PSubCLUS) algorithm based on Spark, inspired by the classical subspace clustering algorithm, SubCLU. Spark is the most popular parallel processing platform for big data. PSubCLUS uses the RDD log base provided by Spark for distributed storage. The two main execution stages of the algorithm, one-dimensional subspace clustering and iterative clustering, can be executed in parallel at each working node of the cluster. PSubCLUS also uses a repartitioning approach based on the number of data points for load balancing. Experimental results show that PSubCLUS has good parallel acceleration and ideal load balancing effect, and is suitable for solving the clustering problem of parallel subspace of high-dimensional big data [14].

Spark is a more efficient distributed big data processing framework following Hadoop. It provides users with more than 180 adjustable configuration parameters, and it is a challenge to automatically select the optimal configuration to make the Spark application run efficiently. The key to solving these problems is the ability to predict the performance of Spark applications in different configurations. Cheng et al. proposed a new adaboost-based approach that can efficiently and accurately predict the performance of a given application with a given Spark configuration. Adaboost is used to build a set of performance models for Spark at the stage level. The proposed approach was evaluated on six typical Spark benchmarks with five input datasets. Experimental results show that the prediction error and cost of the proposed method are less than those of the previous method [15].

RDD only supports coarse-grained operations, and does not support fine-grained operations, for example, RDD does not support updating one of its elements.

In the optimization of dense matrix multiplication, this article optimizes the call method of local native library to reduce frequent data transmission and improve the degree of operation concurrency. Based on the existing conversion method between the distributed block matrix and row matrix in Spark, a more efficient conversion method is proposed, and some system components of Spark are further modified to provide operators to reduce the data disk read and write load in the matrix multiplication join step. Through these optimizations, the overall performance of dense matrix multiplication is significantly improved.

3 Method

3.1 Analysis of two execution strategies based on Spark execution engine

Matrix A with the dimensions $m \times n$ contains $M_b \times K_b$ sub-blocks, and matrix B with dimensions $n \times k$, contains $K_b \times N_b$ sub-blocks.

(1) Reuse methodology manual (RMM): In the RMM execution strategy, there is only one shuffle step. In order to get the final result matrix C, input sub-matrix blocks $A_{i,k}$ and $B_{k,j}$ need to generate multiple copies. In the flatMap phase of RMM, each sub-block of matrix A needs to generate N_b copies, similarly, each sub-block of matrix B needs to generate M_b copies, the amount of data that needs to be shuffled at this stage is $N_b|A| + M_b|B|$. Perform multiplication between sub-blocks in the reduceByKey stage of RMM, unlike Competency of Project Management Model (CPMM), at this time, $M_b \times N_b$ tasks can be executed at the same time, and each task needs to perform K_b sub-block matrix multiplications in series and accumulate to obtain the final sub-block $C_{i,j}$;

(2) CPMM: In this execution strategy, there are two shuffle stages. In the map step of the first shuffle stage, input matrices A and B are connected by key k and written to disk, aggregating the combination $A_{i,k}$ and $B_{k,j}$ of the sub-block matrix, the data volume of shuffle in this stage is $|A| + |B|$. The next stage performs the cross product, namely $P_{i,j}^k = A_{i,k}B_{k,j}$, since each matrix block has been combined by the key k connection, therefore, at most Kb tasks can perform matrix multiplication between sub-blocks at the same time. These intermediate results are then combined according to the key (i, j) to get $P_{i,j}^k$; write to disk, the amount of data required to mix in this step is $K_b\,|C|$ is. In the final reduce stage, the final result matrix $C_{i,j} = \Sigma_k p_{i,j}^k$ is obtained through the reduceByKey operator. Through the above analysis, it can be seen that when CPMM performs sub-block matrix multiplication, only K_b tasks are executed concurrently, in other words, the concurrency is only K_b, the concurrency of RMM can reach $M_b \times N_b$. For the same matrix size, since the computational cost of accumulating the intermediate result sub-matrix is much less than performing sub-matrix multiplication, higher concurrency, when the cluster resources are met, tends to bring better performance [16–18]. So inspired by this, an additional step of accumulating sub-matrices is introduced to greatly increase the concurrency of sub-matrix multiplication. So on the basis of RMM, an optimized execution strategy Competency of Reuse methodology manual (C-RMM) is proposed, which is described in detail as follows: In this strategy, there are two shuffle stages, the first

