MPEC: Distributed Matrix Multiplication Performance Modeling on a Scale-out Cloud Environment for Data Mining Jobs

Jeongchul Kim, Myungjun Son, Kyungyong Lee, *Member, IEEE*

Abstract—Many data mining workloads are being analyzed in large-scale distributed cloud computing environments which provide nearly infinite resources with diverse hardware configurations. To maintain cost-efficiency in such environments, understanding the characteristics and estimating the overheads of a distributed matrix multiplication task that is a core computation kernel in many machine learning algorithms are essential. This study aims to propose a Matrix Multiplication Performance Estimator on Cloud (MPEC) algorithm. The proposed algorithm predicts the latency incurred when executing distributed matrix multiplication tasks of various input sizes and shapes with diverse instance types and a different number of worker nodes on cloud computing environments. To achieve this goal, we first analyze the characteristics of distributed matrix multiplication tasks. With characteristics generated from qualitative analysis, we propose to apply an ensemble of non-linear regression algorithm to predict the execution time of arbitrary matrix multiplication tasks. Thorough experimental results reveal that the proposed algorithm demonstrates higher accuracy than a state-of-the-art machine learning task performance estimation engine, Ernest, by decreasing the Mean Absolute Percentage Error (MAPE) in half.

Index Terms—Big data analytics, Distributed matrix multiplication, Performance modeling, Cloud computing

1 Introduction

Recent improvements in hardware and software systems have made it possible to process large-scale datasets that were previously infeasible. To accommodate the increasing number of big-data analysis applications, systems are increasingly deployed to cloud computing environments which provide scalability and fault tolerance while reducing the overheads incurred by operational tasks. Cloud computing services offer various instances with distinct hardware configurations, and many big-data processing software platforms utilize those resources in a scale-out manner [1], [2], which reduces the efforts required for managing large-scale distributed computing resources and allows application developers to focus only on critical tasks.

In many data mining algorithms, Matrix Multiplication (MM) is a core computation kernel that is well known for incurring significant computational overheads. For example, the multiplication of two dense matrices is the dominant overhead during the feedforward and backward propagation steps in a deep neural network [3]. Similarly, the core computational kernel in a multilayer perceptron classifier involves multiplication of two dense matrices, i.e., features of the input mini-batches and weight vectors of intermediate layers [4]. In many recommendation systems, the main kernel task of matrix factorization algorithms such as Nonnegative Matrix Factorization (NMF) [5] and SVD [6] is dense-dense MM. Thus, to help users select a cost-efficient environment for data mining job execution, it is essential to estimate the MM performance on diverse cloud computing environments that provide various instance types and different number of worker nodes to execute a job.

However, despite the importance of an MM task in machine learning, comprehensive performance analysis and modeling of a distributed cloud computing environment have not yet been thoroughly conducted. Some methods, e.g., Ernest [7], CherryPick [8], and PARIS [9], focus on predicting machine learning task performance in a cloud computing environment. These methods rely on scale-based sampling to estimate the latency of completing a task with the entire original input dataset while showing sufficient accuracy for some machine learning tasks. However, they exhibit poor performance in predicting the latency of distributed MM tasks because the algorithms fail to capture the complexity of the task.

In this paper, we propose the MPEC algorithm to predict the latency of MM tasks with arbitrary shapes and sizes when they are executed using multiple cloud instance types with a various number of worker nodes. The goal of this paper is not to optimize the performance of MM tasks on a cloud computing environment. Many studies have investigated performance optimization and modeling of MM tasks on multi-core shared memory machines using hardware-optimized libraries, such as OpenBLAS [10]. Rather than focusing on a problem of improving MM task performance for specific cases, this study aims to solve a new challenge arising from the era of cloud computing. In using cloud computing resources, users have many options of machine type selection with almost infinite scalability; at the time of writing, AWS provides over 80 instance types, and non-

- *Jeongchul Kim and Kyungyong Lee are with the Department of Computer Science, Kookmin University, 77, Jeongneung-ro, Seongbuk-gu, Seoul, Republic of Korea. E-mail: kjc5443@kookmin.ac.kr, leeky@kookmin.ac.kr (Corresponding author: Kyungyong Lee)*
- *Myungjun Son is with the Department of Computer Science and Enginnering, Penn State Unversity, Unversity Park, USA. This work was done when he was affiliated with Kookmin University. E-mail: mjson@psu.edu*

system experts may have difficulties comprehending the characteristics of each instance type.

To achieve the goal of modeling and predicting the performance of diverse MM tasks on cloud resources, we propose 16 features to model MM task characteristics on the cloud, such as the total number of multiply operations, shuffle overheads, and hardware specifications. Using these selected features, an ensemble model was built using multiple gradient boosting (GB) regressors [11]. To predict the latency of an arbitrary MM task when the number of worker nodes changed, the proposed MPEC algorithm logically partitions the original matrices of an MM task and synthetically generates an MM task whose latency can be predicted by using the model we propose. It multiplies the amount of overhead by the predicted latency of the synthetically generated MM task to estimate the response time of the original MM task. Furthermore, we propose an intelligent heuristic to generate various MM task training datasets to mitigate offline experiment costs. In the heuristic, we use the Latin Hypercube Sampling (LHS) [12] algorithm to generate various exclusive experiment cases from a few representative MM workloads. We apply the D-optimal algorithm [13] to the generated cases to select optimal experiment scenarios without sacrificing model prediction accuracy.

Thorough experiments reveal that the latency prediction accuracy of the proposed algorithm demonstrates an error rate less than 10% MAPE regardless of target cloud instances, and it outperforms other predictor models that employ a linear scaler. Applying the LHS and the D-optimal algorithm to generate optimal training datasets significantly reduces the overhead of running many offline experiments by 90% with only marginal degradation of prediction accuracy. The proposed Design of Experiments (DoE) method [13] generates a unique combination of cloud instance types and MM scenarios, and it allows the prediction of MM task latency for instance types that the experiments have not been undertaken. Comparing the proposed method to Ernest [7] which is a state-of-the-art machine learning task performance estimator, it is revealed that MPEC provides 9% less MAPE on average for a diverse set of MM task latency predictions. Furthermore, compared to most other performance estimator systems that rely on a scale-based sampling methods and require separate experiments for different types of input datasets and cloud instance types, the proposed MPEC algorithm can reuse experimental results from previous runs and significantly improve experimental efficiency. In summary, major contributions of this paper are as follows

- Characterization of distributed MM tasks and cloud computing instance types
- Proposal of unique features to effectively represent distributed MM tasks
- Prediction of MM task latency across different cloud instance types
- Suggestion of a scale-out algorithm to predict MM task latency with different numbers of worker nodes
- Employing an intelligent algorithm to generate comprehensive MM task scenarios while minimizing redundant experiment cases

The rest of this paper is organized in the following way. Section 2 provides motivating examples of dense MM tasks and challenges that we want to solve. Section 3 analyzes characteristics of dense MM tasks on distributed cloud environments. Section 4 describes the architecture and algorithms of the proposed MPEC system. Section 5 explains a heuristic to generate optimal training datasets of distributed MM scenarios. Section 6 thoroughly evaluates MPEC. Section 7 covers related works, and Section 8 concludes this paper.

2 Background and Motivation

MM tasks are widely in-use for many machine learning jobs on cloud. We present data mining jobs that use dense MM as the core computation kernel and how the distributed dense MM performance on cloud can vary for different cloud configurations and MM scenarios.

2.1 Dense Matrix Multiplication For Data Mining Jobs

Many data mining jobs utilize dense MM as a core computation kernel to construct a model with minimized prediction error. Figure 1 illustrates the NMF [5] algorithm, which is widely used in recommendation systems. The NMF algorithm factorizes an input sparse matrix A into two dense matrices W and H without a negative element. Let us assume that the dimension of input sparse matrix A is $M \times N$, and the rank is K. Review scores can therefore be well represented in this format, where M is the number of users, N is the number of items, and K is the number of hidden factors that we want to embed in the model. In Figure 1, a factorized matrix W (dimension: $M \times K$) represents the relation between each user and hidden factors, and factorized matrix H (dimension: $K \times N$) represents the relation between each item and hidden factors. Matrices W and H are computed iteratively by multiplying input matrix A, W, and H of previous iterations. In the numerator of the NMF formula used to calculate H_{new} matrix, the current W^t and A are multiplied (dense-sparse MM). In the denominator, W^t and W matrix are multiplied (dense-dense MM) followed by multiplying H_{old} (dense-dense MM).

As another example, Multi-Layer Perceptron (MLP) classifiers heavily rely on dense-dense MM. The MLP algorithm (Figure 2) comprises multiple hidden layers where each layer is composed of multiple neuron nodes. The nodes in each layer are fully connected to nodes in the next layer where linear combinations of weight vectors are applied (feed-forward propagation). The MLP classifier performs back-propagation to minimize classification errors. In the MLP feedforward step, dense MMs occur between each layer. For example, an input matrix is formed by recording each input record in a row and generating a feature value in each column. Thus, input records with f features that are packed with b mini batches form a $b \times f$ dense matrix. The input data mini batch is fully connected to the first hidden layer with h_1 neuron nodes to form a weight matrix of $f \times h_1$. Dense MM of $(b \times f) \times (f \times h_1)$ occurs in the feedforward propagation step and continues to the output layer.

To measure the importance of a dense MM task quantitatively in real-world data mining applications, we measure

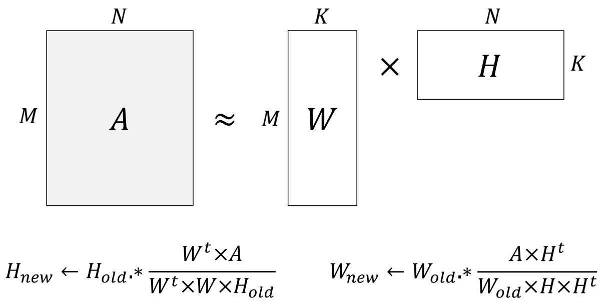

N
M
A
≈
K
M
W
×
N
H
K
$H_{new} \leftarrow H_{old} .* \frac{W^t \times A}{W^t \times W \times H_{old}}$
$W_{new} \leftarrow W_{old} .* \frac{A \times H^t}{W_{old} \times H \times H^t}$

Fig. 1: Dense matrix multiplication application - Nonnegative Matrix Factorization

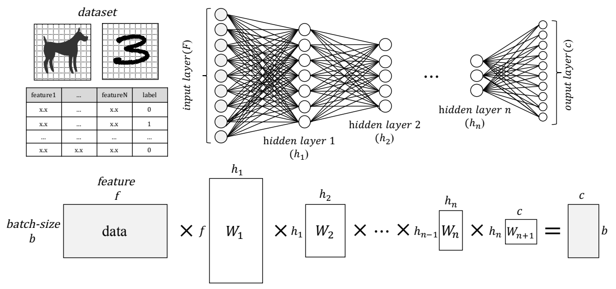
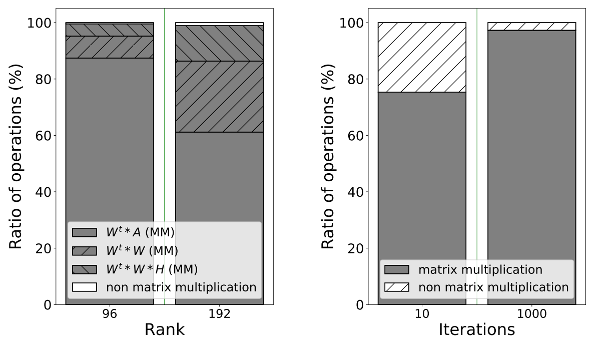


Fig. 2: Dense matrix multiplication application - Multi-Layer Perceptron classifier

the ratio of dense MM time to the overall job completion time (Figure 3). Here, experiments were conducted with four AWS EC2 *R4.2xlarge* instances with Apache Spark 2.2.0 installed on Ubuntu 16.04. Note that OpenBlas [10] is a hardware-optimized library installed on each Spark worker to fully utilize CPU cores fully.

