SystemML: Declarative Machine Learning on MapReduce

Amol Ghoting [#], Rajasekar Krishnamurthy [*], Edwin Pednault [#], Berthold Reinwald [*]
Vikas Sindhwani [#], Shirish Tatikonda [*], Yuanyuan Tian [*], Shivakumar Vaithyanathan [*]

[#] *IBM Watson Research Center* [*] *IBM Almaden Research Center*

{aghoting, rajase, pednault, reinwald, vsindhw, statiko, ytian, vaithyan}@us.ibm.com

***Abstract*—MapReduce is emerging as a generic parallel programming paradigm for large clusters of machines. This trend combined with the growing need to run machine learning (ML) algorithms on massive datasets has led to an increased interest in implementing ML algorithms on MapReduce. However, the cost of implementing a large class of ML algorithms as low-level MapReduce jobs on varying data and machine cluster sizes can be prohibitive. In this paper, we propose SystemML in which ML algorithms are expressed in a higher-level language and are compiled and executed in a MapReduce environment. This higher-level language exposes several constructs including linear algebra primitives that constitute key building blocks for a broad class of supervised and unsupervised ML algorithms. The algorithms expressed in SystemML are compiled and optimized into a set of MapReduce jobs that can run on a cluster of machines. We describe and empirically evaluate a number of optimization strategies for efficiently executing these algorithms on Hadoop, an open-source MapReduce implementation. We report an extensive performance evaluation on three ML algorithms on varying data and cluster sizes.**

I. Introduction

Recently, there has been a growing need for scalable implementations of machine learning (ML) algorithms on very large datasets (ranging from 100s of GBs to TBs of data [1]). This requirement is driven by applications such as social media analytics, web-search, computational advertising and recommender systems. Previous attempts at building scalable machine learning algorithms have largely been hand-tuned implementations on specialized hardware/parallel architectures [1], or as noted in [2], clever methods to parallelize individual learning algorithms on a cluster of machines [3], [4], [5]. The recent popularity of MapReduce [6] as a generic parallel programming model has invoked significant interest in implementing scalable versions of ML algorithms on MapReduce. These algorithms have been implemented over multiple MapReduce architectures [7], [8], [9] ranging from multicores [2] to proprietary [10], [11], [12] and open source implementations [13].

Much of this work reverts back to hand-tuned implementations of specific algorithms on MapReduce [10], [11]. One notable exception is [2] where the authors abstract one common operation – "summation form" – and present a recipe to map instances of this operation onto MapReduce[2]. Several algorithms are then expressed using multiple instances of the summation form mapped appropriately to MapReduce jobs. This approach still leaves two fundamental problems to be addressed:

- Each individual MapReduce job in an ML algorithm has to be hand-coded.
- For better performance, the actual execution plan for the same ML algorithm has to be hand-tuned for different input and cluster sizes.

Example 1: The practical implications of the above two fundamental drawbacks are illustrated using this example. Algorithm 1 shows a popular ML algorithm called Gaussian Non-Negative Matrix Factorization (GNMF [14]) that has applications in document clustering, topic modeling and computer vision. In the context of topic modeling, V is a $d \times w$ matrix with d documents and w words. Each cell of V represents the frequency of a word appearing in a document. GNMF tries to find the model of t topics encoded in W ($d \times t$) and H ($t \times w$) matrices, such that $V \approx WH$. As seen in the algorithm[3], this is an iterative algorithm consisting of two major steps in a while loop, each step consisting of multiple matrix operations. X^T denotes the transpose of a matrix X, XY denotes the multiplication of two matrices X and Y, $X * Y$ and X/Y denote cell-wise multiplication and division respectively (see Table I).

Algorithm 1 Gaussian Non-Negative Matrix Factorization

```
1:  V = read("in/V"); //read input matrix V
2:  W = read("in/W"); //read initial values of W
3:  H = read("in/H"); //read initial values of H
4:  max_iteration = 20;
5:  i = 0;
6:  while i < max_iteration do
7:      H = H * (W^T V / W^T W H); //update H
8:      W = W * (V H^T / W H H^T); //update W
9:      i = i + 1;
10: end while
11: write(W,"out/W"); //write result W
12: write(H,"out/H"); //write result H
```

[1]This refers to the size of the numeric features on which the algorithm operates. The raw data from which the numeric features are extracted may be larger by 1 to 2 orders of magnitude.

[2]A class of ML algorithms compute certain global statistics which can be expressed as a summation of local statistics over individual data points. In MapReduce, local statistics can be computed by mappers and then aggregated by reducers to produce the global statistics.

[3]To simplify the exposition, we leave out straightforward expressions for objective function and convergence criteria in the algorithm description.