shuffle stage is similar to RMM, each sub-block matrix in matrix A and matrix B needs to generate N_b and M_b copies, respectively, but since there is no need to distribute related $A_{i,k}$ and $B_{k,j}$ to the same task in order to obtain the resulting sub-block matrix $C_{i,j}$, C-RMM can simultaneously exist at most $M_b \times K_b \times N_b$ tasks to perform sub-block matrix multiplication in parallel. Finally, a shuffle stage is introduced to accumulate the intermediate results to get the final matrix. In fact, after analysis, we can find that C-RMM is equivalent to RMM or CPMM in two special situations. When inputting the three dimensions of matrices A and B, when the latitude k is much smaller than the dimensions m and n, the distribution of sub-block matrices of matrices A and B can be deduced from $M_b \times K_b \times N_b$ to $M_b \times 1 \times N_b$. In this situation, since the dimension k is not divided, therefore, there is no need for an additional shuffle stage to aggregate and accumulate the intermediate result matrix, therefore, C-RMM and RMM are theoretically equivalent in this special case. When inputting the three dimensions of matrices A and B, when the latitude k is much larger than the dimensions m and n, this scenario can be visually depicted as a very flat matrix A and a very thin and tall matrix B performing multiplication [19–21]. At this time, the distribution of sub-block matrices of matrices A and B can be deduced from $M_b \times K_b \times N_b$ to $1 \times K_b \times 1$. In this case, originally, it is necessary to generate A and B copies of each sub-block matrix in matrix N_b and matrix M_b, respectively, At this time all are 1, therefore, C-RMM and CPMM are theoretically equivalent in this special case. In fact, this scenario is very easy to appear in the matrix transpose multiplication ($H^T \cdot H$), especially when H is a thin and tall matrix. In addition to the above three matrix multiplication execution strategies, since Spark supports broadcasting variables from the driver side (broadcastvariable) [22–24], when two matrices are input, and when the scale of one of them is small, the small matrix can be broadcasted, making each executor node store the small matrix in the process, thus avoiding the shuffle phase, and reducing the read and write of the disk [25–27]. This strategy is called mapside matrix multiplication (MapMM; the network transmission overhead of this strategy is the number of executors $\times$ min($|A|$, $|B|$).

For the CPMM strategy, on the one hand, increasing K_b can improve the multiplication concurrency of the sub-matrix, but on the other hand, the larger the K_b, more data need to be written to the disk after this multiplication stage. Because of the shuffle process, a large amount of data writing may cause some threads to fail to get enough memory, therefore, some data will be spilled to the disk to wait for resource allocation , these behaviors will cause great pressure on the Java Virtual Machine (JVM) and affect the overall performance [28–30]. Assuming a 12-node Spark cluster, with each computing node having 16 logic computing cores, then the maximum number of concurrency in the cluster is 192. To facilitate discussion, suppose the dimensions of the input matrices A and B are the same, in this way, the global shuffle data volume can be expressed in multiples of $|A|$. When these three strategies reach the same degree of concurrency, the global shuffle data volume of the C-RMM strategy is less.

In addition to the above three matrix multiplication execution policies, since Spark supports broadcasting variables from the driver side (broadcast variable, when the two input matrices are small), the small matrix can be broadcast to make the small matrix to be stored within each executor node, thus avoiding the shuffle stage and reducing the read and write of the disk [31,32]. This strategy is called MapMM (matrix multiplication for map), and the network transmission overhead is The number of executors $\times$ min($|A|$, $|B|$). Each-step cost analysis of the above four strategies is summarized in Table 1.