Figure 3a shows the percentage of MM task time required to complete an NMF job. For the NMF job, we used the MovieLens[1] dataset that contains user-movie rating information. In the experiment, we used *ml-latest* dataset which contains 270,000 users, 45,000 movies, and 26,000,000 ratings. We implemented the NMF algorithm on Spark with provided linear algebra libraries [14][2]. To measure the dense MM latency precisely, we invoke the *take(1)* Spark action to execute each transformation. In the NMF implementation, $W^t \times A$ and $A \times H^t$ are sparse-dense MM. Initially, we attempted to implement the task as sparse matrix × dense MM. However, this did not perform well compared to an implementation that transforms the sparse matrix into a dense matrix to perform dense × dense MM. We suspect the reason for the poor performance was a lack of an optimized library for sparse matrices. This observation conforms to the experimental results presented by Liu. et. al. [15]. Furthermore, at the time of writing, the Spark linear algebra library does not support sparse × dense MM natively; the sparse matrix is transformed into a dense matrix internally prior to the multiplication. In the experiment, we updated the W and H matrix for five iterations with different rank values (96 and 192). Overall, the dense MM takes approximately 98% of the total NMF job completion time.

1. https://grouplens.org/datasets/movielens/
2. https://github.com/kmu-bigdata/dense-mm-workload



(a) Nonnegative Matrix Factorization (b) Multi-Layer Perceptron classifier

Fig. 3: Importance of the dense matrix multiplication task in real-world data mining jobs

The remaining time is taken to read input data, write the outcome, transpose the matrix, and perform element-wise matrix operations.

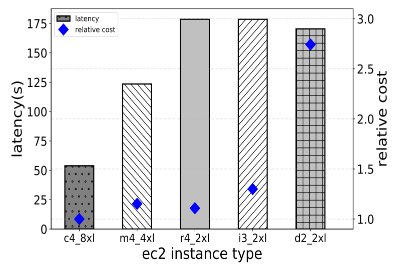
Figure 3b shows the ratio of dense MM tasks to complete the MLP classifier. In this experiment, we used the MNIST dataset[3], which contains 60,000 hand-written number images each of which has 780 features (image pixels). We set the mini-batch size to 2002, with three hidden layers (196, 100, and 25 neuron nodes) and a single output layer with ten classes. In Figure 3b, the horizontal axis is the number of iterations, and the gray bar indicates the ratio of dense MM tasks. When the number of iterations is 10, the dense MM task is not dominant. However, as the number of iterations increases (1,000), dense MM takes approximately 97% of total MLP classifier execution time. As the number of iterations increases, we could achieve better training accuracy with denser MM tasks.

From the experimental results shown in Figure 3, we can observe the importance of dense MM task for real-world data mining applications. It is evident even for NMF implementation with a sparse input dataset, it is evident that dense MM tasks are essential to expedite the completion of data mining jobs with sparse inputs.

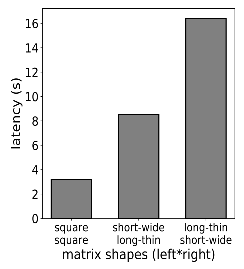
2.2 Distributed MM Performance Variation on Cloud

When running distributed MM tasks on a cloud computing environment, many factors can impact the overall response time. To understand the peculiarity of MM performance in a distributed cloud computing environment, we analyzed the performance of dense MM with various input sizes on distinct cloud computing instances. In the experiments, we multiplied two square matrices (32000, 32000, 32000) on AWS EC2 *C4.8xlarge, M4.4xlarge, R4.2xlarge, I3.2xlarge*, and *D2.2xlarge* instances that have unique hardware features with hourly on-demand price of $1.591, $0.8, $0.532, $0.624, and $1.38, respectively[4]. We chose these instance types, because they have a similar memory size of around 60GB which can execute the given MM workload. We used the

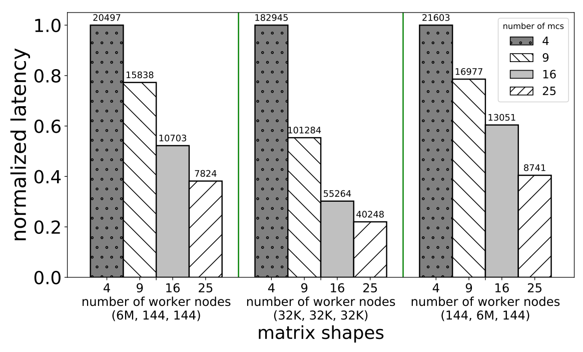
3. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html
4. As of Aug. 2019, with the Linux OS in the US West (Oregon) region



(a) Different EC2 instances



(b) Different matrix shapes



(c) Different number of machines

Fig. 4: Normalized execution times of MM tasks with different shapes and number of machines (lower is better)

Apache Spark MLlib BlockMatrix library [14] to conduct the experiments with four machines with OpenBLAS installed.

In Figure 4a, the horizontal axis represents instance types, and the bar values in the primary vertical axis indicate the latency to complete a MM task for each instance type. The secondary vertical axis shows the normalized cost of running EC2 instances (price $\times$ running time); the value is marked with blue diamond marks. As shown in the figure, distinct instance types result in significantly different latency and cost performance, and users should be cautious of this when attempting to create an optimal execution environment. Figure 4b shows the impact of matrix shapes on the latency to complete a MM task. In the figure, the horizontal axis represents input matrix shapes and the vertical axis shows MM task latency. The multiplication of two non-square matrices exhibits a significantly different performance compared to square MM even when the number of multiply operations is the same, i.e., the number of left matrix rows $\times$ left matrix columns $\times$ right matrix columns.

Figure 4c shows the latency of three different MM tasks when they are executed with a different number of Spark worker nodes. In this experiment, we created an Apache Spark cluster using AWS EC2 *R4.2xlarge* instances with different numbers of worker nodes (4, 9, 16, 25). Here, the four left-most bars show the latency of completing an MM task with a large left row, and the middle four bars represent an MM task with two square matrices. The last four columns show MM tasks, where the left column (or right rows) is large. The numbers in parentheses in the horizontal axis are the number of left rows, left columns, and right columns. As can be seen, MM task latency decreases as the number of worker nodes increases; however, latency improvement rate differs with different matrix shapes. For example, when the left column is large, the change ratio is 1.0, 0.79, 0.6, and 0.4 as the number of Spark worker nodes changes to 4, 9, 16, and 25, respectively. For the square matrix, the change ratio is 1.0, 0.55, 0.3, and 0.22.

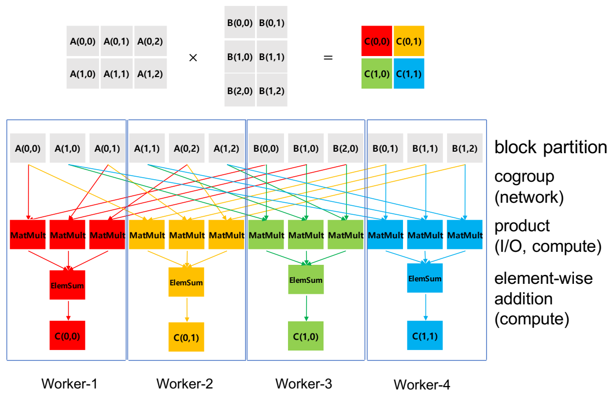


Fig. 5: Block-based distributed matrix multiplication and related overhead in each step. Network, disk I/O, and CPU are the principal resources of the execution.

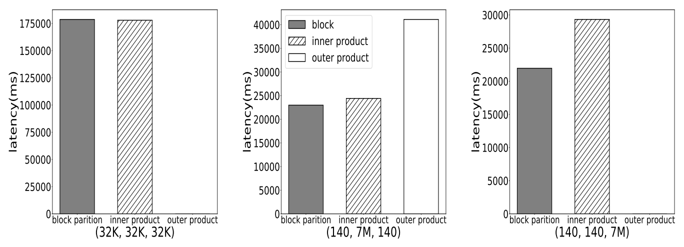
In summary, considering there are more diverse matrix shapes in realistic MM tasks and instance types than those presented in Figure 4, estimating the latency of MM tasks with matrices of various shapes and sizes can be very challenging.

3 Characteristics of Matrix Multiplication in Distributed Computing Environments

The optimization of distributed MM has been well studied in the literature. In the HPC community, many studies have focused on minimizing communications cost using the MPI model. The representative methods include SUMMA [16] and CARMA [17]. These methods demonstrate optimal performance mainly on multi-core shared memory machines by carefully designing an algorithm considering the underlying hardware specifications. The proposed MPEC algorithm is complementary to previous MM performance optimization work in a cloud computing environment. Previous work on the cloud focused on solving an optimization problem specific to an instance type, while this proposed algorithm tries to solve a different problem of finding an optimal instance type to complete an MM task. Using previous optimization studies with this proposed algorithm together can provide more cost and performance efficient environments.

Apache Spark [2] is widely used to conduct various machine learning tasks on a distributed cloud computing environment to process large-scale datasets. The primary abstraction in Spark is a Resilient Distributed Dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines. Spark manages large-scale data using partitions that help to parallelize distributed data processing while guaranteeing fault tolerance with lineage and task execution optimization via lazy evaluation [2].

In Spark, matrix-related linear algebraic operations are supported in the MLlib library [14] with various matrix-partitioning schemes (row- and block-based) and a set of



(a) square-square MM (b) left columns large (c) right column large

Fig. 6: Comparison of partition scheme in matrix multiplication

distributed operation APIs on the matrix. To multiply two matrices, Spark MLlib automatically identifies the optimal way to distribute tasks based on the input matrix-partitioning scheme and uses the Scala Breeze library to perform multiplication. Consider $C = A \times B$, i.e., the multiplication of two matrices. If A is row-partitioned and B is column-partitioned, the Cartesian product is performed for each row block of A and column block of B. If both A and B are block-partitioned, the block dimension of the resulting matrix C is determined by considering the number of worker nodes and input block dimensions. A worker node that is responsible for each resulting block fetches all the necessary blocks from A and B to execute a multiplication operation locally.

Figure 5 shows an example of block-based MM. Here, the left matrix A, right matrix B, and the result matrix C are 2×3, 3×2, and 2×2 block matrices, respectively. In Spark, a *cogroup* operation allows a worker node that is responsible for the result block, to collect the necessary left and right matrix blocks by block IDs. During a *cogroup* operation, network overhead is dominant. After collecting all necessary blocks, each worker node performs product operations, followed by element-wise addition operations. In this step, the I/O overhead that reads the fetched blocks from a *local.dir* location and the computational overheads are dominant.

Other than block-based partitioning in Figure 5, row- and column-based partitioning can also complete a distributed MM task. Row-partitioning the left matrix and column-partitioning the right matrix allows it to perform *inner-product* between rows of the left matrix and columns of the right matrix that generates complete elements in an output matrix. *Outer-product* is doable by column-partitioning of the left matrix and row-partitioning of the right matrix. Multiplying a column in the left matrix to a in the right matrix results in a matrix with an output matrix size, and outputs from each multiplication should be summed to complete an MM task.

To compare the impact of distributed matrix partitioning for the MM task performance, we conduct various MM tasks using Apache Spark with four *AWS EC2 R4.2xlarge* instances. In Figure 6, we differentiate MM task scenarios expressed as (left matrix rows, left matrix columns, right matrix columns). In each scenario, we show the latency when using *block partition, inner-product*, and *outer-product*. Overall, block-partitioning shows the best performance compared to other methods. The inner product shows close performance for a square-square case (Figure 6a), but block-partitioning performs better in other cases. We observed that the outer product could not complete some MM tasks due to an *out-of-memory* error while storing a large output matrix in an executor. As shown in Figure 6, block-based partitioning shows the best performance. Furthermore, the row- or column-partitioning may perform poorly when the operands of a multiply operation changes (such as $A \times B$ and $B \times A$) because this will change the partitioning scheme of the left and right matrices. Therefore, this paper focuses on modeling and predicting MM task performance of block-based partitioning because it shows good performance with generality.

