Noname manuscript No.
(will be inserted by the editor)

S-MPEC : Sparse Matrix Multiplication Performance Estimator on a Cloud Environment

Jueon Park · Kyungyong Lee

Received: date / Accepted: date

Abstract Sparse matrix multiplication (SPMM) is widely used for various machine learning algorithms. As the applications of SPMM using large-scale datasets become prevalent, executing SPMM jobs on an optimized setup has become very important. Execution environments of distributed SPMM tasks on cloud resources can be set up in diverse ways with respect to the input sparse datasets, distinct SPMM implementation methods, and the choice of cloud instance types. In this paper, we propose S-MPEC which can predict latency to complete various SPMM tasks using Apache Spark on distributed cloud environments. We first characterize various distributed SPMM implementations on Apache Spark. Considering the characters and hardware specifications on the cloud, we propose unique features to build a GB-regressor model and Bayesian optimizations. Our proposed S-MPEC model can predict latency on an arbitrary SPMM task accurately and recommend an optimal implementation method. Thorough evaluation of the proposed system reveals that a user can expect 44% less latency to complete SPMM tasks compared with the native SPMM implementations in Apache Spark.

Keywords cloud computing · instance recommendation · sparse matrix multiplication · Apache Spark

Jueon Park
Department of Computer Science. Kookmin University.
Seoul. South Korea.
E-mail: jueon@kookmin.ac.kr

Kyungyong Lee (*Corresponding Author*)
Department of Computer Science. Kookmin University.
Seoul. South Korea.
E-mail: leeky@kookmin.ac.kr

1 Introduction

Large-scale datasets from real-world applications and their relationships can be represented using a graph. In a graph, an edge between two nodes means they are related in some ways. An edge can have a weight value depending on the datasets. Friends connections and users subscriptions in a social network, product-review ratings in an e-commerce site, user-movie ratings in a movie-streaming service, and hyperlinks from a source to a destination website can be expressed using a graph. For extracting valuable information from graph-structured datasets using several types of machine learning algorithms, there is need to express the datasets in a computer-friendly format. Moreover, many data mining algorithms require input datasets to be represented in a sparse matrix format. For example, the power method implementation of the PageRank algorithm [28] and nonnegative matrix factorization (NMF) [22], which is widely used for various recommendation systems, requires input datasets to be represented in a matrix format.

Furthermore, processing various types of big data requires considerable computing power and general-purpose distributed computing platforms, such as Apache Hadoop [11] and Spark [44]. It also requires a set of application programming interfaces (APIs) that abstract complex fault-tolerance mechanisms in a distributed environment, task scheduling with heterogeneous resources, and guaranteeing scalability as demand changes. Using the high-level APIs of Apache Spark, we can implement various machine learning algorithms conducted on a shared-nothing distributed computing environment using the MLLib [25]. This MLLib [25] provides various matrix operations on a distributed computing environment [2], including matrix multiplication and factorization, which are the core kernels of many machine learning algorithms.

Generally, running an Apache Spark task for data mining jobs requires huge computing power. To meet various computing demands, running a Spark cluster in a cloud computing environment becomes a norm. When using cloud resources, the users have to choose compute instances from the perspective of response time and cost. Cloud computing service evolves very fast, and it is challenging for ordinary users to follow updates and apply them to their analysis environment. For instance, at the time of writing, Amazon Web Services (AWS) provides over 100 Elastic Computing Cloud (EC2) instance types, which users have to decide to satisfy application needs.

In addition to the cloud computing service diversity, there are many challenges when running data mining algorithms with SPMM using Apache Spark MLLib [25]. Input datasets for an SPMM task can be diverse as they can be generated from various sources, such as social network service [24]. In addition, there are various ways of implementing SPMM tasks in a distributed environment that even adds complexity to optimize the execution. Despite this complexity, almost every guidance optimally operated SPMM tasks using Apache Spark on a cloud environment.

To help users better understand the distributed SPMM execution characteristics and guide optimal implementation of arbitrary SPMM tasks with different numbers of rows and columns, densities, multiple implementation methods, and cloud instance types, we propose S-MPEC, Sparse Matrix Multiplication Performance Estimator on Cloud. We first propose a set of features that represent the characteristics of input matrices, multiplication implementation details, and underlying cloud instances. Using the proposed features, S-MPEC utilizes a GB regressor [12] modeling to predict latency of various SPMM tasks. To find the optimal hyper-parameters of the proposed model, we employed the Bayesian optimization [36] algorithm. Using the generated model, S-MPEC predicts the latency of a given SPMM workload and recommends proper cloud instance types and SPMM implementation methods. We thoroughly evaluated S-MPEC under realistic scenarios and discovered that S-MPEC could improve the average response time of SPMM 44% compared to the Apache Spark MLLib's native implementation by switching appropriate multiply implementation methods dynamically.

In summary, the main contributions of this paper are as follows.

- We discovered the performance variability of a distributed SPMM operation on Apache Spark and the lack of general guidance regarding performance characteristics.
- We propose features that can represent distributed SPMM tasks and cloud instances.
- We build a model that accurately predicts the latency of arbitrary SPMM tasks on various cloud instances.
- We assess the practicality of building a model that predicts a distributed SPMM performance for diverse scenarios to improve the overall performance.

This paper is organized as follows: Section 2 presents related work in the literature. Section 3 discusses various algorithms to implement SPMM with Apache Spark and presents the performance variability. Section 4 presents the proposed model that predicts SPMM performance, and Section 5 thoroughly evaluates the proposed model. Section 6 concludes this paper with future work.

2 Related Work

Optimizing Matrix Multiplication : Optimization of matrix multiplication in an HPC environment has been thoroughly studied in the literature. Researchers have carried out many studies on minimizing communication overhead using a highly optimized MPI library or carefully designing algorithms on multi-core shared memory machines, such as SUMMA [39], CARMA [8], Patwary et. al. [31], and ScalaPack [6]. Despite the technological advancement in the HPC research community, not much work have been done to characterize and optimize performance of SPMM on a distributed shared-nothing computing environment, especially for Apache Spark [44] and Hadoop [11], which are popular big-data processing engines. The proposed S-MPEC provides a way to build an efficient environment to conduct distributed SPMM tasks using Apache Spark. The previous work focused on the optimization of SPMM on a single machine in a HPC or specialized hardware. They are complementary for our proposed work because each executor in a Spark cluster can equally apply such optimization techniques proposed in the HPC domain.

Matrix Multiplication on Big-Data Processing Framework : Kim et. al. [19] and Marlin [13] proposed an algorithm to predict matrix multiplication performance of Apache Spark. Marlin proposed to use matrix size to decide optimal scheduling of multiplication tasks, whereas Kim et. al. suggested using various matrix multiplication characteristics to predict performance. Despite the improvement in the Kim et. al. and Marlin approaches over the native Spark feature, both work support only dense matrix multiplication which limits the work's applicability. In addition, Marlin does not support various cloud instances in the recommendation. HAMA [33] proposed a MapReduce program using Apache Hadoop to perform distributed matrix multiplication, but it supports only dense matrix cases. Yu. et. al. [43] thoroughly analyzed shuffling overhead of the distributed matrix multiplication and proposed a task execution plan concerning the network overhead. Despite the large amount of work involved in the course of shuffling, other factors also significantly impact its performance. Moreover, aside from the dense matrix multiplication, an SPMM task can be implemented in more diverse ways. This factor has to be considered when choosing an optimal execution plan. Park et. al. [30] proposed an algorithm that predicts the latency of performing SPMM tasks, but they did not consider a cloud environment where diverse instance types exist.