3.2 Efficient conversion of distributed row matrix and block matrix

For the matrix operation library for machine learning, since the distribution of data can naturally be expressed as a row matrix, so how to efficiently process the conversion between the distributed row matrix and the block matrix is particularly important. SparkMLlib emits the original every row vector into a large number of coordinates in the format of (i, j, v). This will not only bring about twice the unnecessary data redundancy for data-parallel Spark-like big data systems based on JVM but this method will bring a large number of small objects in the shuffle phase, frequent disk reads and writes can cause great pressure on the

Table 1: Analysis of different matrix multiplication execution strategies

Execute a strategy	Number of concurrent writes to the disk	Shuffle data amount of the first step	The degree of concurrency to perform sub-block matrix multiplication	Shuffle data for the second step	The concurrency of the summation sub-block matrix
CPMM	$\min(M_b \times K_b + K_b \times N_b, P)$	$\|A\| + \|B\|$	$\min(K_b, P)$	$K_b\|C\|$	$\min(M_b \times N_b, P)$
RMM	$\min(M_b \times K_b + K_b \times N_b, P)$	$N_b\|A\| + M_b\|B\|$	$\min(M_b \times N_b, P)$	Nothing	Nothing
C-RMM	$\min(M_b \times K_b + K_b \times N_b, P)$	$N_b\|A\| + N_b\|B\|$	$\min(M_b \times N_b + K_b \times N_b, P)$	$K_b\|C\|$	$\min(M_b \times N_b, P)$
MapMM	Nothing	Number of executors $\times \min(\|A\|, \|B\|)$ (the amount of broadcast data)	$\min(M_b \times K_b + K_b \times N_b, P)$	Nothing	Nothing

P represents the total number of logical cores of the cluster.

system. The additional problem is that after this inefficient conversion, SparkMLlib treats each sub-block matrix as a sparse matrix, as a result, the subsequent matrix operations cannot call the native library.

As can be seen from Table 1, for CPMM strategy, K_b is increased on the one hand, improving the multiplicative concurrency of the submatrices, but on the other hand, the larger the K_b, more data need to be written to the disk after that multiplicative phase. Because large amounts of data writing during shuffle may cause some threads to get enough memory, spilling (spill) part of the data to disk for resource allocation can put great pressure on JVM and affect overall performance. The comparison of the three matrix multiplication strategies involving shuffle, namely CPMM, RMM, and C-RMM, can be intuitively illustrated in Figure 2. Assuming the Spark cluster of 12 nodes and 16 logical operation cores per calculation node, the maximum number of concurrency in the cluster is 192. To facilitate the discussion, let the size of the input matrices A and B be the same, so that the global shuffle data volume can be expressed by the multiple of $|A|$, which is the vertical axis of Figure 2, and the horizontal axis represents the concurrency in the sub-block matrix multiplication phase. From Figure 2, it is intuitive that the three strategies have less global shuffle data for the C-RMM strategies when reaching the same concurrency.

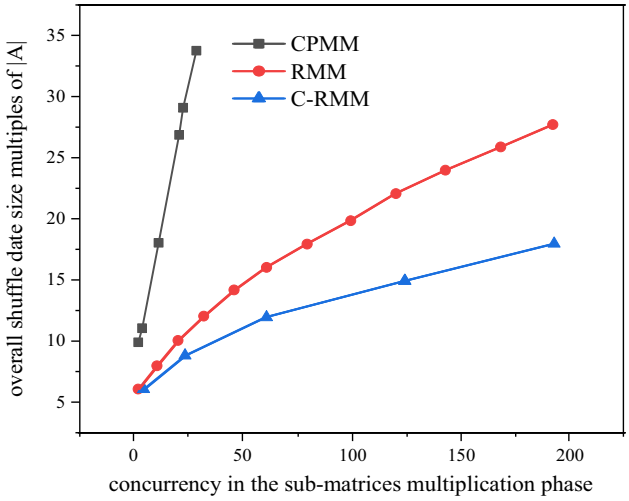

Figure 2: The shuffle data volume of the three multiplication execution strategies, vs An intuitive comparison of the concurrency (192 logos in the cluster Q).

4 Results and analysis

4.1 Experimental design and result analysis of distributed sparse-sparse matrix multiplication operation

First, fix the dimensions of the two input sparse matrices to be 30,000 × 30,000, change the input matrix density, the performance comparison between sparse matrix multiplication operations and related operations stored in the dense matrix format and calling local native libraries are done, and the experimental results are shown in Figure 3. Similarly, the density of the fixed matrix is the usual density of 0.01 in the recommended system data, and on changing the dimension of the input matrix, the relative performance comparison chart is shown in Figure 4.