4 Distributed MM Performance Estimator

Many cloud computing service providers offer many diverse cloud computing instances with almost infinite scalability for users to select based on their demands. However, users may have difficulty in setting an optimal environment for a variety of tasks. This study aims to provide key indications to help establish an optimal cloud environment for various machine learning jobs in which the core kernel is the distributed MM tasks. To achieve the goal, we present an MPEC algorithm that predicts the latency of distributed MM tasks of arbitrary shapes of sizes in scale-out environments with various instance types. Formally, we predict the completion time of an MM task where the left, right, and result matrices are denoted as A, B, and C, respectively ($A \times B = C$). The numbers of rows and columns of A are denoted as LR and LC, respectively, and a pair is denoted as (LR, LC). Similarly, the dimensions of the right and result matrices are denoted as (LC, RC) and (LR, RC), respectively. Note that the number of columns in the left matrix and the number of rows in the right matrix should be equal, and we denote both as LC. The first goal of this paper is to build an arbitrary MM task latency prediction model, f, by using train datasets that are generated from W_{trn} Spark worker nodes. The next goal of this paper is to predict the latency of arbitrary MM tasks when the number of worker nodes changes, and we denote the number of worker nodes that we want to predict as W_{prd}, where $W_{trn} \neq W_{trn}$ (generally, $W_{prd} > W_{trn}$). If the number of worker nodes is not relevant to the prediction or training step, the number of worker nodes is simply denoted as W. Unless otherwise stated, we assume a case where the number of worker nodes is W_{trn} when we discuss MM latency prediction.

We assume that A, B, and C are block-partitioned matrices. According to the task distribution algorithm of Apache Spark MLLib [14], it is better for the number of output matrix blocks to be equal with the total number of worker nodes [18]. Thus, we consider cases where the number of worker nodes equals the number of output matrix blocks. For load-balancing, we constrained the number of Spark worker nodes (W) to the square of an integer to evenly partition the rows and columns of the input and output matrices [18]. Under this constraint, the block size of left matrix row, left matrix column, and right matrix column are $\frac{LR}{\sqrt{W}}$, $\frac{LC}{\sqrt{W}}$, $\frac{RC}{\sqrt{W}}$, respectively, and we name each block size as lr, lc, and rc, respectively. To express block size of row

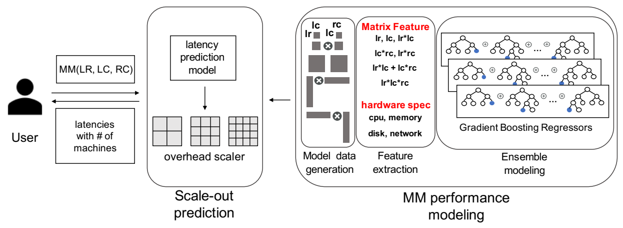


Fig. 7: Proposed system architecture

and column together with W workers, we use $BS_W(r, c)$. The notations used in this paper are summarized in Table 1.

The overall architecture of the proposed system is shown in Figure 7. The system is composed of MM performance modeling and a scale-out prediction step. In the MM modeling step, MPEC builds an MM task latency prediction model, f, with training datasets generated from W_{trn} Spark worker nodes by proposing unique features to represent characteristics of distributed MM tasks. The scale-out prediction step provides expected latency of the input MM task when the number of worker nodes increases or decreases, i.e., $W_{prd} \neq W_{trn}$.

4.1 Matrix Multiplication Performance Modeling

MM performance modeling comprises the generation of training datasets, feature extraction (numbers of rows and columns of the left matrix, the number of columns of the right matrix, and various hardware features), and modeling. Using the features, we propose to build an ensemble model using bagging [19] with GB regressors [11], a tree-based non-linear model algorithm, to predict the latency of unseen scenarios in MM tasks.

4.1.1 Training Data Generation

In the training dataset generation step, the proposed method measures the latency of various shapes and sizes of MM tasks that can be broadly categorized into square $\times$ square, long-thin $\times$ short-wide, and short-wide $\times$ long-thin tasks. To cover the wide range of shapes and sizes of MM tasks, this step synthetically generates arbitrary MM workloads from each category. To record the latency of various MM tasks on different instance types, the proposed method performs offline experiments in a Spark cluster with W_{trn} workers of various instance types. We use the Spark web UI REST API to gather execution performance metrics.

4.1.2 Feature Extraction

As discussed in Section 3, the MM overhead in a distributed computing environment involves various resources. To account for diverse overheads, the proposed method utilizes the dimensions and products of input matrix block sizes, i.e., lr, lc, rc, $lr \times rc$, $lr \times lc$, $lc \times rc$, $lr \times lc + lc \times rc$, and $lr \times lc \times rc$, as features to model MM performance. Here, the $lr \times rc$ term represents the size of the output matrix, the $lr \times lc$ and $lc \times rc$ terms represent the size of the left and right matrix blocks, respectively, where the size impacts the network and I/O disk overheads due to shuffling. The $lr \times lc \times rc$ term represents the total number of multiply operations.

Notation	Meaning
A	the left matrix to multiply
B	the right matrix to multiply
C	the output matrix of multiplication of $A \times B$
LR	the total number of rows in the left matrix
LC	the total number of columns in the left matrix
RC	the total number of rows in the right matrix
lr	the block size of left matrix row
lc	the block size of left matrix column
rc	the block size of right matrix column
W_{trn}	the number of worker nodes used in train data generation
f	MM latency predict model. Train data from W_{trn} workers
W_{prd}	the number of workers to predict MM latency ($\neq W_{trn}$)
(r, c)	a pair of value that corresponds to row and columns
BS_W	block size of a matrix with W workers expressed in (r, c)
T_W	the expected MM latency when the number of workers is W

TABLE 1: Distributed MM task notations

In addition to MM task features, we also generate hardware specification features to predict MM latency in various EC2 instance types. Among the instance types that AWS provides, we select five distinct instance types that emphasize different characteristics. They are M4, C4, R4, I3, and D2. *M4* is a general-purpose instance type, *C4* is a compute-optimized instance type with superb CPU performance, and *R4* is a memory-optimized instance type with a higher RAM size. *I3* and *D2* instances are storage-optimized instance types with a large volume of storage; *I3* instances equip local NVMe SSDs while *D2* instances equip local HDD storage. Instead of using hardware specifications provided by AWS, we measured the performance of CPU, memory, disk, and network performance to reflect them as detailed features in a realistic scenario [20]. To create CPU-related features, we reference the */proc/cpuinfo* file that provides processor information with a number of CPU cores, cache size, and speed that we refer to as *vCPU*, *cache_size*, *cpu_mhz*, respectively. A Linpack [21] benchmark measures how fast a computer solves linear algebra problems. To use them as a feature, we set the input matrix size to $10,000$ and measured GFLOPS and use it as a $linpack$ feature. To measure the performance of the disk IO, we create features of *disk_read_bandwidth* and *disk_write_bandwidth* using the *dd* system command. We performed the writing and reading of a 1GB file into a directory that a Spark executor uses as *spark.local.dir*. To create a network-performance related feature, we measure network bandwidths using *iPerf3* command and use the output as a *network_bandwdith* feature. For the memory feature, we use the configured memory size of each instance. After measuring the feature values, we apply min-max scaling to force the value between 0.0 and 1.0 to avoid the unexpected impact of different original feature value ranges.

Unlike the proposed MPEC system, previous methods that focus on predicting the performance of data mining tasks on cloud computing resources use a scale-based sampling mechanism to generate input features for modeling [7], [8], [9]. Generally, these methods select a considerably small portion of the input dataset and measure performance using a subset of the dataset. Using the outcomes from the sample dataset, these methods apply distinct predictive algorithms, e.g., a non-negative linear equation (Ernest [7]), Bayesian optimization (Cherrypick [8]), and random forest (PARIS [9]) to make a prediction. However, the scale-based sampling mechanism cannot capture the complex nature of the distributed MM tasks, and it considers either lr or rc based on the sampling method. Furthermore, MPEC uses more detailed hardware specifications as features. Accordingly, the proposed method in this paper demonstrates superior performance owing to its rich set of features (Section 6).

4.1.3 Modeling Distributed Matrix Multiplication Task Latency

In this step, the proposed method builds a model to predict the performance of multiplying various matrices. The proposed method utilizes an ensemble [19] of GB regressors [11] as a model.

GB [11] is a flexible non-parametric statistical learning approach for classification and regression. The main idea behind GB is combining multiple weak learners that are generally applicable to only simple linear relations incrementally to model complex and non-linear interactions among features. A GB model is fitted in a forward stage-wise pattern, where at each stage, a new weak learner model is fitted to the residual of the current model; the model focuses more on correcting errors from the previous iterations. Unlike a similar decision tree method, the GB method is robust against overfitting as it creates an ensemble of many weak learner models [22].

Ensemble models in machine learning combine the decisions from multiple models to improve overall predictive performance. We use the bootstrap aggregating (bagging) method [19] which involves making each model in the ensemble have the same weight to improve the stability and accuracy of the regressor. It averages the model to reduce variance and helps to avoid over-fitting which improves generality. In our proposed system, we use three GB regressors with different hyper-parameters of number of estimators and loss function. Son et. al. [23] proposed to use a GB regressor for a similar purpose, but using an ensemble of multiple GB regressors can improve the generality of the model and accuracy.

4.2 Matrix Multiplication Performance Prediction on a Scale-out Environment

In a cloud computing environment, users can easily add or remove computational resources, but estimation of performance impact due to changes is very challenging as there are many factors that might cause differences in performance. Scale-out prediction modules in MPEC are responsible for predicting MM task latency when the number of worker nodes changes in a Spark cluster.

4.2.1 Characters of Distributed Matrix Multiplication Overhead

In MM task performance prediction, scaling the number of worker nodes linearly cannot capture the non-linear interactions of various characteristics that influence the overall MM task completion time. To present the characteristics of MM task overheads as W changes, we summarize three dominant overhead factors in each worker node that is responsible for an output block in Equations 1 (number of products), 2 (shuffle amounts), and 3 (output block size).

$$\begin{aligned} Product(W) &= \frac{LR}{\sqrt{W}} \times \frac{LC}{\sqrt{W}} \times \frac{RC}{\sqrt{W}} \times \sqrt{W} \\ &= \frac{LR \times LC \times RC}{W} \end{aligned} \tag{1}$$

$$\begin{aligned} Shuffle(W) &= \frac{LR}{\sqrt{W}} \times LC + LC \times \frac{RC}{\sqrt{W}} \\ &= LC \times \frac{(LR + RC)}{\sqrt{W}} \end{aligned} \tag{2}$$

$$OutputBlockSize(W) = \frac{LR}{\sqrt{W}} \times \frac{RC}{\sqrt{W}} = \frac{LR \times RC}{W} \tag{3}$$

Here, as W increases, the number of product operations and the output block size overhead in each worker node responsible for a resulting block is proportional to $\frac{1}{W}$, while the shuffle overhead to complete an output block is proportional to $\frac{1}{\sqrt{W}}$. Due to differences in the overhead change ratio, distinct overheads must be treated differently as W changes.

4.2.2 Predicting Matrix Multiplication Latency on a Scale-out Environment

We propose a novel *unit block-based MM latency estimation algorithm* that considers various overhead factors in MM tasks on a scale-out environment. To predict the latency of an MM task of $A \times B$ with W_{prd} workers, we first block-partition A and B, where the block sizes are $BS_{W_{prd}}$ and the number of blocks in each dimension is $\sqrt{W_{prd}}$. Note that we refer to these partitions as *unit-blocks*. Due to the non-linear interactions of multiple factors in MM task overheads, we cannot directly use the expected outcomes from the MM latency prediction model, f, which are generated using MM train datasets obtained with W_{trn} workers (presented in Section 4.1). Instead, we generate a synthetic MM task with the same unit-block size ($BS_{W_{prd}}$) but different numbers of workers (W_{trn}) and number of blocks ($\sqrt{W_{trn}}$).