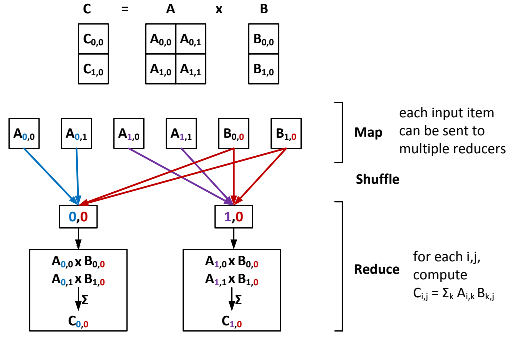


Fig. 1. RMM: Replication based Matrix Multiplication

Consider the expression WHH^T in Step 8 of Algorithm 1. This expression can be evaluated in one of two orders, $od1$: $(WH)H^T$ and $od2$: $W(HH^T)$. At first glance, picking the right order and performing this computation may seem straightforward, but the fact that matrix multiplication itself can be accomplished in multiple ways complicates matters.

Figure 1 and Figure 2 show two alternative MapReduce plans for matrix multiplication (details of the two plans will be discussed in Section IV). The RMM plan in Figure 1 implements a replication-based strategy in a single MapReduce job, while the CPMM plan in Figure 2 implements a cross-product strategy that requires 2 MapReduce jobs. The choice of RMM vs CPMM is dictated by the characteristics of the matrices involved in the multiplication. To compute WHH^T, we have to choose from a total of 8 plans: first choose the order of evaluation, $od1$ or $od2$, and for the chosen order choose from RMM or CPMM for each matrix multiplication. Instantiating the dimensionalities of the matrices reveals the need to choose one plan over another. In the context of topic modeling, the number of topics t is much smaller than the number of documents d and the number of words w. As a result, $od1$ will never be selected as the evaluation order, since WH produces a $d \times w$ large intermediate matrix whereas HH^T in $od2$ results in a $t \times t$ small matrix. When $d = 10^7$, $w = 10^5$ and $t = 10$, H is of medium size and the result of HH^T is tiny. The replication based approach RMM performs very well for both matrix multiplications. The best plan with $od2$ is to use RMM for HH^T followed by another RMM for the pre-multiplication with W. Empirically, this plan is 1.5 times faster than the second best plan of using CPMM followed by RMM. However, when w is changed to 5×10^7, size of H increases 500 times. The overhead of replicating H and H^T makes RMM inferior to CPMM for the computation of HH^T. On the other hand, the result of HH^T remains to be a tiny matrix, so the best plan to compute the pre-multiplication with W is still RMM. A cost model and a detailed discussion on choosing between CPMM and RMM will be provided in Section IV.

As shown above, the choice of a good execution strategy depends significantly on data characteristics. Pushing this burden on programmers will have serious implications in terms of scaling both development and execution time. This paper takes a step towards addressing this problem.

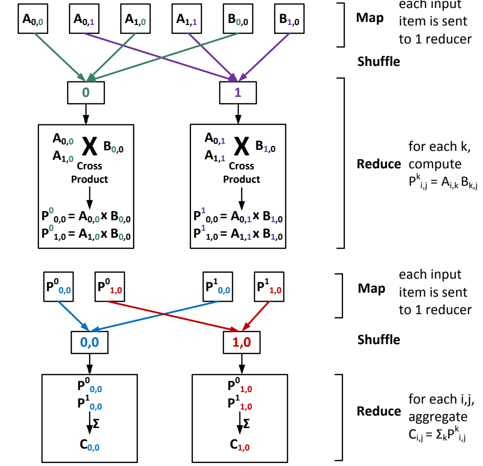


Fig. 2. CPMM: Cross Product based Matrix Multiplication

Problem Statement: Build a scalable declarative machine learning system that

- exposes a declarative higher-level language for writing ML algorithms, thereby freeing the user from low-level implementation details and performance-tuning tasks.
- provides performance that scales to very large datasets and is comparable to hand-tuned implementations of individual algorithms.
- covers a large class of ML and statistical algorithms whose computational cores are linear algebra primitives and iterative numerical optimization procedures. These include (but are not restricted to) linear statistical models, PCA, PageRank, Matrix Factorizations, and so on.

The remainder of the paper is organized as follows. In Section II, we present **SystemML**, in which ML algorithms are expressed in a higher-level language subsequently compiled and automatically parallelized to execute in Hadoop, an open source implementation of MapReduce. We then describe the individual components of SystemML in Section III. We discuss the role of cost based optimization by showing two alternative execution plans for the expensive matrix multiplication operation. We then present extensive experimental results (Section V) to demonstrate the scalability of SystemML and the effectiveness of the optimizations performed at various stages.

II. SystemML Overview

We now give an overview of SystemML. Figure 3(a) shows the overall architecture of SystemML that consists of four components.

Language: Algorithms in SystemML are written in a high-level language called **D**eclarative **M**achine learning **L**anguage (DML). DML exposes mathematical and linear algebra primitives on matrices that are natural to express a large class of ML algorithms, including linear models, PCA, PageRank, NMF etc. In addition, DML supports control constructs such as *while* and *for* to write complex iterative algorithms. Through program analysis, SystemML breaks a DML script into smaller

TABLE I
EXAMPLE OPERATORS IN DML: x_{ij}, y_{ij} AND z_{ij} ARE CELLS IN MATRICES X, Y AND Z, RESPECTIVELY.