Stark [26] presented an implementation of Strassens's matrix multiplication [15] on Apache Spark. The authors showed that their implementation outperforms the native Spark MLLib implementation and Marlin [13]. JAMPI [10] is an implementation of Cannon's distributed matrix multiplication [23] on Apache Spark combining distributed message passing and Spark's barrier execution mode. Comparing to the Spark MLLib implementation, JAMPI showed up to 24% faster response time. Despite of the performance improvement over the baseline implementations, Stark and JAMPI supports multiplication of two dense matrices only, and it limits the applicability of the work because sparse matrix representation is widely adopted in big-data processing.

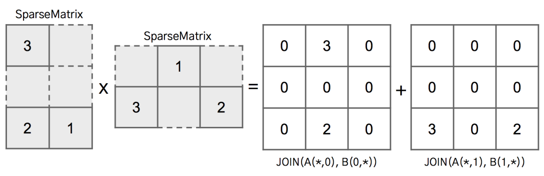
Optimal Big-Data Processing Framework Configuration : Herodotou et. al. proposed Starfish [14] that finds optimal configurations for MapReduce [7] workloads, but it did not consider various cloud instance types. Jalaparti et. al. proposed Bazaar [16], and Wieder et. al. proposed Conductor [41]. They find cost based optimal cloud instances for given workloads on Hadoop, but they did not evaluate for SPMM, so complex characteristics of SPMM cannot be considered from them. Cheng et. al. proposed CAST [5] that helps in select storage on cloud, and Klimovic et. al. proposed Selecta [20] that provides an optimal configuration for cloud storage and instance. They showed good accuracy on prediction, but they use simple workloads, such as sort, join, grep and k-means that cannot consider SPMM's complex characteristics.

Optimal Cloud Environment for Machine Learning : To help users build optimal cloud environments and process a large-scale of datasets, Ernest [40] proposed an algorithm that predicts the performance of various machine learning jobs with different input datasets and various cloud instance types. The authors used a non-linear regression for their prediction model. To achieve a similar goal with Ernest, PARIS [42] applied Random Forest [3] modeling, and CherryPick [1] adopted Bayesian Optimization to select the optimal set of experiments on cloud. FC^2 [27] presents a system which recommends optimal cloud instances for distributed machine learning. The system does not reference machine learning algorithm source code but use resource information and scalability property of a machine learning task. The authors of FC^2 provide a web-interface so that users can easily access the proposed system. The proposed work focused on predicting general machine learning jobs. However, the task of characterizing the SPMM is very different from a general machine learning job whose main computation kernel is not the matrix multiplication. Son et. al. [37] revealed that the input dataset for a matrix multiplication task is distinct from general machine learning jobs, and the performance prediction of matrix multiplication jobs should be treated separately.

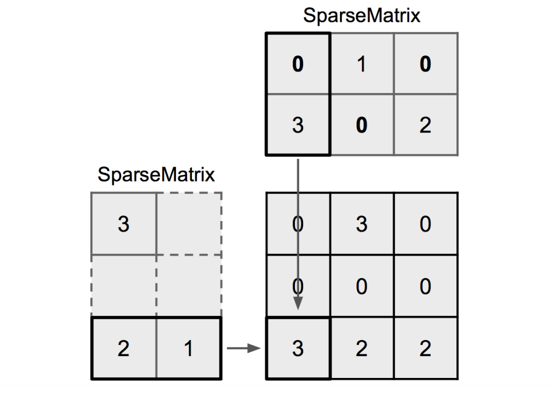
To better evaluate cloud environment for big-data processing, Shen et. al. [35] proposed a model to assess performance of cloud services for various machine learning jobs. S-MPEC is complementary for the work, and it can apply the assessment algorithm proposed in the work to improve execution time accuracy for general workload other than SPMM. Shahidinejad et. al. [34] proposed a workload scheduling algorithm on a cloud computing environment. The authors proposed to categorize heterogeneous workloads using a K-Means algorithm and make a scaling decision using a tree-structure branching algorithm. Using the proposed algorithm, most SPMM tasks can be clustered into a same category and miss various SPMM task characteristics that is presented in this paper. Thus, workload-specific parameters need to be considered in the scheduling as presented in this work.

3 Distributed Sparse Matrix Multiplication

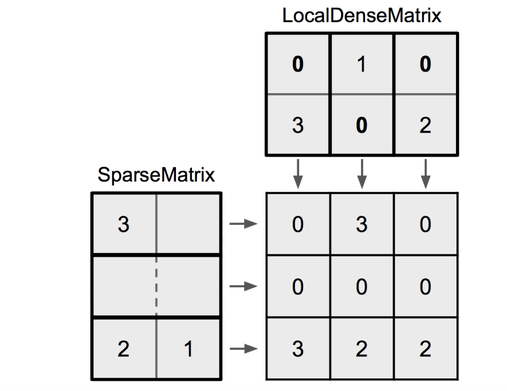
To store and access a sparse matrix using Apache Spark, MLLib [25] provided distributed sparse matrix representations: *indexed-row* and *block* matrices. Both approaches use resilient distributed dataset [44] as the underlying mechanism to store the sparse matrices, while guaranteeing fault-resilience. In the *indexed-row* representation, an input matrix is stored in a row-wise manner where a row is stored as a sparse vector locally in the distributed servers. To store blocks in a column-major order, we transpose an input matrix and store it in the *indexed-row* format. Moreover, in the *block* representation, an entire sparse matrix is partitioned into either the row and/or column direction. After partition-



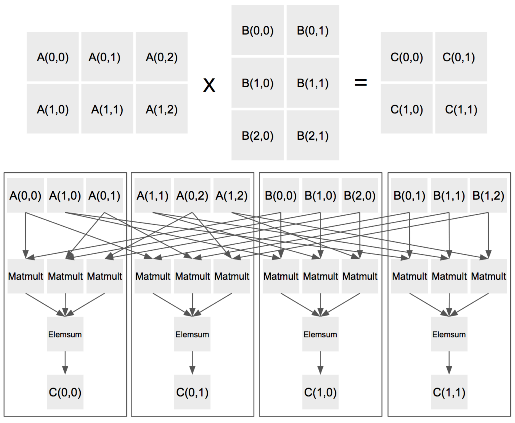
(a) Outer sparse product



(b) Inner sparse product



(c) Indexed-Row multiply



(d) Block multiply

Fig. 1: Various SPMM implementation with distributed sparse matrices

ing, a block of the sparse matrix is stored in compressed sparse column (CSC) format.