From Figure 3, we can observe the use of C-RMM strategy and optimizing the distributed sparse-sparse matrix multiplication of the stand-alone matrix library is significantly better than the distributed dense matrix multiplication using the same execution strategy. For the input of the latter dense matrix, although the density of the matrix is low, due to the dense matrix storage method, the amount of serialized data for

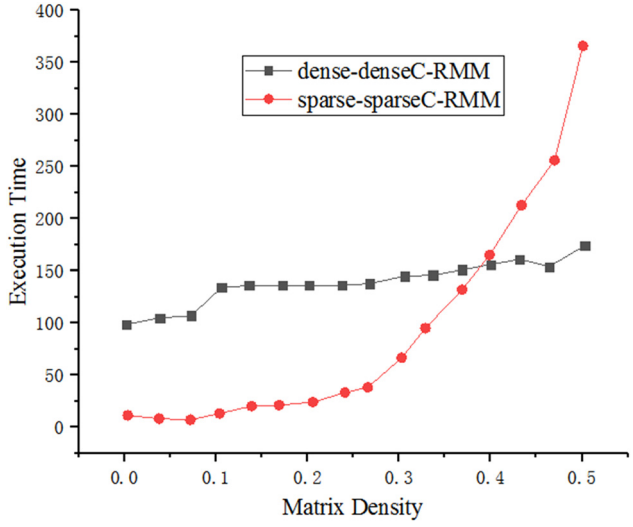

Figure 3: The performance of distributed sparse-sparse matrix multiplication varies with matrix density.

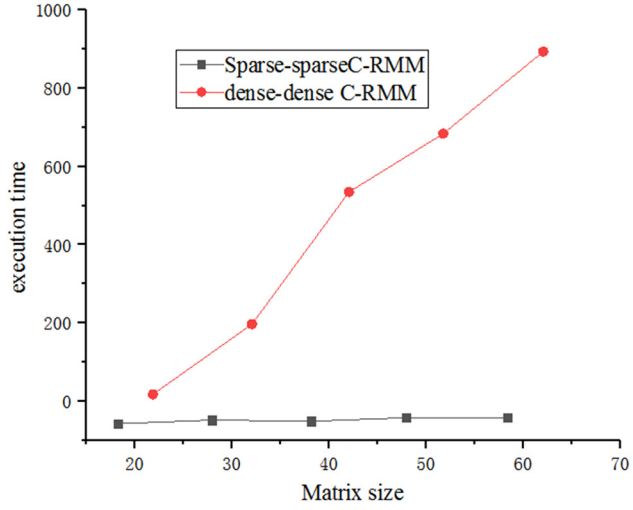

Figure 4: The performance of distributed sparse-sparse matrix multiplication varies with matrix dimensions.

shuffle is still very large. When the density reaches about 0.3, the performance of the original dense matrix multiplication can be better than the sparse-sparse matrix multiplication implemented in this chapter. This is also basically consistent with the calculation performance relationship between the implementation of sparse matrix multiplication and the call of the native library in the stand-alone sparse matrix test. Zhou et al. presented an improved cost estimate for computing the inverse problem of a univariate polynomial $N \times N$ matrix. A deterministic algorithm with worst-case complexity (N3s)1 + O (1) field operation is proved, where $S \geq 1$ is the upper bound of the average column degree of the input matrix. Here the " +o(1)" in the exponent indicates the absence of the factor C1 (log ns)c2 for the positive real constants C1 and C2. As an application, it is shown how to calculate the maximum invariant factor of the input matrix in the field operation of (nωs)1 + O (1), where ω is the exponent of matrix multiplication [33]. Figure 4 shows that when the density of the fixed matrix is 0.01, as the size of the matrix increases, the performance of the implemented distributed sparse-sparse matrix multiplication is far better than the same sparseness; however, the dense matrix format is used for storage and operation strategy.

Combining the above two points, it shows that the performance of the design and implementation of the sparse-sparse matrix is much better than the performance of the original dense matrix to deal with such problems, especially when the density of the sparse matrix is below 0.3, the advantages are more obvious.

4.2 Experimental design and result analysis of distributed dense-sparse matrix multiplication operation

First, fix the dimensions of the two input matrices to be 30,000 × 30,000, one of them is a dense matrix, while the other is a sparse matrix. Change the density of the sparse matrix, multiply dense-sparse matrices, and store in dense matrix format, and call the relevant operations of the local native library for performance comparison. The experimental results are shown in Figure 5. Similarly, fix the dense density of the sparse matrix to 0.01, change the dimension of the input matrix, and the relative performance comparison chart is shown in Figure 5.