Algorithm 1	DML Statement	Semantics	HOP Notation	LOP Notation
$Z = X * Y$	`Z=X*Y`	cell-wise multiplication: $z_{ij} = x_{ij} * y_{ij}$	$b(*) : X, Y$	$group \rightarrow binary(*)$
$Z = X/Y$	`Z=X/Y`	cell-wise division: $z_{ij} = x_{ij}/y_{ij}$	$b(/) : X, Y$	$group \rightarrow binary(/)$
$Z = XY$	`Z=X%*%Y`	matrix multiplication: $z_{ij} = \sum_k x_{ik} * y_{kj}$	$ab(\sum, *) : X, Y$	$(mmrj)$ or $(mmcj \rightarrow group \rightarrow aggregate(\sum))$
$Z = X^T$	`Z=t(X)`	transpose: $z_{ij} = x_{ji}$	$r(T) : X$	$transform(t)$
	`Z=log(X)`	cell-wise logarithm: $z_{ij} = log(x_{ij})$	$u(log) : X$	$unary(log)$
	`Z=rowSum(X)`	row-wise sums: $z_i = \sum_j x_{ij}$	$au(\sum, row) : X$	$transform(row) \rightarrow group \rightarrow aggregate(\sum)$

units called *statement blocks*. Each statement block, separately, is optimized and executed by subsequent components.

High-Level Operator Component (HOP): The HOP component analyzes all the operations within a statement block and chooses from multiple high-level execution plans. A plan is represented in a HOP-Dag, a directed acyclic graph of basic operations (called *hops*) over matrices and scalars. Optimizations considered in this component include algebraic rewrites, selection of the physical representation for intermediate matrices, and cost-based optimizations.

Low-Level Operator Component (LOP): The LOP component translates the high-level execution plans provided by the HOP component into low-level physical plans on MapReduce, represented as LOP-Dags. Each low-level operator (*lop*) in a LOP-Dag operates on key-value pairs or scalars. The LOP-Dag is then compiled into one or more MapReduce jobs by packing multiple lops into MapReduce jobs to keep the number of data scans small. We refer to this strategy as *piggybacking*.

Runtime: The runtime component executes the low-level plans obtained from the LOP component on Hadoop. The main execution engine in SystemML is a generic MapReduce job, which can be instructed to execute multiple lops inside it. A control module orchestrates the execution of different instances of the generic MapReduce job. Multiple optimizations are performed in the runtime component; e.g., execution plans for individual lops are decided dynamically based on data characteristics such as sparsity of the input matrices.

Figure 3(b) shows how a single DML statement `A=B*(C/D)` is processed in SystemML. The language expression consists of untyped variables and is translated into a HOP-Dag consisting of a cell-wise division hop and a cell-wise multiplication hop on matrices. A lower-level execution plan is then generated for this expression as shown in the LOP-Dag. Here, the *Cell-Wise Binary Divide* hop is translated into two lops – a *Group* lop that sorts key-value pairs to align the cells from C and D; followed by the lop *Binary Divide on Each Group*. Finally, the entire LOP-Dag is translated into a single MapReduce job, where (a) the mapper reads three inputs, (b) all groupings are performed implicitly between the mapper and the reducer and (c) the reducer performs the division followed by the multiplication.

III. SYSTEMML COMPONENTS

A. Declarative Machine learning Language (DML)

DML is a declarative language whose syntax closely resembles the syntax of R[4] [16]. To enable more system generated optimization, DML does not provide all the flexibility available in R. However, this loss in flexibility results largely in loss in programming convenience and does not significantly impact the class of ML algorithms that are expressible in DML. The GNMF algorithm (Algorithm 1) is expressed in DML syntax in Script 1. We explain DML constructs using this example.

Script 1: GNMF

```
1: V=readMM("in/V", rows=1e8, cols=1e5, nnzs=1e10);
2: W=readMM("in/W", rows=1e8, cols=10);
3: H=readMM("in/H", rows=10, cols=1e5);
4: max_iteration=20;
5: i=0;
6: while(i<max_iteration){
7:   H=H*(t(W)%*%V)/(t(W)%*%W%*%H);
8:   W=W*(V%*%t(H))/(W%*%H%*%t(H));
9:   i=i+1;}
10:writeMM(W, "out/W");
11:writeMM(H, "out/H");
```

Data Types: DML supports two main data types: matrices and scalars [5]. Scalar data types supported are integer, double, string and logical. The cells in a matrix may consist of integer, double, string or logical values.

Statements: A DML program consists of a sequence of statements, with the default computation semantics being sequential evaluation of the individual statements.

The following constructs are currently supported in DML.

Input/Output: *ReadMM* and *WriteMM* statements are provided for respectively reading and writing matrices from and to files. Optionally, in the *ReadMM* statement, the user can provide additional properties of the matrix such as sparsity (number of non-zero entries or nnzs).

Control Structures: Control structures supported in DML include the *while* statement, *for* statement and *if* statement. Steps 6-9 in Script 1 show an example *while* statement.

Assignment: An *assignment* statement consists of an expression and the result of which is assigned to a variable - e.g., Steps 7 ,8 and 9 in Script 1. Note that the assignment can be to a scalar or a matrix.