Using the distributed representations of sparse matrices, we highlight four distributed SPMM implementations in Figure 1. For the brevity of the descriptions, we define $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, as the left, right, and the result matrix, which is $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. We use i and j to denote an arbitrary row index of $\mathbf{A}$ and a column index of $\mathbf{B}$. Moreover, k denotes an arbitrary column index of $\mathbf{A}$ and row index of $\mathbf{B}$. The capital letters I, K, and J denote the corresponding dimension size. In each matrix, we use subscripts with a parenthesis to denote the row and column indices. For example, $\mathbf{A}_{(i,k)}$ indicates an element in the i-th row and k-th column of matrix $\mathbf{A}$. The $*$ notation in the row or column index of a matrix denote an entire row or column. For example, $\mathbf{A}_{(i,*)}$ indicates the i-th row of a matrix $\mathbf{A}$. It should be noted that N expresses the number of Spark worker nodes to perform an SPMM task and assume that an input matrix can be equally partitioned among N worker nodes.

3.1 Outer-Sparse SPMM

Figure 1a illustrates an SPMM implementation of the outer sparse product. In the method, $\mathbf{A}$ is partitioned into column-wise ($\mathbf{A}_{*,k}$), and $\mathbf{B}$ is partitioned into row-wise ($\mathbf{B}_{k,*}$). A *GroupBy* operation is applied for k. In each grouped result, an outer product is conducted ($\mathbf{A}_{*,k} \otimes \mathbf{B}_{k,*}$) that results in an intermediate output matrix of size $I \times J$. During the outer product operation, each vector remains in a sparse status. We performed the element-wise summation to derive the final result, $\mathbf{C}$.

Shuffle overhead: The shuffle overhead of two sparse matrices multiplication is majorly impacted by the distribution of the number of non-zero (NNZ) elements which is not a static value. However, to compare the shuffle overhead of different SPMM implementations, we omitted the impact from the NNZ distribution in each method. In the outer SPMM implementation, a worker node is responsible for $I \times \frac{K}{N}$ elements from $\mathbf{A}$ and $\frac{K}{N} \times J$ from $\mathbf{B}$. Thus, an executor process fetches necessary partitions from $\mathbf{A}$ and $\mathbf{B}$ that is $I \times \frac{K}{N} + \frac{K}{N} \times J$, which happens N times. Each worker node conducts multiplication and generates an output matrix of size $I \times J$ that is summed element-wise. Hence, the shuffling overhead of the outer SPMM becomes Equation 1.

$$Shuffle_{outer} = K \times (I + J) + N \times I \times J \tag{1}$$

Methods	Dominant Shuffle	ToDense	Native
Outer	$N \times I \times J$	No	No
Inner	$N \times K \times (I + J)$	No	No
IndexedRow	$N \times K \times (I + J)$	Yes	Yes
Block	$\sqrt{N} \times K \times (I + J)$	Yes	Yes

Table 1: Performance characteristics of different SPMM implementations

3.2 Inner-Sparse SPMM

Figure 1b explains an SPMM implementation, which uses an *inner sparse* product. In the method, $\mathbf{C}_{i,j}$ is generated by conducting an inner product of a row from the left matrix and a column from the right matrix, $\mathbf{C}_{i,j} = \mathbf{A}_{i,*} \cdot \mathbf{B}_{*,j}$. In the inner-product operation, we kept each vector in a sparse state. Stitching scalar values from the inner-product operations results in the final output matrix, $\mathbf{C}$.

Shuffle overhead: In the inner SPMM implementation, a worker node is responsible for $\frac{I}{N} \times K$ elements from $\mathbf{A}$ and $K \times \frac{J}{N}$ elements, which are from $\mathbf{B}$. An executor process fetches the necessary partitions from $\mathbf{A}$ and $\mathbf{B}$. That is $\frac{I}{N} \times K + K \times \frac{J}{N}$ which happens N^2 times. After conducting multiplication in each executor, we aggregated the final output in one place, resulting in shuffling overhead of $I \times J$. The total shuffling overhead of the inner SPMM becomes Equation 2.

$$Shuffle_{inner} = N \times K \times (I + J) + I \times J \tag{2}$$

3.3 IndexedRow Partitioning SPMM

Figure 1c displays an SPMV implementation using the *indexed-row* data structure provided by Spark [38]. The multiplication method is natively supported by Apache Spark. In this method, the right matrix, $\mathbf{B}$, should exist locally in a Spark driver. A driver broadcasts the matrix to all workers after converting it to a dense matrix. In each worker node with multiple executors, the inner product between a sparse vector ($\mathbf{A}_{i,*}$) and a dense vector ($\mathbf{B}_{*,j}$) is calculated by $\mathbf{C_{i,j}}$. The shuffling overhead of *indexed-row* is same as *inner-sparse* in the Equation 2 because the matrix distribution mechanism is the same.

3.4 Block Partitioning SPMM

Furthermore, Figure 1d shows an SPMM method using a distributed block-partitioning mechanism that is natively supported by Apache Spark [38]. In the block-partitioning scheme, we divide the matrix into the row and column directions. To calculate a block of the output matrix, the corresponding entire rows from the left matrix and columns from the right matrix are fetched to a node that is used in the calculation. During multiplication, the left matrix is kept in a sparse CSC format, whereas the right matrix is converted into a dense matrix format.

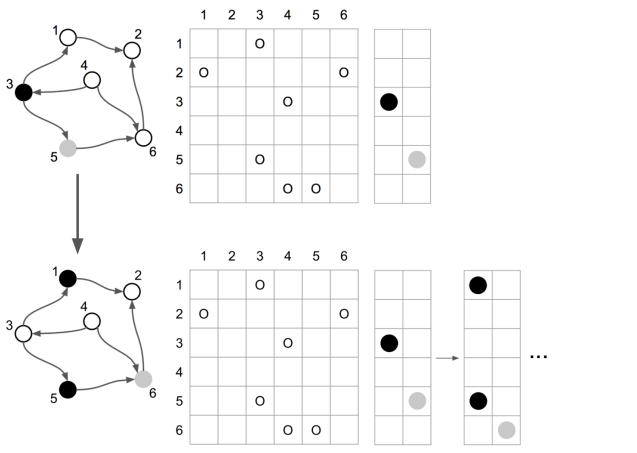


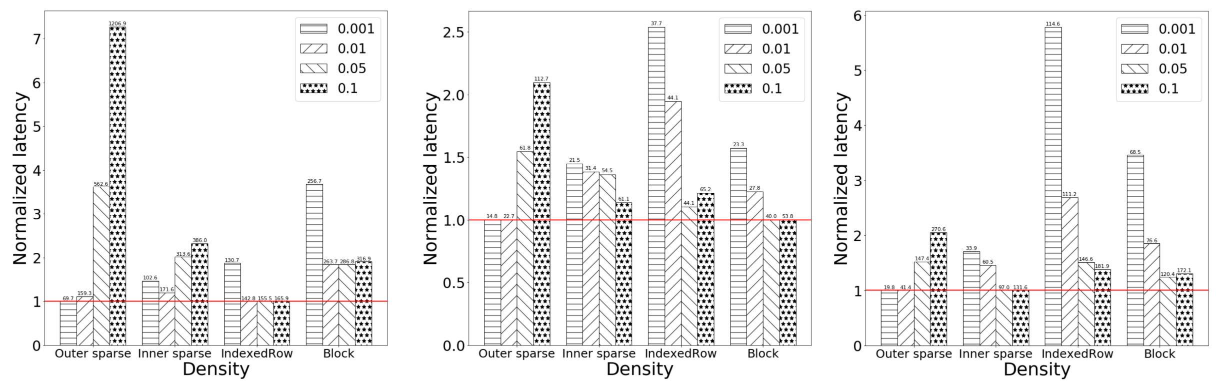
Fig. 2: Multi Source BFS algorithm as a workload to evaluate various distributed SPMMs