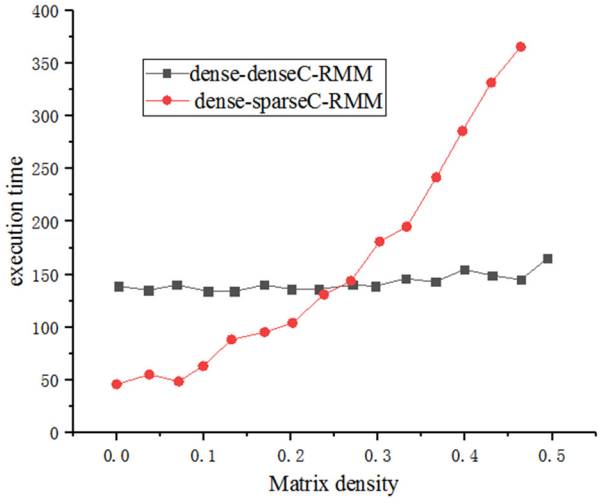

Figure 5: The performance of distributed dense-sparse matrix multiplication varies with matrix density.

Figure 5 shows the adoption of C-RMM strategy. The distributed dense-sparse matrix multiplication that optimizes the stand-alone matrix library is obviously better than the distributed dense matrix multiplication that adopts the same execution strategy. For the input of the latter dense matrix, although the density of the matrix is low, due to the dense matrix storage method, the amount of serialized data for shuffle is still very large. When the density reaches about 0.1, the performance of the original dense matrix multiplication can be better than the implemented sparse-sparse matrix multiplication. This is also implemented with the dense-sparse matrix multiplication in the stand-alone sparse matrix test and the relationship between the pros and cons of the computing performance of calling the e native library is basically the same [34,35].

Figure 6 shows that when the density of the fixed sparse matrix is 0.01, as the size of the matrix increases, the performance of the implemented distributed dense-sparse matrix multiplication is better than the same sparseness but using the related operations of dense matrix storage, and as the dimension increases, the speedup has been increased from 1.88× to 5.71×.

This paper confirms the implementation of sparse distribution with high density of sparse matrix multiplication, and performs the relevant actions using previous comparisons using sparse matrix multiplication distribution and density matrix preservation. The experimental results show that when the thickness of dense and sparse distribution is below 0.3, the performance is better than that ofdense and sparse

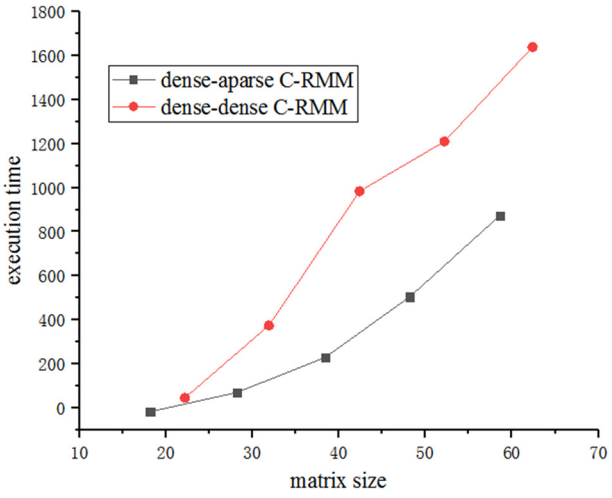

Figure 6: The performance of distributed dense-sparse matrix multiplication varies with matrix dimensions.

matrix multiplication; when the sparsity is less than 0.1, the performance of distributed dense-sparse matrix multiplication is better than that of original dense matrix multiplication.