Here each output block size of the synthetic MM task is the same as that of the original MM task we aimed to predict (i.e., $BS_{W_{prd}}$); however, the total number of products required to conduct and shuffle overheads differs as the sizes of the left and right matrices differ. Formally, the total number of products of an MM task synthetically generated from the unit-block is proportional to $BS_{W_{prd}} \times \sqrt{W_{trn}}$, while that of the original MM task is proportional to $BS_{W_{prd}} \times \sqrt{W_{prd}}$. Relative to the shuffle overhead, each output block must fetch the $\sqrt{W_{trn}}$ rows from A and $\sqrt{W_{trn}}$ columns from B in the synthetic MM tasks, while the original MM task must fetch the $\sqrt{W_{prd}}$ rows from A and $\sqrt{W_{prd}}$ columns from B. After obtaining the expected latency of executing an MM task of $BS_{W_{prd}}$ with W_{trn} workers using f, the proposed algorithm multiplies $\frac{\sqrt{W_{prd}}}{\sqrt{W_{trn}}}$ by the expected latency of the synthetic MM task to calculate the expected latency of the original MM task. Here, the rationale is that MM tasks with equal output block size and a different number of worker nodes incur proportional shuffle and compute overheads that must be processed by

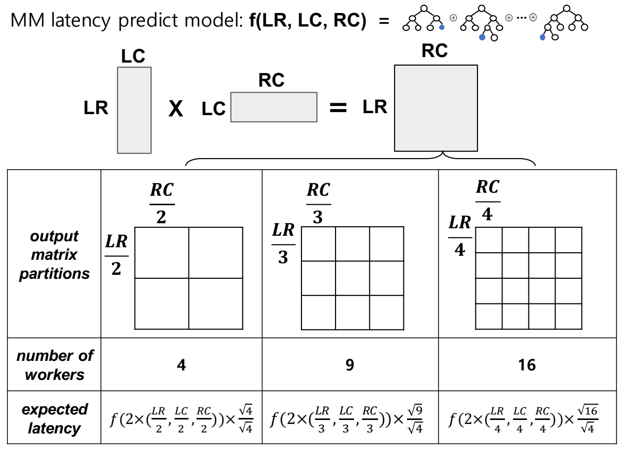

MM latency predict model: f(LR, LC, RC) =
LC
LR
X
RC
LC
=
RC
LR
output matrix partitions
RC/2
LR/2
RC/3
LR/3
RC/4
LR/4
number of workers
4
9
16
expected latency
$f(2\times(\frac{LR}{2},\frac{LC}{2},\frac{RC}{2}))\times\frac{\sqrt{4}}{\sqrt{4}}$
$f(2\times(\frac{LR}{3},\frac{LC}{3},\frac{RC}{3}))\times\frac{\sqrt{9}}{\sqrt{4}}$
$f(2\times(\frac{LR}{4},\frac{LC}{4},\frac{RC}{4}))\times\frac{\sqrt{16}}{\sqrt{4}}$

Fig. 8: Scaling-out prediction example

each worker node, and we multiply the ratio of shuffle and compute overheads by the expected latency of the synthetic MM task. The proposed algorithm is summarized in Equation 4.

$$T_{W_{prd}} = f(\sqrt{W_{trn}} \times BS_{W_{prd}}) \times \frac{\sqrt{W_{prd}}}{\sqrt{W_{trn}}} \tag{4}$$

Figure 8 shows an example of MM task latency estimation with a different number of worker nodes. In this example, we assume that the dimensions of A, B, and C are (48000, 24000), (24000, 36000), and (48000, 36000), respectively, and we use a constructed model using train datasets obtained with four Spark worker nodes, i.e., $W_{trn} = 4$. With this example, we demonstrate how we can predict the latency of an MM task executed with nine nodes. When $W_{prd} = 9$, the BS_9 of the left, right, and output matrices are (16000, 8000), (8000, 120000), and (16000, 12000), respectively. A synthetic MM task with the same unit-block size as BS_9 that runs with $W_{trn} = 4$ workers becomes (32000, 16000), (16000, 24000), and (32000, 24000). We predict the MM task latency of the synthetic case using model, f, which returns $T_4 = f(32000, 16000, 24000)$. Then, T_4 is multiplied by $\frac{\sqrt{9}}{\sqrt{4}}$ to obtain the expected latency when the original MM task is executed with nine worker nodes.

Algorithm 1 MM task latency prediction

```
1:  function PREDICTLATENCY($(LR, LC, RC)$, $W_{prd}$)
2:      if $W_{prd} = W_{trn}$ then
3:          return $f(LR, LC, RC)$;
4:      else
5:          ub = $(\frac{LR}{\sqrt{W_{prd}}}, \frac{LC}{\sqrt{W_{prd}}}, \frac{RC}{\sqrt{W_{prd}}})$
6:          mat = $\sqrt{W_{trn}} \times$ ub
7:          $T_{trn}$ = PredictLatency(mat, $W_{trn}$)
8:          ratio = $\frac{\sqrt{W_{prd}}}{\sqrt{W_{trn}}}$
9:          return (ratio $\times T_{trn}$)
10:     end if
11: end function
```

Algorithm 1 shows the sequence of the scale-out prediction step, i.e., the *PredictScaleOutLatency* function. Here, lines 2-3 mean that if the number of worker nodes a user wants to predict is the same as the number of worker nodes used to build train datasets, the algorithm returns the predicted latency from a constructed model, f. Otherwise, lines 4-10, calculate the block size of the input matrix (Line 5). Here, a synthetic MM task is created with the calculated block size assuming the number of worker nodes is the same as that in the train dataset generation (Line 6). The latency of the synthetically generated MM task (Line 7) is then predicted. The overhead ratio is also calculated by considering the number of blocks of the original and synthetic tasks (Line 8). In Line 9, the algorithm multiplies the overhead ratio by the predicted latency to calculate the expected latency of the input MM task.

5 Generating Optimal Set of MM Experiment Scenarios

To build an accurate model to predict the latency of arbitrary MM tasks, it is important to generate well-represented train datasets. Model accuracy is likely to improve as more data points are added to the training dataset, but it might be too expensive to perform offline experiments with a large number of MM task cases on cloud instances with different types. To overcome the issue, we propose an intelligent algorithm of generating a diverse MM task scenario and choose optimal cases among them to make offline experiments compact.

5.1 Generation of Exclusive MM Experiment Cases

In the step of diverse MM task scenario generation, memory requirements of the underlying Spark cluster should be satisfied. Despite having an equal number of total products, one scenario may work, and another may not. For example, a (10^2, 10^8) × (10^8, 10^2) MM task might work, however, a (10^5, 10^2) × (10^2, 10^5) task might cause an out-of-memory issue because the output block size is significantly larger ($10^5, 10^5$) than that of the former ($10^2, 10^2$) even though the total number of products is the same (10^{12}). We employ the LHS algorithm to generate diverse MM experimental cases while satisfying memory constraints.

LHS is a type of space-filling design that spreads design points almost evenly or uniformly throughout the entire experimental space [13]. These designs are decent choices if the experimenter believes that interesting effects are likely to be found in different regions in the experimental space. With LHS, the basic idea is to obtain a sampling point distribution close to the probability density function of the user's choice [12]. Thus, if one chooses uniform distribution, LHS can spread samples more evenly across all possible values. If we designate F features and N sample points to generate each feature, LHS partitions each feature distribution into N intervals of equal probability and selects one sample from each interval. It then shuffles the sample for each feature such that there is no correlation between inputs. The output of LHS scales values are in the range of 0.0 to 1.0. By multiplying the output of the LHS algorithm with an arbitrary number of samples K, the values of the new sample points would be less than K and evenly distributed in an interval 0 to K.

To apply the LHS algorithm to MM task scenario generation, we multiply the N-three dimensional LHS algorithm

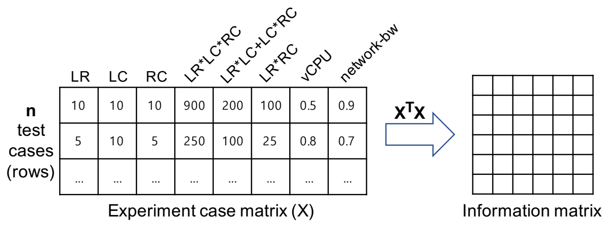


Fig. 9: Building an information matrix from experiment scenarios

output to LR, LC, and RC of the input MM task, generating N combinations of matrices distributed uniformly between (0, 0, 0) and (LR, LC, RC). We first categorize the values of each dimension of an MM task as High (H), Medium (M), and Low (L), and allocate each value to LR, LC, and RC. We use seven distinct combinations to cover various scenarios, i.e., (H-L-L), (L-H-L), (L-L-H), (M-L-M), (M-M-L), (L-M-M), and square $\times$ square MM. For each combination, we first check the maximum size that can be executed in a Spark environment for the generation of training datasets. For example, the maximum sizes of MM tasks on four worker nodes of an *R4.2xlarge* instance and an aggregated memory size of 244GB, are ($8M$-128-128), (128-$8M$-128), (128-128-$8M$), ($75K$-128-$75K$), ($75K$-$75K$-128), (128-$75K$-$75K$) and ($32K$-$32K$-$32K$). If we generate 35 three-dimensional sample points using the LHS algorithm and multiply each combination by these sample points, we can create 245 ($= 7 \times 35$) evenly distributed combinations while satisfying the memory requirements of the target Spark environment. Using the generated MM task scenarios, final experimental cases are generated by extracting all possible combinations of MM task scenarios with target cloud instance configurations.

5.2 Selection of Representative Experiment Cases

The set of MM task scenarios extracted from the LHS algorithm that satisfy the underlying cluster memory requirement may contain too many scenarios to be executed in the training dataset generation step. To relieve the burden of running too many similar experiments, e.g., linearly dependent cases, we apply the D-Optimal algorithm to select a subset of experimental cases that can represent all scenarios well.

To apply the D-optimal algorithm to generate an optimal set of MM task training data points, we express all experiment cases as a matrix (Figure 9), where each row contains a single MM task experiment case with a cloud instance configuration with distinct features. For the features, we utilized those presented in Section 4.1.2; three features in the prediction model of a distributed MM task (i.e., $LR \times LC \times RC$, $LC \times (LR + RC)$, and $LR \times RC$) and the dimension information of the matrices (i.e., LR, LC, and RC). We denote the entire global experiment case matrix generated by the LHS algorithm as X. The Fisher information matrix [24] of X is expressed as $X^T X$, which is the inverse of the covariance matrix of X. The goal of the D-optimal algorithm is to select z cases among x test cases (i.e., the number of rows in X), where $z < x$, while maximizing information with the selected z experiment cases. Note that the determinant increases as z increases.

To select z samples that maximize the information, the D-Optimal algorithm attempts to maximize the determinant of the Fisher information matrix ($|X^T X|$) [24] with the selected z rows. Maximizing the determinant of the information matrix is equivalent to minimizing the inverse of the information matrix (i.e., the covariance matrix), and this provides a set of experimental scenarios that reduce the variance error term. Alternatively, maximizing the determinant of information matrix results in spreading the experiment cases to the largest volume possible in the experimental region [25]. Besides, the D-optimal DoE algorithm supports the selection of a subset of established experimental cases in irregular spaces [25], and it well fits the selection of optimal MM task scenarios that must meet the memory requirements of underlying clusters.

When selecting the optimal z test cases among x possible experiment sets, we apply the Fedorov algorithm [26] which starts from z randomly selected samples as an experiment scenario Z among all x scenarios in X. It exchanges a sample from Z with points in $X - Z$ (samples not in Z) and stops the exchange when no further profitable exchange is possible. Depending on the randomly selected starting z samples, the Fedorov algorithm may suggest an optimum local set of scenarios. Note that the *AlgDesign* package in *R* provides an implementation of the Fedorov method that employs the D-optimal algorithm [27].

The D-optimal algorithm suggests some optimal experimental cases in the entire experimental space. However, we must still determine how many MM tasks, i.e., z, should be considered when performing experiments to build a model. As z increases, the prediction accuracy of a model improves at the cost of increased overheads in the experiment. To balance between prediction accuracy improvement and overheads, we infer the experiment stopping condition based on the information embedded in the experiment case matrix, Z. Here, we initially set Z to $\emptyset$, and, by incrementing the size of Z, we run the *Fedorov* algorithm to obtain a subset of optimal experiment cases. We then calculate the increase in information by measuring the determinant of the normalized dispersion matrix of Z, i.e., $|\frac{(Z^T Z)^{-1}}{z}|$, which is the exchange criterion of the *Fedorov* algorithm [26]. As we continue to add more experimental MM task cases to Z, we record the delta of improvement in the metric value (the determinant of the normalized dispersion matrix). We stop the experiments when the delta of improvement becomes constant. Here, the rationale is that as we add more experimental cases to Z, the information in the matrix saturates at some points and adding more cases from a saturated point provides marginal prediction accuracy improvement at the expense of increased experimental overheads.