Shuffle overhead: For brevity of the analysis, we assume that N worker nodes can divide the input matrices evenly [19]. A worker node that is responsible for an output block fetches $\frac{I}{\sqrt{N}} \times K$ from $\mathbf{A}$ and $K \times \frac{J}{\sqrt{N}}$ from $\mathbf{B}$. After each worker's node performs multiplication locally, the final result is gathered that results in shuffle overhead of $I \times J$ across all executors. Thus, the total shuffling overhead of block SPMM becomes Equation 3.

$$Shuffle_{block} = \sqrt{N} \times K \times (I + J) + I \times J \tag{3}$$

We do not compare the number of multiply operations for different SPMM implementations because the value is the same across all the implementation mechanisms [21].

Table 1 summarizes the characteristics of the four distinct SPMM implementations, as shown in the first column. The second column shows the dominant factor that impacts the overall shuffle overheads. We can see that the shuffle overhead is mainly dependent on the shape of the left and right matrices. The third column indicates whether there exists conversion from the sparse format to the dense format. The *IndexedRow* and *Block* requires the right matrix to be transformed into a dense format, whereas the left matrix remains in a sparse format. The fourth column indicates whether the corresponding SPMM implementation is natively



(a) Orkut dataset (b) DBLP dataset (c) Youtube dataset

Fig. 3: Sparse matrix multiplication using Apache Spark with different matrix distribution mechanisms

supported by the Apache Spark, MLLib [38]. Notably, Spark does not natively support the outer product and inner sparse product. Thus, we implement these ourselves to make S-MPEC cover wide range implementations.

3.5 Performance Dynamics of Distributed SPMM

To understand the performance of different SPMM mechanisms presented in Figure 1, we conducted a performance evaluation on different heuristics. In the experiments, we performed SPMMs using a left sparse matrix and a right matrix with various densities. To generate a realistic SPMM workload, we employed the multiple-source breadth-first search (BFS) algorithm [17], which is explained in Figure 2. The algorithm repeatedly performs an SPMM operation with a left matrix built from an input sparse dataset and a right matrix indicating the source to the destination mappings. A right matrix for a multiply operation is first defined by setting a source ID node to 1 and others to 0 in each column. The result of the left and right matrix multiplications indicates the path from the source to the destination. During iterative multiplications, a result matrix becomes the right matrix for the next iteration, indicating the path connection in multiple hops. Because a right matrix is updated in every iteration, the density changes in every iteration. Thus, it can provide various multiplication scenarios by operating in multiple iterations.

To evaluate the performance of the four different distributed implementations of SPMM in Figure 1, we used Orkut (Figure 3a), DBLP (Figure 3b), and YouTube (Figure 3c) datasets as left matrices. The datasets are downloaded from the SNAP site [24]. For the right matrix, we set a random source element to be 1, and update the right matrix in every iteration using a result matrix of a previous SPMM. To investigate the characteristics from various multiplication scenarios, we set the density of the right matrix to 0.001, 0.01, 0.05, and 0.1. Because the right matrix becomes denser in different ratios for a distinct left matrix dataset, we choose cases in which the right matrix density is the closest the configured densities. In each figure, different distributed SPMM implementations are expressed on the horizontal axis. Each SPMM method has four bars that represent the performance of the distinct right matrix densities. The vertical axis indicates the normalized latency for the best performance of the same right matrix density with different distributed SPMM mechanisms. The experiments were conducted on AWS Elastic MapReduce version 5.27.0 with one master and four workers, and we used $r5.2xlarge$ instances. All workload scenarios were conducted three times, and the median value was selected from the results to remove the impact from experimental noise in cloud [29, **?**].

To represent the relative performance difference of the SPMM method with their corresponding different right matrix densities, we grouped the bars of the different densities with the same SPMM method. For example, the *outer sparse* in Figure 3a, exhibits a better performance than other SPMM methods when the right matrix density is 0.001, but the performance degrades significantly when the density of the right matrices increases. When the right matrix density becomes 0.1, and the *outer sparse* implementation shows about seven times more latency than the best case (*indexed-row*) for the Orkut dataset. Different input datasets show a different pattern of performance changes with different right matrix densities. Orkut dataset shows over seven times performance difference. For DBLP dataset, it is about two and a half times. Furthermore, the performance change pattern is not consistent for all SPMM

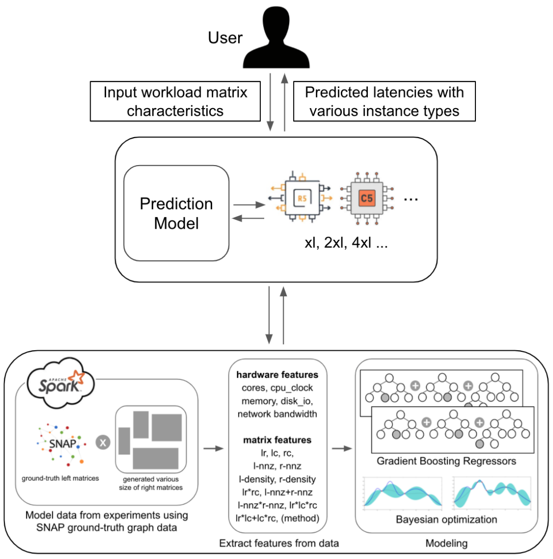

User
Input workload matrix characteristics
Predicted latencies with various instance types
Prediction Model
R5
C5
...
xl, 2xl, 4xl ...
Spark
SNAP
ground-truth left matrices
generated various size of right matrices
Model data from experiments using SNAP ground-truth graph data
hardware features
cores, cpu_clock
memory, disk_io,
network bandwidth
matrix features
lr, lc, rc,
l-nnz, r-nnz
l-density, r-density
lr*rc, l-nnz+r-nnz
l-nnz*r-nnz, lr*lc*rc
lr*lc+lc*rc, (method)
Extract features from data
Gradient Boosting Regressors
Bayesian optimization
Modeling

Fig. 4: S-MPEC architecture

implementations. For *inner-sparse* implementation, the latency keeps increasing as the right matrix densities increase for Orkut dataset. However, for the Youtube dataset, the latency keeps decreasing as the right matrix densities increase. From the figures, we can observe that the performance difference among different SPMM implementations, right matrix densities, and left matrix characteristics are significant and noticeable. Unfortunately, we cannot determine a globally optimal SPMM implementation, which can result in considerable performance variability for various data mining jobs implemented on Apache Spark.