Table I lists several example operators allowed in expressions in DML. The arithmetic operators $+, -, *, /$ extend naturally to matrices where the semantics is such that the operator is applied to the corresponding cells. For instance, the expression $Z = X * Y$ will multiply the values in the corresponding cells in X and Y, and populate the appropriate cell in Z with the result. Several internal functions, specific to particular data types, are supported – e.g., *rowSum* computes

[4]R is prototypical for a larger class of such languages including Matlab [15]

[5]We treat vectors as a special case of matrices.

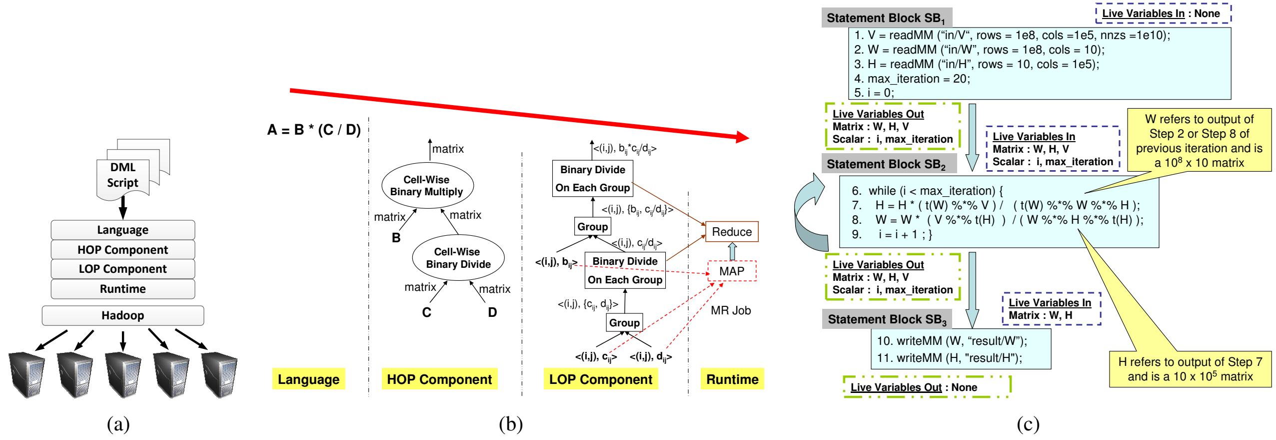


Fig. 3. (a) SystemML Architecture, (b) Evaluation of A=B*(C/D): conceptually, each key-value pair contains the index and the value of a cell in the matrix, (c) Program Analysis

the sum of every row in a matrix and returns a column matrix (i.e., a vector), while *t(·)* computes the transpose of a matrix.

DML also allows users to define their own functions using the syntax *"function (arglist) body"*. Here, the *arglist* consists of a set of formal input and output arguments and the body is a group of valid DML statements.

Comparison with R programming language: As pointed out before, we have made some choices in the design of DML to better enable system optimizations. For example, DML does not support *object oriented features*, *advanced data types* (such as lists and arrays) and *advanced function support* (such as accessing variables in the caller function and further up in the call-stack). Besides these advanced features for programming convenience, R also supports extensive *graphical procedures* that are clearly beyond the scope of DML.

Program Analysis: We now describe the sequence of steps a DML script goes through to generate a parsed representation. Figure 3(c) shows the result of program analysis for Script 1.

Type Assignment: The first step is to assign data types to each variable in the DML script. For instance, *ReadMM* statements (Steps 1-3) are used to type V, W and H as matrices, while *Assignment* statements (Steps 4-5) are used to identify $max_iteration$ and i as scalar variables.

Statement Block Identification: As control constructs (such as *while*) and *functions* break the sequential flow of a DML program, they naturally divide the program into *statement blocks*. Each statement block consists of consecutive *Assignment*, *ReadMM* and *WriteMM* statements, as the operations involved in these statements can be collectively optimized. Figure 3(c) illustrates our example algorithm broken down into three statement blocks (SB_1, SB_2 and SB_3).

Live Variable Analysis: The goal of this step is twofold: (a) Connect each variable use with the immediately preceding write(s) for that variable across different evaluation paths. For example, variable W used in Step 7 refers to the output of Step 2 for the first iteration of the loop and Step 8 for second iteration onwards. (b) For each statement block, identify the variables that will be required from previous statement blocks (*Live Variables In*) and the variables that will be output by the current statement block (*Live Variables Out*). The results of live variable analysis are shown in Figure 3(c).

B. High-Level Operator Component (HOP)

The HOP component takes the parsed representation of a statement block as input, and produces a HOP-Dag representing the data flow.

Description of hops: Each hop in the HOP-Dag has one or more input(s), performs an operation or transformation, and produces output that is consumed by one or more subsequent hops. Table II lists some example hops supported in SystemML along with their semantics[6]. In addition, the instantiation of hops from the DML parsed representation is exemplified in Table I. Consider the matrix multiplication `Z=X%*%Y` as an instance, an *AggregateBinary* hop is instantiated with the binary operation $*$ and the aggregate operation $\sum$. The semantics of this hop instance, denoted by $ab(\sum, *)$, is to compute, $\forall i, j, \sum_k (x_{i,k} * y_{k,j})$.