Figure 10 shows the overall process of building MM task scenarios for generating training datasets. After determining baseline representative MM task scenarios, we use the LHS algorithm to generate exclusive ratio sets that are multiplied by the LR, LC, and RC values of the baseline scenarios. To ensure that the generated MM tasks can meet the memory usage requirements of the Spark cluster worker nodes, we constrain the ratio values between 0.0 and 1.0. The MM

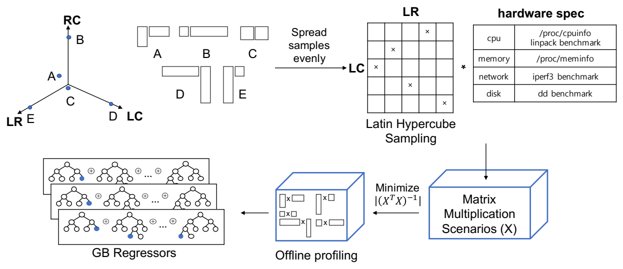


Fig. 10: The procedure of generating optimal training datasets

task scenarios are generated by combining hardware spec features with MM task scenarios after applying the LHS algorithm. Using the generated experimental cases, we apply the D-optimal algorithm to select a subset of MM task scenarios. To determine how many experiments to perform, we calculate the delta of information improvement as we add more experiment scenarios. If the delta of improvement stabilizes, we stop adding experiment cases.

Algorithm 2 Generating representative experiment cases

```
1: $N$ - The number of exclusive test cases to generate
2: $M$ - Representative MM scenarios that satisfy memory requirements
3: $LHS_k$ - $k$ sampling points from LHS
4: $dOptDet(T, k)$ - Returns determinant of the normalized dispersion matrix with $k$ optimal rows from $T$
5: function GenerateExperimentScenario($N$)
6:     $T_i \leftarrow \emptyset, i = 1, \ldots, N$
7:     for each matrix dimension $m \in M$ do
8:         $T \leftarrow T \cup (m \times LHS_{\frac{N}{|M|}})$
9:     end for
10:    $D_i \leftarrow \emptyset, i = 1, \ldots, N$
11:    $D_1 \leftarrow dOptDet(T, 1)$
12:    for $i = 2, \ldots, N$ do
13:        $D_i \leftarrow dOptDet(T, i-1) - dOptDet(T, i)$
14:        if $D_i = D_{i-1}$ then
15:            return $[T_1 \ldots T_{i-1}]$
16:        end if
17:    end for
18: end function
```

Algorithm 2 represents the process of the offline modeling step; *GenerateExperimentScenario* function. Lines 1-4 show prerequisites for the function. Line 6 initializes a variable to store experimental cases. Lines 7-9 fills experimental cases by multiplying LHS generated samples ($LHS_{\frac{N}{|M|}}$) to representative MM task scenarios that meet the memory requirements. Line 10 initializes a variable to store the determinant of information matrix with k rows, and Line 11 calculates the determinant with the minimum number of experimental cases. Line 13 calculates the delta of the determinant of the normalized dispersion matrix returned from the D-optimal algorithm. If the delta becomes constant, the function returns the current set of experimental cases recommended by the D-optimal algorithm.

6 Evaluation

We thoroughly evaluate the performance of the proposed algorithms with various types of AWS EC2 instances, input matrix sizes, shapes, and numbers of worker nodes. In summary, applying an ensemble GB regressor algorithm with the proposed features outperforms a linear regressor and exhibits 79% lower RMSE. Evaluations of the proposed method with various types of cloud instances demonstrates the effectiveness of the proposed algorithms regardless of the underlying instance types. The proposed scale-out performance estimator provides a less than 10% MAPE, which is superior to a linear scaler with a penalty term (29% MAPE).

Regarding the effectiveness of an algorithm in choosing optimal training datasets, generating MM task scenarios using the LHS algorithm outperforms other heuristics, including the expert pick, with at least 52% (relative to $RMSE$) higher accuracy. The D-optimal algorithm reduces the overheads of the experiments by approximately 90% with a less than 0.22% reduction (R^2) in accuracy. Finally, compared to Ernest which is a state-of-the-art machine learning task performance estimator, the proposed algorithm demonstrates 9% (relative to MAPE) improved accuracy on average across diverse MM task scenarios.

6.1 Setup

The experiments were conducted on *R4.2xlarge* (8 vCPUs, 61GB RAM), *M4.4xlarge* (16 vCPUs, 64GB RAM), *C4.8xlarge* (36 vCPUs, 60GB RAM), *I3.2lxarge* (8 vCPUs, 61GB RAM) and *D2.2xlarge* (8 vCPUs, 61GB RAM) AWS EC2 instances that have similar memory configurations. Please note that the feasibility of running given MM tasks rely on the aggregated memory size of a cluster, and we make the total RAM size the same to apply the same set of MM tasks. Using the EC2 resources, an Apache Spark [2] cluster was created with different numbers of worker nodes according to the experimental scenarios. Here, we used Spark MLLib version 2.2.0 to perform linear algebraic operations, and we used the *spark-ec2*[5] tool to deploy the Spark cluster. Each EC2 instance includes OpenBLAS [10], a hardware-optimized linear algebra library to fully utilize the computational capacity of the underlying resources. For all MM task cases, we ran the experiments five times and recorded the median value to remove the impact of unusual noise from the cloud resources. We used *scikit-learn* 0.18.1 with *Python* 2.7.12 to perform modeling and*R* 3.2.4 is used with *AlgDesign* 1.1 to apply the D-optimal selection using the Fedorov algorithm [27]. In the evaluation, we used training and test datasets generated by the algorithm presented in Section 5.

To evaluate the proposed system quantitatively, we used the coefficient of determination (R^2), the RMSE, and the MAPE. As shown in Equation 5, the R^2 metric measures the ratio of resemblance of the predicted outcome to the true value, where the best score is 1.0 (larger values are

5. https://github.com/kmu-leeky/spark-ec2

better). The R^2 metric is widely used to evaluate the quality of regressor prediction.

$$R^2 = \frac{\sum_i (\hat{y}_i - \overline{y})^2}{\sum_i (y_i - \overline{y})^2} \quad (5)$$

The RMSE metric measures the amount of difference between the predicted and true values. Differing from RMSE, the MAPE measures the ratio of difference between the true and predicted values to the true value. In MM latency prediction, the task completion time may vary significantly as we evaluated an extensive set of test cases; thus, presenting both the error ratio and absolute error facilitates a thorough evaluation of the quality of proposed models.

6.2 Matrix Multiplication Performance Modeling Evaluation

We first evaluate the efficiency of MM performance modeling module that aims to predict the MM latency when the number of worker nodes in the training step (W_{trn}) are same as the ones in the prediction step (W_{prd}).

6.2.1 Feature Importance

The proposed method uses various combinations of left and right matrix block sizes and instance specifications as features to model complex distributed MM operations. In Figure 11, we visualize the relative importance of the proposed features calculated by counting the number of times a feature is selected for splitting a GB regressor tree, where each split is weighted by the improved performance [28]. Here, the *GB regressor* method is performed 100 times with all the training datasets and the average importance value is shown with the minimum and maximum values in error bars. The total number of multiplication operation ($lr \times lc \times rc$) term is the dominant feature (0.246). Features that represent the output matrix size ($lr \times rc$) and the shuffle overheads ($lr \times lc + lc \times rc$) are also dominant factors, with values of 0.195 and 0.113, respectively. This observation corresponds to Figure 4b, which shows that latency may differ for MM tasks with the same number of multiplication operations ($lr \times lc \times rc$) but with different shapes and output sizes, e.g., long-thin $\times$ short-wide and short-wide $\times$ long-thin tasks. The next dominant feature is the *linpack* which measures the CPU performance of cloud instances with a value of 0.08, and this implies that the CPU performance is the most important feature in deciding the latency of distributed MM tasks with different instance types. The next important hardware feature is the *network* with a value of 0.06 that directly impacts the shuffle performance for a distributed MM task. To quantitatively analyze the impact of each feature on model accuracy, we show the coefficient of determination value along the secondary vertical axis. Starting with the most important feature, we cumulatively add the next most important feature (in the x-axis) and construct a model. The best model accuracy is achieved only with the four most important features: the number of multiplication operations, the output matrix size, the amount of shuffle overheads, and the *linpack* CPU feature. We can observe that the model accuracy spikes when adding the *linpack* feature on top of three dominant MM dimension features, and they represent MM scenarios and hardware features across different instance types well.

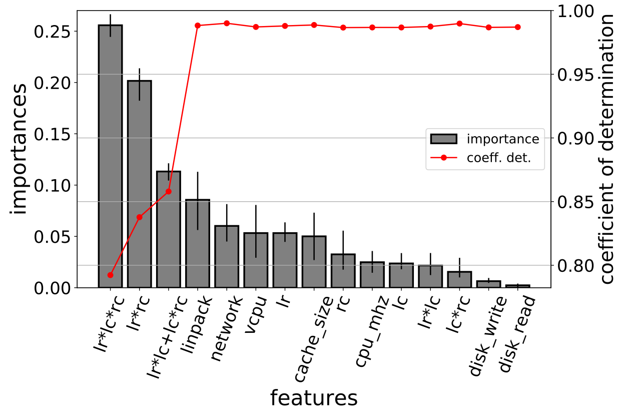
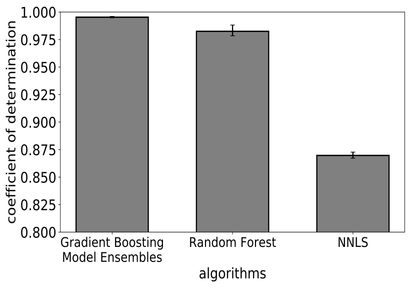


Fig. 11: Relative importance of features and the impact on model accuracy. The number of multiply operations, output matrix size, the shuffle overhead, and linpack are the most important features.

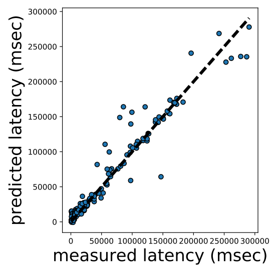
6.2.2 Comparing Multiple Predictive Algorithms

In this section, we present the efficiency of using a GB regressor to predict the execution time of MM tasks. For comparison, experiment results from a variant of a decision tree regression algorithm and a linear regression variant method are presented. For the decision tree variant, we compare *GB regressor* and random forest [22]. Unlike GB, the random forest regressor builds multiple regressor trees by randomly selecting features and samples. The GB and random forest regressors represent the non-linear characteristics of the input data while preventing overfitting by combining outcomes from many weak learners. For the linear regression variant, we adopt non-negative least squares (NNLS) regressor. When predicting latency, a non-negativity constraint is imposed to avoid latency being less than zero. The NNLS regressor finds the optimal linear model that minimizes the prediction error using this constraint.

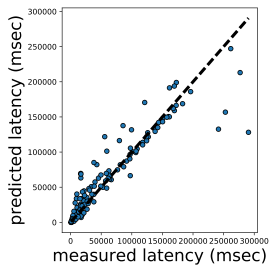
Figure 12a shows the prediction accuracy of the three algorithms. For each algorithm, 100 experiments are performed with 10-fold cross validation by randomly selecting exclusive test datasets from the training datasets (no overlap). The average R^2 value (higher R^2 values are better) is plotted in Figure 12a, with the minimum and maximum values in error bars. The *ensemble model* using *GB regressor* demonstrates the best accuracy with an R^2 value of 0.996, followed by a random forest regressor and NNLS, with R^2 values of 0.988 and 0.872, respectively. Figure 12a also shows that the linear equation cannot capture the non-linear interactions among the features proposed herein. Figures 12b (*GB regressor*) and 12c (NNLS) show the measured and predicted latency in the x and y-axes, respectively. Here, the dotted line indicates the prediction with no error (slope of one) and the scattered points close to the line represent accurate predictions. From the figures, we confirm that the *GB regressor* provides better latency prediction accuracy.