4 S-MPEC Architecture

As we cannot determine a globally optimal SPMM implementation, we propose S-MPEC to recommend the optimal distributed SPMM execution environment. S-MPEC receives details about input matrices to conduct a SPMM task from the user. Using the input matrices information, S-MPEC predicts latency to complete an input workload under four different SPMM implementations and cloud instance types. In addition, S-MPEC helps users choose optimal environments to complete the input workload considering latency and cost using the predicted latency.

4.1 Modeling Distributed SPMM Performance

The performance of distributed SPMM implementations differs significantly as the input datasets vary. It is very critical to predict the estimated latency of an arbitrary SPMM task because it is among the core kernel of many machine learning jobs. To predict the performance of various SPMM tasks under diverse cloud environments, we first propose features to represent characteristics of various SPMM workloads and cloud instance types. Using the proposed features, S-MPEC builds a prediction model using a GB-regressor [12] that accurately represents non-linear interactions among features. Furthermore, S-MPEC applies Bayesian optimization [36] to determine the optimal hyper-parameters.

In building a prediction model, the first step is how to determine a representative set of features. In sparse matrix, we use the dimension of left and right matrices and name them as lr, lc, and rc to denote our representative set of features, where lr, lc, and rc is the number of left matrix rows, left matrix columns, and the right matrix columns, respectively. Because the target workload of the proposed model is a sparse matrix, the density of a matrix is an important factor that must be considered. We call it $l - density$ and $r - density$ for left matrix density and the right matrix density, respectively.

In addition to the left and right densities, we also add the number of nonzero (nnz) elements of the left and right matrices, $l - nnz$ and $r - nnz$, accordingly. Note that the nnz of a matrix is already reflected in the density feature because it is calculated by dividing the nnz ($l - nnz$ for a left matrix) by the total number of elements ($lr \times lc$ for a left matrix). Ideally, such a relationship should be captured while building a model. Deep neural-net [32] is good at finding hidden relationships from a very large-scale input dataset. However, it requires a significant number of input datasets to detect hidden characteristics, which is impractical to generate the numerous input datasets required for SPMM tasks. Thus, we add the nnz and density feature manually.

We add $lr \times rc$ to represent the dimension of an output matrix from the SPMM task. Because we target sparse input datasets, the size and NNZ elements from an output matrix cannot be estimating before the actual computation. We expect that the combination of $l - nnz$, $r - nnz$, and $lr \times rc$ can estimate the overhead of the node that is responsible for storing an output matrix. We add $l - nnz + r - nnz$ to represent the shuffling overhead for all nodes during computation. To consider the actual number of product computations in the sparse matrix format, we add $l - nnz \times r - nnz$.

Referencing the performance estimation of the distributed dense matrix multiplication using Spark [19], [37], we add $lr \times lc \times rc$ and $lr \times lc + lc \times rc$, which represent the total number of product operations and the shuffle overhead, respectively. To build a unified model that is applicable for the four methods of an SPMM operation, we add a *method* feature categorically.

To express the characteristics of various cloud instances, we add hardware features that are the number of CPU cores, CPU clock speed, memory size, disk, and network bandwidth of cloud instance types. To use practical disk IO bandwidth, we conducted *dd* workloads, whereas the *iperf3* command in the FunctionBench [18] is used for the network bandwidth.

We build our prediction model using the GB regressor [12] with the features that are mentioned above. The GB regressor produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It produces a classifier based on the classifier's accuracy generated on the previous step and ensembles those classifiers to generate a more accurate final model.

In GB regressor modeling, various hyper-parameters are used, such as *learning_rate*, *max_depth*, *gamma*, and so on. It is not easy to find the best hyper-parameter combination in modeling. Although experimenting with all possible combinations of hyper-parameters provides the best hyper-parameters, it takes a long time. Therefore, we employed the Bayesian optimization to determine the optimal hyper-parameters with extra minimal overhead.

Figure 4 shows the overall architecture of the proposed S-MPEC. A user submits workload characteristics of a SPMM task, such as the dimension of left and right matrices, and densities. S-MPEC equips a model that is built offline using various SPMM scenarios with diverse cloud instance types. Using the model, S-MPEC predicts the latency of the input workload scenario for various cloud instance types and sizes.

5 Evaluation

To present the applicability and advantage of predicting the performance of SPMM operations with various sparse matrices, distributed implementation, and cloud instance types, we conducted experiments covering thorough scenarios. To present various input dataset, we used *Orkut*, *DBLP*, and *Youtube* graph datasets from SNAP [24]. The *Orkut* dataset contains 3,072,441 nodes, 117,185,083 edges, and 234,370,166 *nnz*. The *DBLP* dataset has 317,080 nodes, 1,049,866 edges, and 2,099,732 *nnz*. The *Youtube* dataset also contains 1,134,890 nodes, 2,987,624 edges, and 5,975,248 *nnz*. We applied four distributed SPMM implementations to the input datasets, but some implementations could not complete the given workloads because of the memory limitation. We used six AWS EC2 instances types: *c5.xlarge, c5.2xlarge, c5.4xlarge, r5.xlarge, r5.2xlarge,* and *r5.4xlarge* as our cloud instances. $c5$ instances are designed for compute-intensive workloads, while $r5$ instances are optimized for workloads that require large amounts of memory. For example, $c5.xlarge$ instance offers 2 CPU cores and 4GB memory, while $r5.xlarge$ instance offers 2 CPU cores and 32GB memory. $r5$ instances provides 8 times more memory than $c5$, but CPU core of $c5$ instances offer 3.0GHz clock rate, while $r5$ instances offer 2.5GHz. Thus, some matrix multiplication scenarios can be completed with $r5$ instances, but $c5$ instances cannot complete because of the memory limitation. Using the input dataset as a left matrix, we conducted multi-source BFS algorithms in multiple iterations using Apache Spark. To reproduce a practical BFS scenario, we varied the sparsity of the right matrix to different degrees. The experiments were conducted on AWS Elastic MapReduce version 5.27.0 with one master and four workers. All the experiments are conducted three times, and we take median for presentation. We used *Python 3.8.5* with *xgboost 1.2* and *bayesian-optimization 1.2* libraries in building the optimized prediction model from the matrix multiplication results.