5 Conclusion

The realization of Spark technology in matrix distributed computing algorithm was proposed, the two execution strategies based on the Spark execution engine are analyzed, and the conversion of efficient distributed row matrix and block matrix was analyzed. The distributed sparse-sparse matrix multiplication operation experiment is designed, and as a result of the experiment, the dimensions of the two input sparse matrices are fixed to be 30,000 × 30,000, the input matrix density is changed, and the performance of sparse-sparse matrix multiplication operation stored in dense matrix format and the related operations of the local native library are compared. The fixed matrix density is the common density 0.01 in the recommended system data, the amount of serialized data for shuffle is still very large. When the density reaches about 0.3, the performance of the original dense matrix multiplication can be better than the implemented sparse-sparse matrix multiplication. This is also implemented with sparse matrix multiplication in the single machine sparse matrix test and the relationship between the pros and cons of invoking the local native library is basically the same. When the density of the fixed sparse matrix is 0.01, as the size of the matrix increases, the performance of the implemented distributed dense-sparse matrix multiplication is better than the same sparseness but using the related operations of dense matrix storage, and as the dimension increases, the speedup also increases from 1.88× to 5.71×. The performance of the dense-sparse matrix design and implementation is better than the original performance when the dense matrix is used to deal with such problems, especially when the density of the sparse matrix is below 0.1, the advantages are more obvious.

The present work in this article cannot achieve a completely automated distributed matrix partitioning strategy, and most of the time it depends on the partitioning settings provided by users. These three are different for different cluster hardware environment, thus affecting the partition of matrix blocks. The next step could be to introduce machine learning models to train a model that can intelligently select the cutting method through pre-performance tests.

Conflict of interest: The authors declare that they have no competing interests.

References

[1] Kamburugamuve S, Wickramasinghe P, Ekanayake S, Fox GC. Anatomy of machine learning algorithm implementations in MPI, SPARK, and FLINK. Exp Mech. 2018;32(1):61–73.

[2] Scholkmann F, Boss J, Wolf M. Ampd: an algorithm for automatic peak detection in noisy periodic and quasi-periodic signals. Algorithms. 2016;5(4):588–603.

[3] Xie S, Low KS, Gunawan E. A distributed transmission rate adjustment algorithm in heterogeneous CSMA/CA networks. Sensors. 2015;15(4):7434–53.

[4] Zeng R, Wang YY. Forward looking infrared target matching algorithm based on depth learning and matrix double transformation. Clust Comput. 2019;22(3):7055–62.

[5] Zhang W, Liu W, Wang X, Liu L, Ferrese F. Online optimal generation control based on constrained distributed gradient algorithm. IEEE Trans Power Syst. 2015;30(1):35–45.

[6] Xiong L, Teng GW, Yu ZP, Zhang WX, Feng Y. Novel stability control strategy for distributed drive electric vehicle based on driver operation intention. Int J Automot Technol. 2016;17(4):651–63.

[7] Heidari A, Agelidis VG, Zayandehroodi H, Pou J, Aghaei J. On exploring potential reliability gains under islanding operation of distributed generation. IEEE Trans Smart Grid. 2016;7(5):2166–74.

[8] Wang Z, Zhao Y, Liu Y, Chen Z, Lv C, Li Y. A speculative parallel decompression algorithm on apache spark. J Supercomputing. 2017;73(9):1–30.

[9] Guo Y, Zhang Z, Jiang J, Wu W, Zhang C, Cui B, et al. Model averaging in distributed machine learning: a case study with Apache Spark. VLDB J. 2021;30(4):693–712.

[10] Wang H, Li L, Zhou C, Lin H, Deng D. Spark-based parallelization of basic local alignment search tool. Int J Bioautomot. 2020;24(1):87–98.

[11] Alnafessa HA, Casale G. Artificial neural networks based techniques for anomaly detection in Apache Spark. Cluster Computing. 2020;23(4):1–16.

[12] Zainab A, Ghrayeb A, Abu-Rub H, Refaat SS, Bouhali O. Distributed tree-based machine learning for short-term load forecasting with Apache Spark. IEEE Access. 2021;9:57372–84.

[13] Moertini VS, Ariel M. Scalable parallel big data summarization technique based on hierarchical clustering algorithm. J Theor Appl Inf Technol. 2020;98(21):3559–81.

[14] Xiao W, Hu J. PsubCLUS: a parallel subspace clustering algorithm based on Spark. IEEE Access. 2020;9:2535-44.

[15] Cheng G, Ying S, Wang B, Li Y. Efficient performance prediction for Apache Spark. J Parallel Distrib Comput. 2021;149(5):40–51.