(a) latency prediction accuracy



(b) gradient boosting regressor



(c) linear regressor

Fig. 12: Prediction accuracy of various algorithms

	R^2	MAPE	RMSE
model_1	0.984	10.77%	4338
model_2	0.99	11.99%	3310
model_3	0.994	9.52%	2657
ensemble	0.995	8.7%	2204

TABLE 2: Model Ensemble and the improved performance

6.2.3 *Effectiveness of Model Ensemble*

In order to increase predictor model accuracy, we built a model of an ensemble of three GB regressors. The first model has 5,000 estimators with the least squares loss function. The second model has 10,000 estimators with the absolute deviation loss function. The third model has 15,000 estimators and a combination of two prior loss functions. Table 2 compares the performance of three distinct GB regressor model and its ensemble. Compared to a single *GB regressor* model, the ensemble model improves all the accuracy metrics resulting in 2% less MAPE and about as half RMSE. The ensemble model improves generality by combining multiple models that may be over-fitted, and this results in better performance in the prediction of various MM tasks latency.

6.2.4 *Evaluation with Diverse Cloud Computing Instance Types*

To present the applicability of the proposed MPEC algorithm on various cloud computing instance types, we built a model with training datasets from MM task scenarios that involve five instance types (*M4.4xlarge, C4.8xlarge, R4.2xlarge, I3.2xlarge*, and *D2.2xlarge*). Figure 13 shows the prediction accuracy of the MPEC model. In each evaluation metric, we show the test results across all instance types (*all*) followed by test results of each instance type. In the model accuracy evaluation, 10-fold cross-validation is performed 100 times by exclusively splitting training datasets. The value in the vertical axis is the normalized one based on the value across all instance types (*all*). Regardless of instance types, it shows decent prediction accuracy; the R^2 accuracy metric is greater than 0.98 and the MAPE is lower than 10%.

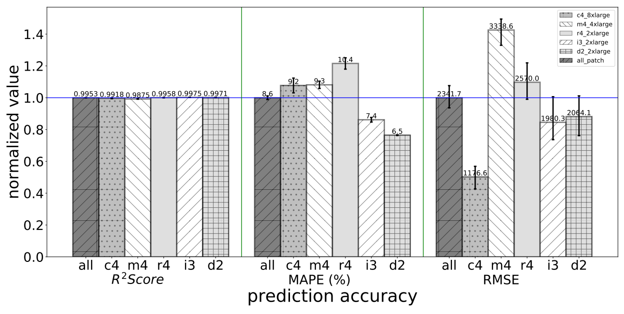
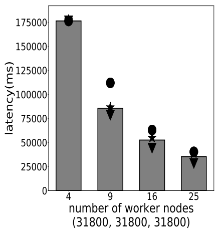
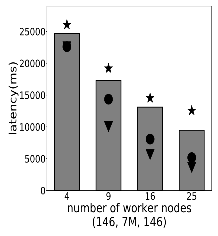


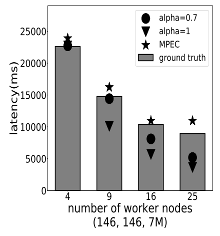
Fig. 13: Accuracy of a MPEC model built with different EC2 instance types



(a) square-square MM



(b) left columns large



(c) right column large

Fig. 14: The predicted latency of various algorithms with a fixed matrix size by varying a number of worker nodes

6.3 Evaluation of the Predictor on a Scale-out Environment

In this section, we evaluate the accuracy of the predictor on a scale-out environment as we vary the number of worker nodes. The predictor uses an MM performance estimator model built with training datasets generated from four Spark worker nodes. Figure 14 shows the predicted and true latency from the module. Here, each graph shows the latency of different MM task scenarios, i.e., (LR, LC, RC) of $(31800, 31800, 31800)$, $(146, 7000000, 146)$, and $(146, 146, 7000000)$. With the fixed matrix size, we vary the number of worker nodes (4, 9, 16, and 25), as shown on the horizontal axis. We include a linear scaler with a penalty term for comparison. The linear scaler references the latency predicted by the MPEC MM performance estimator constructed with four worker nodes, i.e., T_4, and estimates the latency by using Equation 6. Here, the α, which represents scaling overheads, ranged from 0.0 - 1.0.

$$T_{W_{prd}} = T_4 \times \frac{4}{W_{prd} \times \alpha} \tag{6}$$

As shown in Figure 14, MPEC's scale-out predictor provides consistently good prediction accuracy. Here, the star indicates the latency predicted using MPEC, and the bar indicates the true latency. Circles and triangles show the expected latency from a linear scaler predictor with an α value of 0.7 and 1.0, respectively. The linear scaler predictor sometimes predicts well; however, we could not find a linear

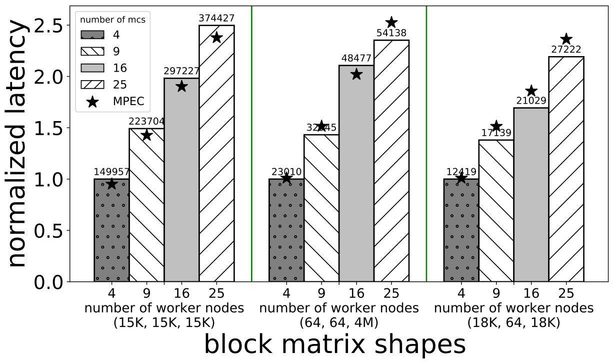


Fig. 15: Latency prediction as the number of worker nodes increases with a fixed block size

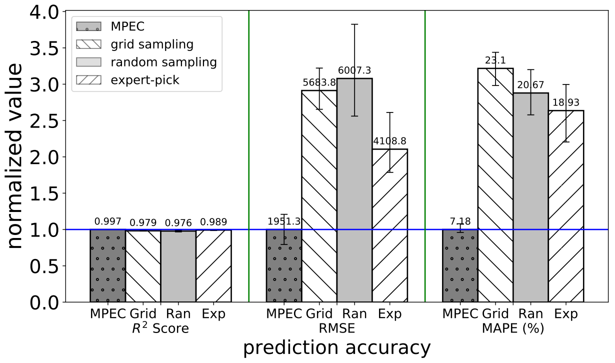
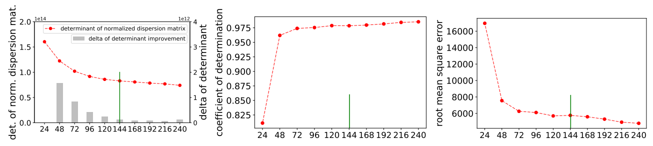


Fig. 16: Comparison of MPEC train dataset generation algorithm to Grid sampling, Random sampling, and Expert-pick methods



(a) train dataset information (b) R^2 (c) RMSE

Fig. 17: Effectiveness of D-optimal algorithm to reduce overhead when generating training datasets

scaler model that works well across all the test cases, e.g., $\alpha = 0.7$ shows better performance for $(146, 7000000, 146)$, and $(146, 146, 7000000)$, while $\alpha = 1.0$ shows better performance for $(31800, 31800, 31800)$.

Figure 15 shows the predicted MM task latency with different numbers of worker nodes. Different from experiments from Figure 14, here, we fix the block size and increase the number of blocks according to the number of workers. Thus, the total size of input matrices become larger as the number of worker nodes increase. Each bar in the figure shows the true latency, and the left-most four bars show the latency of a square block matrix $(15000, 15000)$. The next four bars show the experimental results when the number of columns of a right matrix (4000000) was much larger than the number of rows of left and right columns of the matrix (64). The last four bars show the results when the number of rows of left and right columns the matrix (18000) are larger than the number of columns in the left matrix (64). We normalized each value to the latency when the number of worker nodes was four and the value predicted by MPEC is indicated with a star. The true value is shown by the bars. The proposed MPEC shown provides consistently accurate predictions. The average MAPE of the experiments (including cases not shown) was 8%.

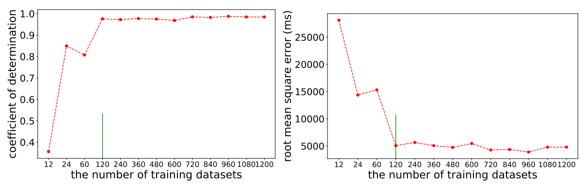
6.4 Efficiency of Train Dataset Generation

We evaluated the efficiency of an MM task scenario generation mechanism using the LHS algorithm. To quantitatively demonstrate the efficiency of applying the LHS algorithm, we compare three different methods, i.e., *Random*, *Grid*, and *Expert-pick*. Here, the *Random* and *Grid* algorithms generate 240 test datasets comprising sets of triple rational numbers ranging from 0.0 to 1.0 that are multiplied by representative MM task scenarios ((H-L-L), (L-H-L), (L-L-H), (M-L-M), (M-M-L), (L-M-M), and square × square MM). The *Random* method generates random triple values, and the *Grid* method divides the LR and RC into three regions (0.25, 0.5, and 0.75) and LC into four regions (0.125, 0.375, 0.625, and 0.875), and then permutes the values, generating 240 test datasets. In the *Expert-pick* method, a linear algebra expert from the authors' institution suggested 236 cases of MM task scenarios that are executable in a Spark environment. The expert recommended a diverse range of scenarios. To reproduce the outcomes of the experiments, the experiment scenarios of Random, Grid, Expert-pick, the proposed method in this paper, and the test datasets have been published online[6].

We use the differently generated training datasets from a Spark cluster of four AWS EC2 *R4.2xlarge* instances to build a model using the GB regressor. The prediction accuracy of distinct training datasets is shown in Figure 16. The vertical axis shows the normalized value to that of the proposed method which shows the best accuracy across all metrics. The four left-most bars show the R^2 accuracy of the proposed method, random, grid, and expert-pick methods (higher values are better). The proposed method shows slightly better values than the other methods. The next four bars show the RMSE and MAPE values (lower values are better). Similar to R^2, the proposed method demonstrates the least errors compared to other heuristics. The experimental results show that using a heuristic to generate complex MM task experiment scenarios improves the prediction accuracy without complex model tuning.

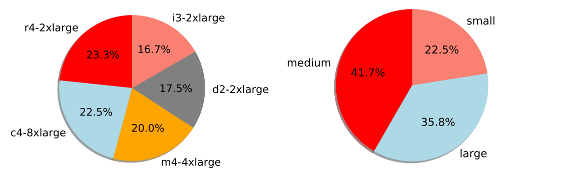
Figure 17 shows the effectiveness of applying the D-optimal algorithm relative to reducing the overheads of running many experiments without sacrificing model accuracy. Recall that implementing the Fedorov D-optimal algorithm may suggest different sets of experimental cases; thus, we repeat the tests 100 times and show the average value. This experiment is conducted with training datasets generated only from *R4.2xlarge* instance type. Figure 17a shows the amount of information changes (primary vertical axis) as we add more experimental cases (horizontal axis) in the solid line. The information value is measured with

6. http://bit.ly/2IRcLeN



(a) R^2 (b) RMSE

Fig. 18: Effectiveness of D-optimal algorithm in all instances



(a) ec2-type (b) the total number of multiply operations

Fig. 19: Distribution of D-optimal algorithm

the metric of the determinant of the normalized dispersion matrix which directly impacts the variance error term of a model. As we continue to add more training datasets, the amount of variance error decreases. To demonstrate the pattern of information improvement as more training datasets are added, we show the delta of the improvement in the gray bar (the value is given in the secondary vertical axis). Overall, the delta value is greater when the number of training datasets is smaller (left part of the x-axis), and the delta becomes smaller as more training datasets (right part of the x-axis) are added. It can be explained qualitatively that, as we expand the training datasets, it becomes more likely that similar data points will be added. Thus, the information gained is not as much as we would expect when the number of training datasets is small. The experimental results indicate that the proposed algorithm should terminate the experiment after obtaining 144 training datasets out of a total of 240 complete datasets. To measure model quality, we show the R^2 and RMSE values with different numbers of training datasets in Figures 17b and 17c, respectively. We omit MAPE as it shows a similar pattern with RMSE. As we add more training datasets, the overall accuracy increases. However, the amount of improvement is reduced beyond 144 training datasets (a vertical solid line).