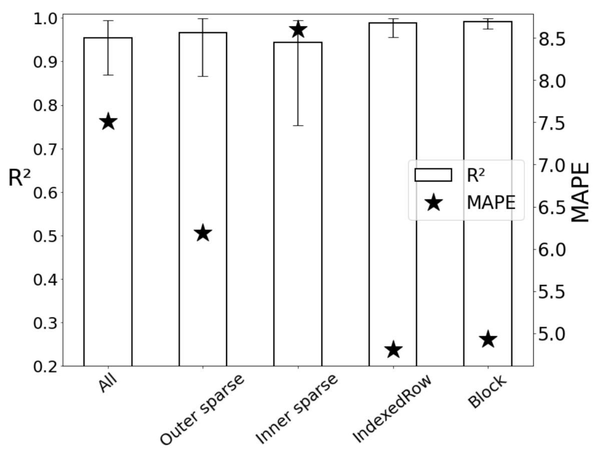


Fig. 5: Prediction accuracy of a model built with GB regressor

5.1 Model Accuracy for Distinct SPMM Implementations

We first evaluated the GB-regressor model's prediction accuracy, which is S-MPEC's prediction modeling algorithm, with the proposed feature sets. We performed K-fold cross validation ten times, dividing the training and test datasets into an 8:2 ratio. We measured the prediction accuracy using R^2, and the mean absolute percentage error (MAPE) metric. The R^2 metric measures the degree of resemblance of the predicted value

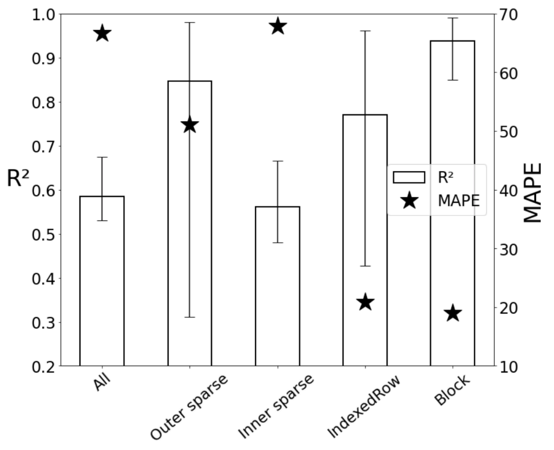


Fig. 6: Prediction accuracy of a model built with NNLS

to the true value. The higher R^2 value represents an accuracy of a model with the maximum value of 1.0.

Figure 5 displays the R^2 value (higher is better) in the primary vertical axis whose values are represented in the bar with minimum and maximum indicators. The MAPE value (lower is better) is shown in the secondary vertical axis, where their values are represented as star marks. The horizontal axis, reveals the distributed SPMM implementations. The first horizontal axis value, *All*, indicates a case that a prediction model is built using datasets from all SPMM implementation methods, and only the *All* model contains *method* feature. The latter four values indicate the prediction accuracy of a model built from training datasets, generated from each method explained exclusively in Section 3. We observe that the models built from exclusive datasets regarding the SPMM implementation method exhibit better prediction accuracy than the unified model. However, even the worst performing unified *All* method demonstrates accurate results with 0.95 of R^2 and a MAPE of about 7.6%. The best prediction accuracy is achieved with the *Block* implementation for R^2 metric. Overall, we can observe that the proposed S-MPEC can accurately model the latency of various SPMM scenarios with diverse inputs.

To present superb prediction accuracy for a model with a GB-regressor algorithm, we developed another model using the nonnegative least square (NNLS) linear regressor algorithm [4]. The result is presented in Figure 6. The overall prediction accuracy pattern is very similar to that presented with the GB-regressor algorithm. However, the degree of accuracy is significantly lower. For instance, the R^2 value of the *All* method is only about 0.6, and MAPE is over 60%. From the result, we conclude that there is a non-linear interaction among the suggested features for various SPMM implementations with diverse cloud instance types, because a linear model cannot capture such characteristics effectively.

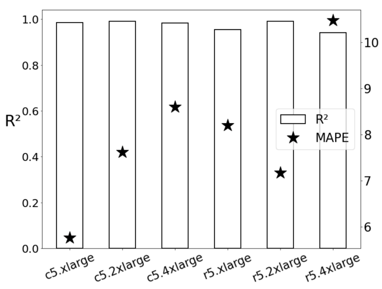
In the modeling step, S-MPEC utilizes the Bayesian optimization algorithm to find an optimal set of parameters for the GB-regressor model. Table 2 presents parameters that users can set during the GB-regressor modeling and the default values from the *xgboost* library to show the performance improvement from the hyper-parameter search step. The last column shows the optimal hyper-parameters for the prediction model. In running Bayesian optimization, we set the objective metric as $-1 \times (MAPE \times 10 + RMSE)$ that is minimized during the step. We multiplied the MAPE value by ten to fit the range of the absolute value with the RMSE. Through the hyper-parameter optimization, we can see that the prediction accuracy improves significantly for all the evaluation metrics, and it contributes to improving the prediction accuracy of S-MPEC. Comparing to the default configuration, S-MPEC adopts a complex model with a larger number of estimators which implies the complexity of the prediction task.

	Default	S-MPEC
n_estimators	100	9304
colsample_bytree	1	0.972
learning_rate	1	0.019
max_depth	6	5
gamma	0	6.63
subsample	1	0.7
R^2	0.93	0.97
MAPE	13.51	7.32
RMSE	45.23	34.71

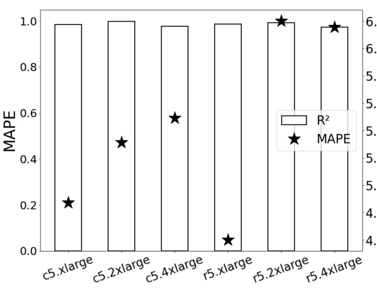
Table 2: Improved performance by using Bayesian optimization

5.2 Model Accuracy for Distinct Cloud Instance Types

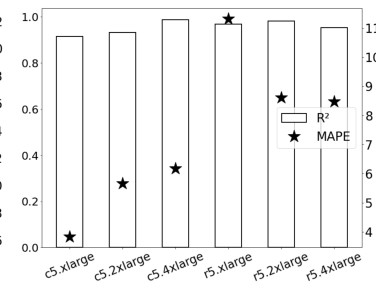
To evaluate the prediction accuracy of S-MPEC for different cloud instance types, we build a model of S-MPEC after excluding a specific instance type (K-fold cross validation for a cloud instance type). Figure 7 shows the prediction accuracy of various SPMM implementations for different instance types, and the horizontal axis represents a target instance type to predict. We build a model after excluding a target instance type on the horizontal axis in our modeling steps. Using the model built, we predict the latency of the various SPMM scenarios by setting the target instance type appropriately in the features. The primary vertical axis shows the R^2 value which is represented as a bar, and the secondary vertical axis shows the MAPE value that



(a) All methods



(b) Outer sparse



(c) Indexed-Row

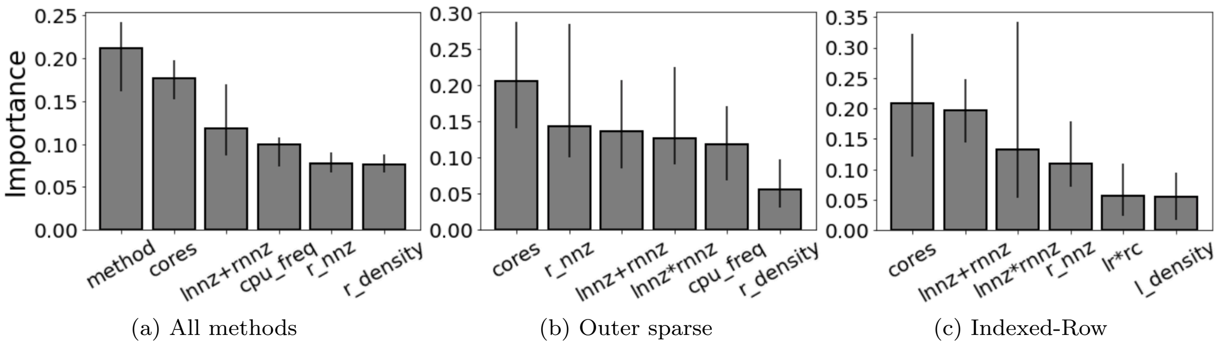
Fig. 7: Prediction accuracy for various cloud instance types