[16] Huang B, Ma C. Symmetric least squares solution of a class of sylvester matrix equations via MINRES algorithm. J Frankl Inst. 2017;354(14):6381–404.

[17] Li X, Zhao X, Chu D, Zhou Z. An autoencoder-based spectral clustering algorithm. Soft Comput. 2020;24(3):1661–71.

[18] Lee SH, Kim YH, Lee JK, Lee DG. Hybrid app security protocol for high speed mobile communication. J Supercomputing. 2016;72(5):1715–39.

[19] Chávez-Mejía AC, Villegas-Suárez G, Zaragoza-Sánchez PI, Magaa-López R, Jiménez-Cisneros BE. Photocatalytic activity of TiO_2 synthesized by anodization and anodic spark deposition. MRS Adv. 2020;5(61):1–12.

[20] Yu J, Fu Z, Sarwat M. Dissecting GeoSparkSim: a scalable microscopic road network traffic simulator in Apache Spark. Distrib Parallel Databases. 2020;38(4):963–94.

[21] Popov SE, Zamaraev RY. A fast algorithm for classifying seismic events using distributed computations in Apache Spark framework. Program Computer Softw. 2020;46(1):35–48.

[22] Hong S, Choi J, Jeong WK. Distributed interactive visualization using GPU-optimized spark. IEEE Trans Vis Computer Graph. 2020;27(9):3670–84.

[23] Yang A, Qian J, Chen H, Dong Y. A ranking-based hashing algorithm based on the distributed Spark platform. Inf (Switz). 2020;11(3):148.

[24] Myung R, Yu H. Performance prediction for convolutional neural network on Spark cluster. Electronics. 2020;9(9):1340.

[25] Akinwamide SO, Lesufi M, Akinribide OJ, Mpolo P, Olubambi PA. Evaluation of microstructural and nanomechanical performance of spark plasma sintered TiFe-SiC reinforced aluminium matrix composites. J Mater Res Technol. 2020;9(6), 12137–48.

[26] Kumar SA, Subathra M, Kumar NM, Malvoni M, Chopra SS. A novel islanding detection technique for a resilient photo-voltaic-based distributed power generation system using a tunable-q wavelet transform and an artificial neural network. Energies. 2020;13(16):4238.

[27] Nguyen N, Killeen NS, Nguyen DP, Stameroff AN, Pham AV. A wideband gain-enhancement technique for distributed amplifiers. IEEE Trans Microw Theory Tech. 2020;68(9):3697–708.

[28] Zhang F, Cheng L, Li X, Sun YZ. A prediction-based hierarchical delay compensation (PHDC) technique enhanced by increment autoregression prediction for wide-area control systems. IEEE Trans Smart Grid. 2020;11(2):1253–63.

[29] Nguyen DP, Nguyen N, Stameroff AN, Camarchia V, Pham AV. A wideband highly linear distributed amplifier using intermodulation cancellation technique for stacked-HBT cell. IEEE Trans Microw Theory Tech. 2020;68(7):2984–97.

[30] Balogun BF. Distributed firewalls mechanism for the resolution of packets forwarding problems in computer networks using RSA-CRT technique. Int J Computer Appl. 2021;174(15):32–8.
[31] Sirige SS, Choudhury S, Jayalakshmi NS. Islanding detection of distributed generation systems using hybrid technique for multi-machine system. Int J Power Electron Drive Syst. 2020;11(4):2046.
[32] Faturrahman MI, Yoyo Y, Zaini AR. Technique and quality translation of idhafi in The Matan Hadits of Arba'in al-Nawawi. J Al Bayan J Jur Pendidik Bhs Arab. 2020;12(2):208–24.
[33] Zhou W, Labahn G, Storjohann A. A deterministic algorithm for inverting a polynomial matrix. J Complex. 2015;31(2):162–73.
[34] Walunj G, Bearden A, Patil A, Larimian T, Borkar T. Mechanical and tribological behavior of mechanically alloyed Ni-TiC composites processed via spark plasma sintering. Materials. 2020;13(22):5306.
[35] Adesina OT, Sadiku ER, Jamiru T, Adesina OS, Salifu S. Polylactic acid/graphene nanocomposite consolidated by SPS technique. J Mater Res Technol. 2020;9(5):11801–12.