6.5 Effective Experiment Scenario Generation for Diverse Instance Types

In Section 6.4, we presented the effectiveness of the proposed DoE algorithm of MPEC when selecting various MM task scenarios from a single instance type. We further evaluated the effectiveness of the proposed algorithm with various instance types. Figure 18 shows the R^2 (Figure 18a) and RMSE value (Figure 18b) when applying the DoE algorithm when selecting an optimal experimental scenario from $1,200$ cases generated from five different instance types. Interestingly, the maximum accuracy reached only 120 experiment cases out of the $1,200$. Compared to a single instance type case (Figure 17) that showed the maximum accuracy with a similar number of experiments, the intelligent selection of experimental cases significantly improved the efficiency of the system by incurring comparable overheads with a single instance type predictor while providing accurate prediction for multiple instance types. To further evaluate how the proposed DoE algorithm selects optimal 120 experiment cases, we show the distribution of EC2 instance types (Figure 19a) and MM task sizes (Figure 19b) in Figure 19a. We could see that different instance types were selected quite evenly. To determine matrix size, we divide MM tasks based on the number of multiplication operations and grouped them into three categories. Deeper analysis reveals that different MM task scenarios are covered in the different instance types, and it could compensate missing MM task scenario latency from other related experimental result; for example, a missing MM task scenario can be inferred from the results of other instance types.

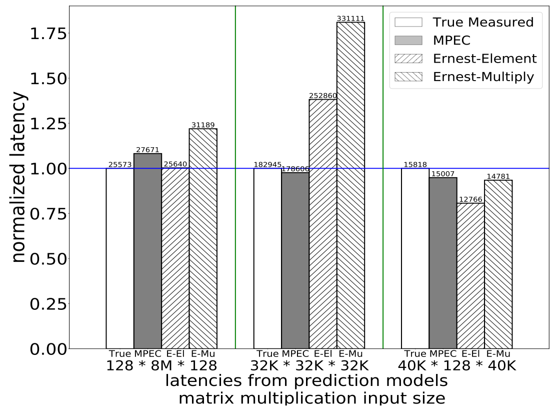


Fig. 20: Comparison of the proposed method with Ernest

6.6 Comparing MPEC to State of the Art System

To show the efficiency of the proposed system, we compare MPEC with Ernest [7], a recently created performance prediction system for general machine learning algorithms. It is composed of the experiment design and performance prediction components. In the experimental design step, Ernest constructs test cases using a small fraction of a sampled dataset and a distinct number of machines to run the experiments. In the prediction step, Ernest uses a non-negative linear regressor with training datasets gathered from the experiments. Please note that the native Ernest algorithm does not support a sampling scenario for MM tasks, and we suggest two sampling mechanisms based on the number of elements and the multiplication overheads. The prior approach considers the total number of elements in the left and right matrices of size $LR \times LC$ and $RC \times LC$ as a target metric to scale. The multiplication overhead sampling mechanism scales the total number of multiplication operation of an MM task ($LR \times LC \times RC$). In both approaches, we multiply the fraction suggested by Ernest to the target metric (total number of elements or multiplication operations).

Figure 20 shows the comparison between Ernest and the proposed MM performance prediction in the case where the number of Spark worker nodes is same for both training and prediction (four workers). On the horizontal axis, we show different matrix shapes. On the vertical axis, we show the true measured latency, the predicted latency of MPEC, the Ernest-Element, and Ernest-Multiply. Regardless of the sampling mechanism, Ernest prediction models show poor accuracy compared to the proposed method, except for the (32000,32000,32000) workload. On average, the prediction accuracy (with the metric of RMSE) of the proposed method is better than that of the Ernest-Element and Ernest-Multiply by 91% and 96%, respectively.

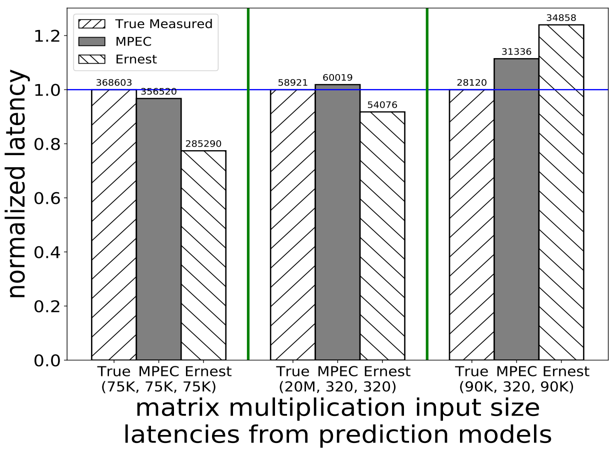


Fig. 21: Comparing MPEC with Ernest with various MM task sizes

In the next experiment, we compare Ernest with a scale-out performance predictor module where the number of Spark worker nodes is different for training (four workers) and prediction (25 workers). We select various MM task scenarios whose matrix size is available for execution with 25 *R4.2xlarge* EC2 instances. The selected MM task scenarios are $(75000, 75000, 75000)$, $(20000000, 320, 320)$, and $(90000, 320, 90000)$. In the experimental design step of Ernest, we used up to nine machines and the fraction of the datasets suggested in the literature [29] for each scenario. For the Ernest sampling mechanism, we use a multiplication-based mechanism as it shows better performance.

Figure 21 shows the true latency (left upward diagonal line), the latency predicted by MPEC (gray bar with no pattern) and Ernest (right upward diagonal line). Here, the latency value is shown at the top of each bar in milliseconds. As can be seen, the proposed MPEC outperforms Ernest regardless of the MM task scenarios. Although $(90000, 320, 900000)$ shows a slightly higher error rate, the MAPE of the proposed MPEC is less than 11%, while that of Ernest is approximately 20%. On average, the prediction accuracy (RMSE) of MPEC is better than that of Ernest by 190% for all test cases. In addition to the poorer prediction accuracy of Ernest, with respect to reusing the experimental results of MM tasks, Ernest is limited because its experiment design involves scaling down the (LR, LC, RC) precisely by the fraction and number of machines, while the proposed MPEC approach can reuse experimental results to predict the latency of arbitrary shapes and sizes of MM tasks. Furthermore, the Ernest does not support performance prediction across different instance types, and a same set of experiments should be conducted for different instance types.

7 Related Work

Big data workload performance estimation on cloud: To analyze large-scale datasets, a MapReduce programming model [30] is widely used to express complex data-oriented operations. Hadoop [1] and Spark [2] are popular platforms that implement the MapReduce programming model. Deploying such big-data analytic environments via cloud computing resources is becoming the norm, and researchers are investigating how to provide optimal cloud computing environments to run Hadoop and Spark. To find the optimal Hadoop environment on cloud, StarFish [31] finds optimal configurations for Hadoop tasks. Its core component includes profiler and what-if analysis modules that understand the characteristics of the Hadoop tasks and find a better configuration. The StartFish algorithm is specific to an instance type, and it is not generally applicable to other various instance types as our proposed work does. Bazaar [32] and Conductor [33] find optimal instance types and pricing options including spot instances [34] by applying input sampling for Hadoop. Though they present good accuracy in the prediction, the applications they use in the evaluation are fairly simple, such as wordcount, sorting, tf-idf. As presented in Section 6.6, capturing the complex characteristics of MM tasks using a sampling method is very challenging. Different from other approaches, MRPerf [35] is a simulator for Hadoop frameworks that focuses mainly on data locality and network topology simulations. Simulations are beneficial as they reduce offline experimental overheads significantly, but the model does not support simulation of various cloud computing resource features.

To address challenges in cloud computing resource selection when using Spark, PARIS [9] employs offline profiling and online prediction. In offline profiling, PARIS runs target applications with scale-based sample datasets while measuring hardware provided metrics, such as CPU and memory utilization. Based on these usage metrics, PARIS applies the Random Forest (RF) algorithm to make a prediction. Cherrypick [8] uses Bayesian optimization to select the next instance type to run experiments and suggest the most appropriate instance type among the tested instances. Ernest [7] applies the NNLS equation to model interactions among different hardware configurations with profiled experiment results from scale-based sample datasets. Such approaches rely on the scale-based sampling approach to reduce the overheads incurred by the entire datasets. However, the approach reruns tests for different datasets and the experimental results from previous runs are generally not reusable. In addition, these previous studies focused on predicting the response time to execute high-level machine learning algorithms; thus, they could not capture the complex characteristics of MM tasks. In the optimal cloud environment recommendation, CAST [36] and Selecta [37] focused on various cloud storage service options. They provide good accuracy in the prediction, but the target applications in the evaluation are limited to simple ones, such as sort, join, grep, and SQL workloads.

FiM [38] and Mariani et al. [39] employ an algorithm to predict the latency of running HPC applications using machine learning algorithms. They also rely on the profiling of HPC applications using a scale-based sampling method to gather various metrics for different cloud computing resources, and it applies the RF algorithm to discover the correlation between HPC applications and cloud resources. Differing from such previous studies, the proposed method uses unit blocks to predict MM task performance by synthetically creating an MM task where the offline profiling experiments are performed, and it enables the utilization of previous experimental results of any kind to improve model accuracy.

Vicent Sanz et al. [40] presented a mixtures-of-experts approach to model the memory behavior of Spark applications. This system utilizes several distinct models that are constructed offline, and an expert selector using a K-Nearest Neighbor classifier determines which model should be invoked based on the run-time information (CPU and memory) of an application. Although it covers a wide range of Spark application domains, its training cost can be prohibitive for MM tasks that cover a broad range of scenarios.

Differing from previously mentioned works, Dione [41] attempted to predict the execution time of various data mining jobs by referencing an execution DAG graph provided by Apache Spark. After grouping similar DAG execution graphs, the authors could predict the execution time of unseen tasks with minimal training overheads. Note that they did not consider the scale-out environments, and the proposed algorithm is not directly applicable if the number of worker nodes changes. As presented in this paper, different applications can have different scaling patterns, which may result in poor prediction accuracy.

Optimizing distributed matrix multiplication on cloud: MM is an essential task in machine learning jobs with large-scale datasets. Due to the importance of the tasks and the ever-increasing sizes of datasets, many studies have focused on optimizing the task in a distributed cloud computing environment. Yu et al. [42] thoroughly investigates the communication overheads of various distributed MM shapes and proposes a task execution plan to minimize the communication cost. Marlin [43] proposes a distributed MM algorithm on Spark to minimize the shuffle overheads. As discussed quantitatively in this study, shuffle overheads are crucial for determining the performance of distributed MM tasks; however, other than the shuffle overhead, the output matrix size and total number of products also impose a significant impact on the overall task completion time. The MM task optimization works are complementary to the proposed MPEC system; the optimization of MM performance is specific to an instance type, while finding the optimal instance type for a given MM workload is across many instance types, and the algorithms can be improved independently.

Methods of choosing optimal experiment scenarios: In the field of system performance prediction on cloud resources, offline profiling experiments are conducted to generate the inputs for the training model. Because the generated training dataset can be prohibitive, several previous studies adopted DoE [13] to determine optimal scenarios for offline experiments. Mariani et al. [39] use the central composite DoE algorithm to determine an application's parameters by minimizing the uncertainty of a nonlinear polynomial model considering the interactions among parameters. Ernest [7] uses the A-optimal DoE algorithm to maximize the traces of an information matrix to select the number of machines and a fraction of input datasets to test in the offline profiling step. Packing Light [44] employs a response surface design that helps to understand and optimize how the query workload responds to changing hardware configurations. Packing Light also employs LHS to take random samples. The algorithm proposed in this paper is unique because it uses the LHS algorithm to generate a comprehensive set of MM task scenarios and it uses the D-optimal algorithm to select a subset of optimal test cases to reduce the cost of offline experimental tests.