is represented as a star marker. At this stage, we show three SPMM implementation methods because other SPMM implementation methods present a similar pattern. In choosing two implementations for presentation to demonstrate our claim, we considered the uniqueness in the operation (whether transforming a right matrix to a dense format). From the figures obtained, we can observe that S-MPEC predicts the latency of various SPMM tasks when they are executed on diverse cloud computing instances with MAPE less than 11%. On *indexed-row*, MAPE values of $r5$ instances are relatively higher than $c5$, because they could not get enough experiment results from $c5$ instances. Please note that the *indexed-row* method requires more memory than other matrix multiplication methods and $c5$ instances have difficulties in completing the task. This experiment result demonstrates the applicability of S-MPEC in a cloud computing environment, where various instances are supported by many public vendors.

Using a GB-regressor model with Bayesian optimization for optimal hyper-parameter selection, S-MPEC accurately modeled the response time of various distributed SPMM implementations with the proposed features when they are executed using Apache Spark on various cloud environments. To understand which features make significant contributions during modeling, we calculated the feature importance, while building the model. Figure 8 indicates the six most important features for the various distributed SPMM implementations. Each figure reveals the relative importance of each feature. The importance is calculated by counting the number of times a feature is selected during the decision tree build process [9]. For *All* model, the *method* feature is the most dominant feature which indicates the drastic latency difference of different SPMM implementation. Other implementations consider both matrix features and hardware features as important to make accurate prediction in diverse environments. Most methods consider *cores* as an important feature. If given memory is enough to run the workload, the number of CPU cores become the key factor to processing tasks in parallel. Following *cores*, $(l-nnz+r-nnz)$ is an important feature that represents the shuffle overhead. The compute overhead $(l-nnz \times r-nnz)$ is also an important factor to decide the latency. Although we added important features for distributed dense matrix multiplication, which were presented in [37], the features do not show a noticeable influence, which indicates a drastic difference in the dense and sparse matrix multiplication task characteristics. For SPMM tasks, the density of left and right matrices is more important than the dimensions of left and right matrices.

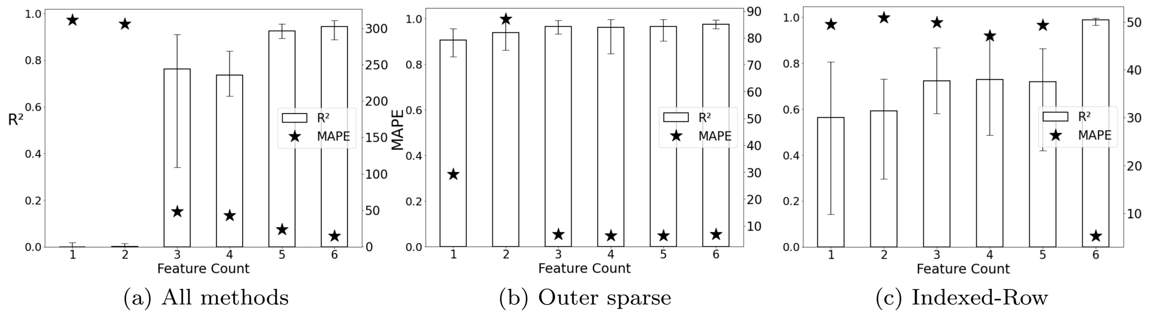
Figure 8 showed the relative importance of proposed features. To understand the impact of the important features to the overall prediction accuracy, we built models by adding few important features that cumulatively measured the prediction accuracy in Figure 9. The horizontal axis represents the number of features used in the modeling, whereas the features are added in a cumulative manner in the order of the importance, presented in Figure 8. For instance, the *All* implementation in Figure 9a of x-axis value of being three means that three most important features (*method, cores,* and $(l-nnz+r-nnz)$) are used in building our model. The primary vertical axis shows the values of R^2, and the secondary vertical axis shows the MAPE value. From the figures, we can deduce that a model built using only top few important features shows accurate result which implies that shuffling and computing overheads are the dominant factors to decide SPMM performance in a distributed cloud environment. The result from Figure 8 and 9 shows that different SPMM implementations require different features to make an accurate prediction model, and the models are difficult to be generalizable which needs a specialized process for the prediction.

Using S-MPEC, users can choose an optimal SPMM implementation method to execute a multiplication task. Assuming that an input dataset is loaded into an Apache



(a) All methods (b) Outer sparse (c) Indexed-Row

Fig. 8: The most important 6 features for each methods and all methods aggregated



(a) All methods (b) Outer sparse (c) Indexed-Row

Fig. 9: Prediction accuracy improvement by adding important features gradually

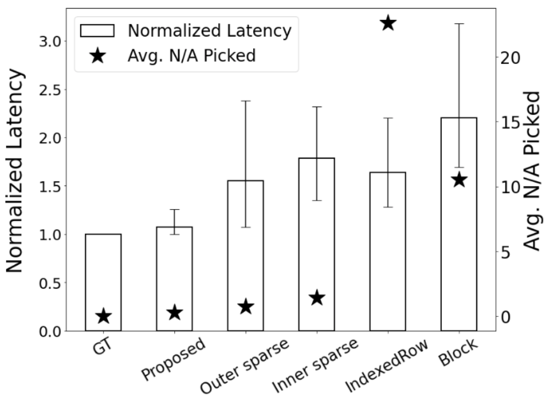


Fig. 10: Performance benefit of using S-MPEC over the Apache Spark native SPMM implementations

Spark driver as an RDD object, users can easily decide which SPMM implementation to execute considering the input characteristics, such as densities of left and right matrices. Please note that *Indexed-Row* and *Block* implementations are natively supported by Apache Spark MLLib. Further, we release our implementation of *Inner-Sparse* and *Outer-Sparse* as an open source to help users choose when necessary.[1]

To present the performance gain from using S-MPEC, we tested various SPMM tasks with distinct input dataset sizes. Each SPMM scenarios have different implementations for the best performance, and the result is shown in Figure 10. The horizontal axis shows the different SPMM mechanisms. We name the best performing implementation method of each workload as ground truth (GT), and S-MPEC (*Proposed*) recommends the predicted best implementation method using a proposed model. To make recommendations, S-MPEC predicts the latency when using four distinct methods and returns the fastest completing SPMM method. The latter four mechanisms use a single implementation method statically, which means a user does not change SPMM implementation based on the input datasets and sticks to one corresponding implementation. The latter four static implementation selection scenario is applied in most cases for ordinary Spark users.