Construction of HOP-Dag: The computation in each statement block is represented as one HOP-Dag [7]. Figure 4(a) shows the HOP-Dag for the body of the *while* loop statement block in Figure 3(c) constructed using the hops in Table II. Note how multiple statements in a statement block have been combined into a single HOP-Dag. The HOP-Dag need not be a connected graph, as shown in Figure 4(a).

The computation `t(W)%*%W` in the statement block is represented using four hops – a *data(r):*W hop that reads W is fed into a Reorg hop *r(*T*)* to perform the matrix transposition, which is then fed, along with the *data(r):*W hop, into an AggregateBinary hop $ab(\sum, *)$ to perform the matrix multiplication.

The grayed *data(r)* hops represent the live-in variables for matrices W, H, and V, and the scalar i at the beginning of an iteration[8]. The grayed *data(w)* hops represent the live-out

[6]Table II describes the semantics of hops in terms of matrices. Semantics of hops for scalars are similar in spirit.

[7]Statement blocks for control structures such as *while* loops have additional HOP-Dags, e.g. for representing predicates.

[8]The max_iteration variable is used in the HOP-Dag for the *while* loop predicate.

TABLE II
EXAMPLE HOPS IN SYSTEMML: x_{ij}, y_{ij} ARE CELLS IN MATRICES X, Y, RESPECTIVELY.

HOP Type	Notation	Semantics	Example in Table I
Binary	$b(op) : X, Y$	for each x_{ij} and y_{ij}, perform $op(x_{ij}, y_{ij})$, where *op* is $*, +, -, /$ etc.	$b(*) : X, Y$
Unary	$u(op) : X$	for each x_{ij}, perform $op(x_{ij})$, where *op* is *log*, *sin* etc.	$u(log) : X$
AggregateUnary	$au(aggop, dimension) : X$	apply *aggop* for the cells in dimension, where *aggop* is $\sum$, $\prod$ etc, and *dimension* is *row* (row wise), *col* (column wise) or *all* (the whole matrix).	$au(\sum, row) : X$
AggregateBinary	$ab(aggop, op) : X, Y$	for each i, j, perform $aggop(\{op(x_{ik}, y_{kj}) \| \forall k\})$, where *op* is $*, +, -, /$ etc, and *aggop* is $\sum$, $\prod$ etc.	$ab(\sum, *) : X, Y$
Reorg	$r(op) : X$	reorganize elements in a matrix, such as transpose ($op = T$).	$r(T) : X$
Data	$data(op) : X$	read ($op = r$) or write ($op = w$) a matrix.	

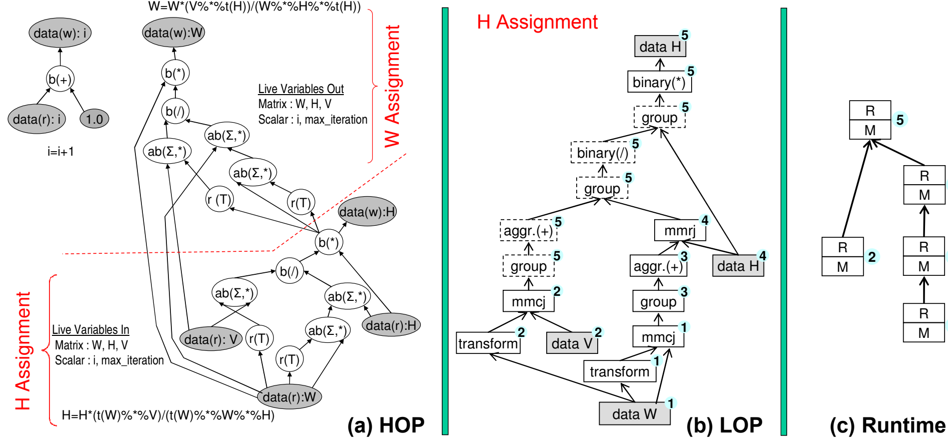


Fig. 4. HOP-Dag, LOP-Dag and Runtime of the while Loop Body in Figure 3(c)

variables at the end of an iteration that need to be passed onto the next iteration. These data hops – which are transient – implicitly connect HOP-Dags of different statement blocks by mapping the transient *data(w)* hops (sinks) of one statement block to the transient *data(r)* hops (sources) of the next statement block, or the next iteration of the *while* loop.

C. Low-Level Operator Component (LOP)

The LOP component translates HOP-Dags into corresponding low-level physical execution plans (or LOP-Dags). In this section, we detail the low-level operators (lop) that describe individual operations over key-value pairs and show how a LOP-Dag is constructed from a HOP-Dag. We also present a greedy *piggybacking heuristic* for packaging lops into small number of MapReduce jobs.

Description of lops: Lops represent basic operations in a MapReduce environment. Each lop takes one or more sets of key-value pairs as input and generates one set of key-value pairs as output that can be consumed by one or more lops. Example lops[9] are provided in Table III.