8 Conclusion and Future Work

With the rapid development of cloud computing services, big-data application developers have many options when deploying execution engines. To build a cost-efficient big-data analytics platform using various cloud computing instance types, we have proposed MPEC to accurately estimate the latency of MM tasks of various shapes and sizes, which is a core computational kernel of many machine learning algorithms. We first characterize the overheads of distributed MM and propose 16 features that represent various MM task scenarios and diverse hardware specifications. We then leverage a reusable algorithm to predict the latency of MM tasks of various shapes and sizes with different numbers of worker nodes and instance types. For optimal generation of training datasets, we propose an intelligent way to design experimental scenarios to reduce overheads incurred by running multiple experiments to generate training datasets. Thorough experimental results reveal important features that determine the overall latency of distributed MM tasks, and the proposed MM latency estimation algorithm on scale-out environments consistently provide accurate prediction. Furthermore, the proposed MPEC outperforms a state-of-the-art machine learning performance predictor, Ernest, with 190% less prediction error (relative to RMSE).

The proposed MPEC system has rooms for improvement. The current version assumes that a cluster is composed homogeneously with the same instance type, but a cluster can be configured heterogeneously for efficiency. No prior works had yet investigated the performance estimation and optimization of heterogeneous Spark environment on the cloud. We mainly focused on the prediction of block-based matrix partitioning because it shows the best performance and is generally applicable. Our investigation of the prediction of other matrix partitioning mechanisms is on-going. On cloud, there are many pricing options that users can choose. Among the price options, using opportunistic resources at cheaper price [34], [45] for distributed MM tasks and machine learning jobs can result in different observation.

ACKNOWLEDGEMENTS

This work is supported by the ICT R&D program of IITP (2017-0-00396), the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) (No. NRF-2015R1A5A7037615 and NRF-2016R1C1B2015135), and the AWS Cloud Credits for Research program.

REFERENCES

[1] A. S. Foundation, Apache hadoop (2004). URL http://hadoop.apache.org/

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, 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), USENIX, San Jose, CA, 2012, pp. 15–28.

[3] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Networks 61 (2015) 85 – 117. doi:http://dx.doi.org/10.1016/j.neunet.2014.09.003. URL http://www.sciencedirect.com/science/article/pii/S0893608014002135

[4] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (5) (1989) 359–366. doi:10.1016/0893-6080(89)90020-8. URL http://dx.doi.org/10.1016/0893-6080(89)90020-8

[5] D. D. Lee, H. S. Seung, Algorithms for non-negative matrix factorization, in: In NIPS, MIT Press, 2000, pp. 556–562.

[6] G. H. Golub, C. Reinsch, Singular value decomposition and least squares solutions, Numer. Math. 14 (5) (1970) 403–420. doi:10.1007/BF02163027. URL http://dx.doi.org/10.1007/BF02163027

[7] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, I. Stoica, Ernest: Efficient performance prediction for large-scale advanced analytics., in: NSDI, 2016, pp. 363–378.

[8] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, 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), USENIX Association, Boston, MA, 2017, pp. 469–482. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[9] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, 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, SoCC '17, ACM, New York, NY, USA, 2017, pp. 452–465. doi:10.1145/3127479.3131614. URL http://doi.acm.org/10.1145/3127479.3131614

[10] Z. Xianyi, W. Qian, Z. Chothia, Openblas, URL: http://xianyi.github. io/OpenBLAS.

[11] J. H. Friedman, Greedy function approximation: A gradient boosting machine., Ann. Statist. 29 (5) (2001) 1189–1232. doi:10.1214/aos/1013203451. URL https://doi.org/10.1214/aos/1013203451

[12] A. Olsson, G. Sandberg, O. Dahlblom, On latin hypercube sampling for structural reliability analysis 25 (1) (2003) 47–68. URL http://dx.doi.org/10.1016/S0167-4730(02)00039-5

[13] D. C. Montgomery, Design and Analysis of Experiments, John Wiley & Sons, 2006.

[14] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, M. Zaharia, Matrix computations and optimization in apache spark, KDD '16, ACM, 2016, pp. 31–38. doi:10.1145/2939672.2939675.

[15] F. Liu, K. Lee, I. Roy, V. Talwar, S. Chen, J. Chang, Gpu accelerated array queries : The good , the bad , and the promising, in: HP Tech. Report., HP Labs., 2014.

[16] R. A. van de Geijn, J. Watts, Summa: Scalable universal matrix multiplication algorithm, Tech. rep., Austin, TX, USA (1995).

[17] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, O. Spillinger, Communication-optimal parallel recursive rectangular matrix multiplication, in: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 261–272. doi:10.1109/IPDPS.2013.80. URL http://dx.doi.org/10.1109/IPDPS.2013.80

[18] A. Spark, Apache spark mllib distributed matrix computation, https://goo.gl/Vnii2M, [Online; accessed 20-Nov-2017] (2017).

[19] L. Breiman, Bagging predictors, Machine Learning 24 (2) (1996) 123–140. doi:10.1007/BF00058655. URL https://doi.org/10.1007/BF00058655

[20] J. Kim, K. Lee, Functionbench: A suite of workloads for serverless cloud function service, in: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019, pp. 502–504. doi:10.1109/CLOUD.2019.00091.

[21] J. J. Dongarra, P. Luszczek, A. Petitet, The linpack benchmark: past, present and future, Concurrency and Computation: Practice and Experience 15 (9) (2003) 803–820. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728, doi:10.1002/cpe.728. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728

[22] L. Breiman, Random forests, Machine Learning 45 (1) (2001) 5–32. doi:10.1023/A:1010933404324.

[23] M. Son, 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, 2018, pp. 638–645. doi:10.1109/CLOUD.2018.00088. URL doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00088

[24] A. Ly, M. Marsman, J. Verhagen, R. P. Grasman, E.-J. Wagenmakers, A tutorial on fisher information, Journal of Mathematical Psychology 80 (2017) 40 – 55. doi:https://doi.org/10.1016/j.jmp.2017.05.006. URL http://www.sciencedirect.com/science/article/pii/S0022249617301396

[25] F. Triefenbach, Design of experiments: The d-optimal approach and its implementation as a computer algorithm.

[26] K. Ogungbenro, G. Graham, I. Gueorguieva, L. Aarons, The use of a modified fedorov exchange algorithm to optimise sampling times for population pharmacokinetic experiments, Computer Methods and Programs in Biomedicine 80 (2) (2005) 115 – 125. doi:https://doi.org/10.1016/j.cmpb.2005.07.001. URL http://www.sciencedirect.com/science/article/pii/S016926070500146X

[27] B. Wheeler, AlgDesign: Algorithmic Experimental Design, r package version 1.1-7.3 (2014). URL https://CRAN.R-project.org/package=AlgDesign

[28] J. Elith, J. R. Leathwick, T. Hastie, A working guide to boosted regression trees, Journal of Animal Ecology 77 (4) (2008) 802–813. doi:10.1111/j.1365-2656.2008.01390.x.

[29] A. Lab, amplab/ernest, https://github.com/amplab/ernest, [Online; accessed 18-May-2018] (2018).

[30] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI'04, USENIX Association, Berkeley, CA, USA, 2004, pp. 10–10. URL http://dl.acm.org/citation.cfm?id=1251254.1251264

[31] H. Herodotou, S. Babu, Profiling, what-if analysis, and cost-based optimization of mapreduce programs., PVLDB 4 (11) (2011) 1111–1122. URL http://dblp.uni-trier.de/db/journals/pvldb/pvldb4.html#HerodotouB11

[32] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, A. Rowstron, Bridging the tenant-provider gap in cloud services, in: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12, ACM, New York, NY, USA, 2012, pp. 10:1–10:14. doi:10.1145/2391229.2391239. URL http://doi.acm.org/10.1145/2391229.2391239

[33] A. Wieder, P. Bhatotia, A. Post, 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), USENIX, San Jose, CA, 2012, pp. 367–381. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/wieder

[34] S. Alkharif, K. Lee, H. Kim, Time-series analysis for price prediction of opportunistic cloud computing resources, in: W. Lee, W. Choi, S. Jung, M. Song (Eds.), Proceedings of the 7th International Conference on Emerging Databases, Springer Singapore, Singapore, 2018, pp. 221–229.

[35] G. Wang, A. R. Butt, P. Pandey, K. Gupta, A simulation approach to evaluating design decisions in mapreduce setups, in: 2009 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009, pp. 1–11. doi:10.1109/MASCOT.2009.5366973.

[36] Y. Cheng, M. S. Iqbal, A. Gupta, 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, HPDC '15, ACM, New York, NY, USA, 2015, pp. 45–56. doi:10.1145/2749246.2749252.
URL http://doi.acm.org/10.1145/2749246.2749252

[37] A. Klimovic, H. Litz, C. Kozyrakis, Selecta: Heterogeneous cloud storage configuration for data analytics, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), USENIX Association, Boston, MA, 2018, pp. 759–773.
URL https://www.usenix.org/conference/atc18/presentation/klimovic-selecta

[38] J. Bhimani, N. Mi, M. Leeser, Z. Yang, Fim: Performance prediction for parallel computation in iterative data processing applications, in: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), 2017, pp. 359–366. doi:10.1109/CLOUD.2017.53.

[39] G. Mariani, A. Anghel, R. Jongerius, G. Dittmann, Predicting cloud performance for hpc applications: A user-oriented approach, in: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid '17, IEEE Press, Piscataway, NJ, USA, 2017, pp. 524–533. doi:10.1109/CCGRID.2017.11.
URL https://doi.org/10.1109/CCGRID.2017.11

[40] V. S. Marco, B. Taylor, B. Porter, Z. Wang, Improving spark application throughput via memory aware task co-location: A mixture of experts approach, Middleware '17, ACM, Las Vegas, NV, USA, 2017, pp. 95–108. doi:10.1145/3135974.3135984.
URL http://arxiv.org/abs/1710.00610

[41] N. Zacheilas, S. Maroulis, V. Kalogeraki, Dione: Profiling spark applications exploiting graph similarity, in: 2017 IEEE International Conference on Big Data (Big Data), 2017, pp. 389–394. doi:10.1109/BigData.2017.8257950.

[42] Y. Yu, M. Tang, W. G. Aref, Q. M. Malluhi, M. M. Abbas, M. Ouzzani, In-memory distributed matrix computation processing and optimization, in: ICDE, 2017, pp. 1047–1058. doi:10.1109/ICDE.2017.150.

[43] R. Gu, Y. Tang, Z. Wang, S. Wang, X. Yin, C. Yuan, Y. Huang, Efficient large scale distributed matrix computation with spark, in: 2015 IEEE International Conference on Big Data (Big Data), 2015, pp. 2327–2336. doi:10.1109/BigData.2015.7364023.

[44] J. Duggan, Y. Chi, H. Hacigumus, S. Zhu, U. Cetintemel, Packing light: Portable workload performance prediction for the cloud, in: Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on, IEEE, 2013, pp. 258–265.

[45] V. Khandelwal, A. Chaturvedi, C. P. Gupta, Amazon ec2 spot price prediction using regression random forests, IEEE Transactions on Cloud Computing (2017) 1–1doi:10.1109/TCC.2017.2780159.

Jeongchul Kim is an M.S. student in the Department of Computer Science at Kookmin University, South Korea. His research interests include distributed computing and cloud computing. He is currently working on performance analysis in a serverless computing system. He received a bachelor degree in the department of Computer Science at Kookmin University.

Myungjun Son is a Ph.D student in the department of Computer Science and Engineering of Penn State University, University Park, USA. He received the M.S. degree in the Department of Computer Science at Kookmin University, South Korea. His research involves designing systems and algorithms for large-scale data analysis and machine learning. He received a bachelor degree in the department of Computer Science at Kookmin University.

Kyungyong Lee is an assistant professor in the College of Computer Science at Kookmin University. His current research topic covers big data platforms, large-scale distributed computing resource management, cloud computing, and peer-to-peer systems. He received the Ph. D. degree in the Department of Electrical and Computer Engineering at the University of Florida.