In the experiments, all implementations could not complete some tasks because of memory limitations. We mark such incomplete cases as N/A, and their values are shown in the secondary vertical axis as a star marker. In the figure, the primary vertical axis shows each method's normalized latency on the horizontal axis as a bar. We normalize latency to the GT value because it consists of only the best performing implementation

[1] https://github.com/kmu-bigdata/spark-spmm-compute

of each workload which is the fastest. Thus, the latency value close to 1.0 implies good performance. In calculating the normalized latency, we consider only the cases when a given implementation successfully completes input workloads. As shown in the figure, the proposed S-MPEC shows very good performance close to *GT*. Two native Spark implementations (*IndexedRow* and *Block*) show poor performance when compared to the other two methods in terms of the number of N/A picked because they force the input right matrix to be transformed to a dense format, which causes out-of-memory problems. This result represents the limitation of the current Spark native implementation. From the experiment result, we can conclude that S-MPEC recommends the best performing implementations 87% of cases. If we include top two performing implementation methods for recommendation, S-MPEC recommends 96% of times correctly. Furthermore, it improves the average latency by 44% on average over the Spark native SPMM implementations that are *IndexedRow* and *Block*.

6 Conclusion

This work presents S-MPEC, which accurately predicts the latency of various SPMM tasks when they are executed on diverse cloud computing environments. After summarizing various SPMM implementations in a distributed environment, we demonstrated the performance variability for diverse SPMM implementations and input datasets that necessitates the performance predictor for optimal performance. We proposed feature sets covering input sparse matrix and cloud instance types to build a model that optimizes hyper-parameters. We evaluated S-MPEC thoroughly under realistic scenarios and showed that it improve the Spark native SPMM performance by 44% on average.

Acknowledgement

This work is supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) (NRF-2020R1A2C1102544, NRF-2016R1C1B2015135, and NRF-2015R1A5A7037615), the ICT R&D program of IITP (2017-0-00396), and Research Credits provided by AWS.

References

1. O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard
2. R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia, "Matrix computations and optimization in apache spark," ser. KDD '16. ACM, 2016, pp. 31–38.
3. L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001.
4. D. Chen and R. J. Plemmons, "Nonnegativity constraints in numerical analysis," in *in A. Bultheel and R. Cools (Eds.), Symposium on the Birth of Numerical Analysis, World Scientific*. Press, 2009.
5. Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt, "Cast: Tiering storage for data analytics in the cloud," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 45–56. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749252
6. J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: a scalable linear algebra library for distributed memory concurrent computers," in *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992, pp. 120–127.
7. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264
8. J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal parallel recursive rectangular matrix multiplication," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 261–272. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.80
9. J. Elith, J. R. Leathwick, and T. Hastie, "A working guide to boosted regression trees," *Journal of Animal Ecology*, vol. 77, no. 4, pp. 802–813, 2008.
10. T. Foldi, C. von Csefalvay, and N. A. Perez, "Jampi: Efficient matrix multiplication in spark using barrier execution mode," *Big Data and Cognitive Computing*, vol. 4, no. 4, 2020. [Online]. Available: https://www.mdpi.com/2504-2289/4/4/32
11. A. S. Foundation, "Apache hadoop," 2004. [Online]. Available: http://hadoop.apache.org/
12. J. H. Friedman, "Greedy function approximation: A gradient boosting machine." *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, 10 2001. [Online]. Available: https://doi.org/10.1214/aos/1013203451
13. R. Gu, Y. Tang, Z. Wang, S. Wang, X. Yin, C. Yuan, and Y. Huang, "Efficient large scale distributed matrix computation with spark," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 2327–2336.
14. H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs." *PVLDB*, vol. 4, no. 11, pp. 1111–1122, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/pvldb/pvldb4.htmlHerodotouB11

15. S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull, "Implementation of strassen's algorithm for matrix multiplication," in *Supercomputing '96:Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996, pp. 32–32.
16. V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Bridging the tenant-provider gap in cloud services," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 10:1–10:14. [Online]. Available: http://doi.acm.org/10.1145/2391229.2391239
17. J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. Society for Industrial and Applied Mathematics, 2011. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719918
18. J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019, pp. 502–504.
19. J. Kim, M. Son, and K. Lee, "Mpec: Distributed matrix multiplication performance modeling on a scale-out cloud environment for data mining jobs," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019.
20. A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 759–773. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/klimovic-selecta
21. D. Langr and I. Simecek, "Analysis of memory footprints of sparse matrices partitioned into uniformly-sized blocks," *Scalable Comput. Pract. Exp.*, vol. 19, no. 3, pp. 275–292, 2018. [Online]. Available: https://www.scpe.org/index.php/scpe/article/view/1358
22. D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *In NIPS*. MIT Press, 2000, pp. 556–562.
23. H.-J. Lee, J. P. Robertson, and J. A. B. Fortes, "Generalized cannon's algorithm for parallel matrix multiplication," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 44â51. [Online]. Available: https://doi.org/10.1145/263580.263591
24. J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
25. X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, p. 1235â1241, Jan. 2016.
26. C. Misra, S. Bhattacharya, and S. K. Ghosh, "Stark: Fast and scalable strassen's matrix multiplication using apache spark," *IEEE Transactions on Big Data*, pp. 1–1, 2020.
27. j. . C. p. . S. y. . . d. . h. Nguyen Binh Duong Ta, title = FC2: cloud-based cluster provisioning for distributed machine learning.
28. L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/
29. J. Park, , H. Kim, and K. Lee, "Evaluating concurrent executions of multiple function-as-a-service runtimes with microvm," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020.
30. J. Park and K. Lee, "Performance prediction of sparse matrix multiplication on a distributed bigdata processing environment," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, 2020, pp. 30–35.
31. M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds. Cham: Springer International Publishing, 2015, pp. 48–57.
32. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
33. S. Seo, E. J. Yoon, J. Kim, S. Jin, J. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, 2010, pp. 721–726.
34. A. Shahidinejad, M. Ghobaei-Arani, and M. Masdari, "Resource provisioning using workload clustering in cloud computing environment: a hybrid approach," *Cluster Computing*, vol. 24, pp. 1–24, 03 2021.
35. C. Shen, W. Tong, K.-K. R. Choo, and S. Kausar, "Performance prediction of parallel computing models to analyze cloud-based big data applications," *Cluster Computing*, vol. 21, 06 2018.
36. J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 2951–2959. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999325.2999464
37. M. Son and K. Lee, "Distributed matrix multiplication performance estimator for machine learning jobs in cloud computing," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 638–645. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00088
38. A. Spark, "Apache spark mllib distributed matrix computation," https://goo.gl/Vnii2M, 2017, [Online; accessed 20-Nov-2017].
39. R. A. van de Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," Austin, TX, USA, Tech. Rep., 1995.
40. S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics." in *NSDI*, 2016, pp. 363–378.
41. A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues, "Orchestrating the deployment of computations in the cloud with conductor," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 367–381. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/wieder
42. N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple

public clouds: A data-driven performance modeling approach," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 452–465. [Online]. Available: http://doi.acm.org/10.1145/3127479.3131614
43. Y. Yu, M. Tang, W. G. Aref, Q. M. Malluhi, M. M. Abbas, and M. Ouzzani, "In-memory distributed matrix computation processing and optimization," in *ICDE*, April 2017, pp. 1047–1058.
44. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28.