Construction of LOP-Dags: A HOP-Dag is processed in a bottom-up fashion to generate the corresponding LOP-Dag by translating each hop into one or more lops. Figure 5 describes the translation of a *Binary* hop to the corresponding lops for the expression `C/D` (Figure 3(b)). At the bottom, each of the two *data* lops returns one set of key-value pairs for the input matrices, conceptually, one entry for each cell in the individual matrices. (In practice, as will be described in Section III-D1, *data* lop typically returns multiple cells for each key where the number of cells is determined by an appropriate blocking strategy.) A *group* then groups or sorts the key-value pairs from the two inputs based on their key. Each resulting group is then passed on to a *binary* lop to perform the division of the corresponding cell-values. Other example translations of hops to lops are provided in Table I.

Figure 4(b) shows the generated LOP-Dag for the "H Assignment" part of the HOP-Dag in Figure 4(a). Note that the *AggregateBinary* hop for matrix multiplication can be translated into different sequences of lops (see the last column of the 3rd row in Table I). In our example of Figure 4(b), *mmcj* $\rightarrow$ *group* $\rightarrow$ *aggregate*($\sum$) is chosen for `t(W)%*%V` and `t(W)%*%W`, and *mmrj* is chosen for multiplying the result of (`t(W)%*%W`) with `H`.

Packaging a LOP-Dag into MapReduce jobs: Translating every lop to a MapReduce job, though straightforward, will result in multiple scans of input data and intermediate results. If, however, multiple lops can be packaged into a single MapReduce job, the resulting reduction in scans may result in an improvement in efficiency. Packing multiple lops into a single MapReduce job requires clear understanding of the following two properties of lops:

Location: whether the lop can be performed in Map, Reduce, either or both phases. Note that the execution of certain lops, such as *group*, spans both Map and Reduce phases.

Key Characteristics: whether the input keys are required

[9]Lops over scalars are omitted in the interest of space.

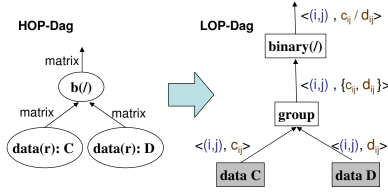


Fig. 5. Translating hop to lop for expression C/D from Figure 3(b)

to be grouped, the output keys produced are grouped, and whether the lop generates different output keys.

These properties for the individual lops are summarized in Table III. Algorithm 2 describes the greedy *piggybacking algorithm* that packs the lops in a LOP-Dag into a small number of MapReduce jobs. The nodes in a given LOP-Dag are first topologically sorted, and then partitioned into multiple lists based on their execution location property. Note that the nodes within each list are in topologically sorted order. The approach then iteratively assigns the lops to one or more MapReduce job(s). During each iteration, it allocates a new MapReduce job and assigns lops first to the Map phase, then assigns lops that span the Map and Reduce phases, and finally assigns lops to the Reduce phase. This assignment is carried out by invoking the method $addNodesByLocation$.

Lop nodes with execution locations of Map or $MapOrReduce$ can be assigned to the Map phase provided their descendants in the LOP-Dag have already been assigned. Note that the descendants of a given lop p are the ones that have a directed path to p, and they appear prior to p in a topological sort. When no more lops can be added to the Map phase, we proceed to add lops that span the Map and Reduce phases, ensuring that another descendant with execution location $MapAndReduce$ will not be assigned to the same job. Finally, lops with the execution locations of $MapOrReduce$ and $Reduce$ are directly added to the Reduce phase of the job provided their descendants have already been assigned. Group lops (with execution location $MapAndReduce$) can be added to the reduce phase provided the same MapReduce job has been assigned a descendant group lop and that none of the intermediate lops between the two group lops alter the keys. For example, consider the five lops shown as dotted boxes in Figure 4(b). The first *group* lop is assigned to span Map and Reduce phases of the job. Remaining two *group* lops are executed in the Reduce phase because the $aggr.(+)$ and $binary(/)$ lops do not alter the keys. Therefore, the entire LOP-Dag is packed into just five MapReduce jobs (see Figure 4(c)). The job number is shown next to each lop in Figure 4(b). Overall runtime complexity of our piggybacking strategy is quadratic in LOP-Dag size. While Pig [17] also makes an effort to pack multiple operators into MapReduce jobs, their approach is not readily applicable for complex linear algebraic operations.

D. Runtime

There are three main considerations in the runtime component of SystemML: *key-value representation* of matrices, an *MR runtime* to execute individual LOP-Dags over MapReduce, and a *control module* to orchestrate the execution of all the MapReduce jobs for a DML script.

Algorithm 2 Piggybacking : Packing lops that can be evaluated together in a single MapReduce job

1: Input: LOP-Dag
2: Output: A set of MapReduce Jobs($MRJobs$)
3: [$N_{Map}, N_{MapOrRed}, N_{MapAndRed}, N_{Red}$] = TopologicalSort(LOP-Dag);
4: **while** (Nodes in LOP-Dag remain to be assigned) **do**
5: Job $\leftarrow$ create a new MapReduce job;
6: addNodesByLocation($N_{Map} \cup N_{MapOrRed}$, Map, Job);
7: addNodesByLocation($N_{MapAndRed}$, MapAndReduce, Job);
8: addNodesByLocation($N_{MapOrRed} \cup N_{MapAndRed} \cup N_{Red}$, Reduce, Job);
9: add Job to $MRJobs$;
10: **end while**
11:
12: **Method:** *addNodesByLocation (S, loc, Job)*
13: **while** (true) **do**
14: Z $\leftarrow \phi$
15: **while** (S is not empty) **do**
16: $n \leftarrow S$.next()
17: **if** (n is not yet assigned and all descendants of n have been assigned) **then**
18: **if** (loc is Map) **then**
19: add n to Z
20: **else if** (loc is MapAndReduce) **then**
21: add n to Z if n does not have any descendant lop in Z and Job whose location is MapAndReduce
22: **else if** (loc is Reduce) **then**
23: add n to Z if n is not a group lop
24: if n is a group lop: add n to Z only if n has a descendant group lop in Z or Job & none of the lops between these two group lops alter keys
25: **end if**
26: **end if**
27: **end while**
28: break if Z is empty
29: add Z to Job.Map, Job.MapAndReduce, or Job.Reduce, based on loc
30: **end while**

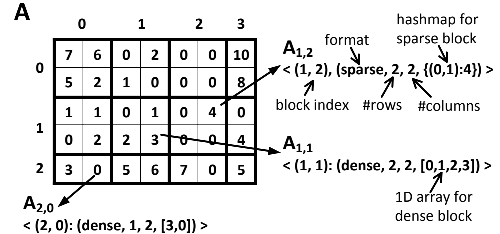


Fig. 6. Example Block Representation

1) Matrices as Key-Value Pairs: SystemML partitions matrices into blocks (using a blocking operation) and exploits local sparsity within a block to reduce the number of key-value pairs when representing matrices.

Blocking: A matrix is partitioned into smaller rectangular sub-matrices called blocks. Each block is represented as a key-value pair with the key denoting the block id and the value carrying all the cell values in the block. Figure 6 shows a matrix partitioned into 2×2 blocks. Note that cell, row and column representations are special cases of blocks. Varying the block sizes results in a trade-off between the number of key-value pairs flowing through MapReduce and the degree of parallelism in the system.

Local Sparsity: Local Sparsity refers to the sparsity of an individual block, i.e. the fraction of non-zero values in the block. To achieve storage efficiency, the actual layout of the values in a block is decided based upon its local sparsity. A parameter T_{sparse} provides a threshold to choose between a

TABLE III
EXAMPLE LOPS IN SYSTEMML: $\{\langle(i,j), x_{ij}\rangle\}$ IS THE CONCEPTUAL KEY-VALUE REPRESENTATION OF MATRIX X

LOP Type	Description	Execution Location	Key Characteristics
data	input data source or output data sink, in key value pairs $\{\langle(i,j), x_{ij}\rangle\}$	Map or Reduce	none
unary	operate on each value with an optional scalar, $\{\langle(i,j), x_{ij}\rangle\}, s \mapsto \{\langle(i,j), op(x_{ij}, s)\rangle\}$	Map or Reduce	none
transform	transform each key, $\{\langle(i,j), x_{ij}\rangle\} \mapsto \{\langle trans(i,j), x_{ij}\rangle\}$	Map or Reduce	keys changed
group	groups values by the key, $\{\langle(i,j), x_{ij}\rangle\}, \{\langle(i,j), y_{ij}\rangle\}... \mapsto \{\langle(i,j), \{x_{ij}, y_{ij}...\}\rangle\}$	Map and Reduce	output keys grouped
binary	operate on two values with the same key, $\{\langle(i,j), \{x_{ij}, y_{ij}\}\rangle\} \mapsto \{\langle(i,j), op(x_{ij}, y_{ij})\rangle\}$	Reduce	input keys grouped
aggregate	aggregate all the values with the same key, $\{\langle(i,j), values\rangle\} \mapsto \{\langle(i,j), agg(values)\rangle\}$	Reduce	input keys grouped
mmcj	cross product computation in the CPMM matrix multiplication, $\{\langle(i,k), x_{ik}\rangle\}, \{\langle(k,j), y_{kj}\rangle\} \mapsto \{\langle(i,j), op(x_{ik}, y_{kj})\rangle\}$	Map and Reduce	none
mmrj	RMM matrix multiplication, $\{\langle(i,k), x_{ik}\rangle\}, \{\langle(k,j), y_{kj}\rangle\} \mapsto \{\langle(i,j), agg(\{op(x_{ik}, y_{kj})\})\rangle\}$	Map and Reduce	none

sparse and a *dense* representation on a per-block basis. For example with $T_{sparse} = 0.3$ in Figure 6, the block $A_{1,2}$ (local sparsity 0.25) is treated as sparse, and hence, only its non-zero cells are stored. In comparison, the block $A_{1,1}$ with local sparsity 0.75 is considered dense and all its cell values are stored in a one-dimensional array.

Dynamic Block-level Operations Based on Local Sparsity: When employing blocking, all matrix operations are translated into operations on blocks at the lowest level. Local sparsity information is also used to dynamically decide on the appropriate execution of per-block operations at runtime. For every block-level operation, there are separate execution plans to account for the fact that individual blocks may be dense or sparse. Suppose we want to perform matrix multiplication on two individual blocks. The actual algorithm chosen for this operation is based on the local sparsity of the two input blocks. If both blocks are dense, the runtime chooses an algorithm that cycles through every cell in both blocks. If, however, one or both of the blocks is sparse, the runtime chooses an algorithm that operates *only* on the non-zero cells from the sparse block(s).

2) Generic MapReduce Job (G-MR): G-MR is a generic MapReduce job and is the main execution engine in SystemML. It is instantiated by the piggybacking algorithm (Algorithm 2) with the runtime instructions associated with one or more lops. The job is then executed in the MapReduce environment. As an example, consider the MapReduce job marked 1 in Figure 4(c). It contains instructions to execute three different lops – a *data*; a *transform*; and a *mmcj* (the lops are also marked 1 in Figure 4(b)). The instructions for the first two lops are executed in the Map phase of the job whereas the instruction for the third lop is executed both in Map and Reduce phases.

3) Control Module: The control module is responsible for orchestrating the execution of the instantiated MapReduce jobs for a DML script. Operations performed in the control module include scalar computations, such as arithmetic operations and predicate evaluations, and metadata operations such as deletion of intermediate results while executing the DML script.

IV. MATRIX MULTIPLICATION ALGORITHMS

For the expensive matrix multiplication operation, SystemML currently supports two alternative execution plans: RMM and CPMM. For CPMM, we describe a runtime optimization using a *local aggregator* that enables partial aggregation in the reducer. Using a cost model, we detail a comparison of the two plans under different data characteristics.

A. RMM and CPMM

Consider two matrices A and B represented in blocked format, with $M_b \times K_b$ blocks in A and $K_b \times N_b$ blocks in B. The matrix multiplication can be written in blocked format as follows: $C_{i,j} = \sum_k A_{i,k} B_{k,j}$, $i < M_b, k < K_b, j < N_b$.

RMM: The replication based matrix multiplication strategy, as illustrated in Figure 1, requires only one MapReduce job. The LOP-Dag for this execution plan contains a single *mmrj* lop. Each reducer in this strategy is responsible for computing the *final* value for one or more blocks in the resulting matrix C. In order to compute one result block $C_{i,j}$, the reducer must obtain all required blocks from input matrices, i.e., $A_{i,k}$ and $B_{k,j}$, $\forall\, k$. Since each block in A and B can be used to produce multiple result blocks in C, they need to be replicated. For example, $A_{i,k}$ is used in computing the blocks $C_{i,j}$s, $0 \le j < N_b$.

CPMM: Figure 2 demonstrates the cross product based algorithm for matrix multiplication. CPMM is represented in a LOP-Dag with three lops *mmcj* $\rightarrow$ *group* $\rightarrow$ *aggregate*($\sum$), and requires 2 MapReduce jobs for execution. The mapper of the first MapReduce job reads the two input matrices A and B and groups input blocks $A_{i,k}$s and $B_{k,j}$s by the common key k. The reducer performs a cross product to compute $P^k_{i,j} = A_{i,k} B_{k,j}$. In the second MapReduce job the mapper reads the results from the previous MapReduce job and groups all the $P^k_{i,j}$s by the key (i,j). Finally, in the Reduce phase, the *aggregate* lop computes $C_{i,j} = \sum_k P^k_{i,j}$.

B. Local Aggregator for CPMM

In CPMM, the first MapReduce job outputs $P^k_{i,j}$ for $1 \le k \le K_b$. When K_b is larger than the number of available reducers r, each reducer may process multiple cross products. Suppose a reducer applies cross products on $k = k'$ and $k = k''$, then both $P^{k'}_{i,j} = A_{i,k'} B_{k',j}$ and $P^{k''}_{i,j} = A_{i,k''} B_{k'',j}$ are computed in the same reducer. From the description of CPMM, we know that the second MapReduce job aggregates the output of the first job as $C_{i,j} = \sum_k P^k_{i,j}$. Instead of outputting $P^{k'}_{i,j}$ and $P^{k''}_{i,j}$ separately, it is more efficient to aggregate the partial results within the reducer. Note that this local aggregation is

applicable only for *mmcj*. This operation is similar in spirit to the *combiner* [6] in MapReduce, the major difference being that here partial aggregation is being performed in the reducer.

There is still the operational difficulty that the size of the partial aggregation may be too large to fit in memory. We have, therefore, implemented a disk-based local aggregator that uses an in-memory buffer pool. CPMM always generates the result blocks in a sorted order, so that partial aggregation only incurs sequential IOs with an LRU buffer replacement policy. One aspect worth noting is that no matter how many cross products get assigned to a single reducer the result size is bounded by the size of matrix C, denoted as $|C|$. We demonstrate in Section V-C, that this seemingly simple optimization significantly improves the performance of CPMM.

C. RMM vs CPMM

We start with a simple cost model for the two algorithms. Empirically, we found that the distributed file system (DFS) IO and network costs were dominant factors in the running time, and consequently we focus on these costs in our analysis.

In RMM, the mappers replicate each block of A and B, N_b and M_b times respectively. As a result, $N_b|A| + M_b|B|$ data is shuffled in the MapReduce job. Therefore, the cost of RMM can be derived as $cost(\mathit{RMM}) = \mathit{shuffle}(N_b|A| + M_b|B|) + \mathit{IO}_{dfs}(|A| + |B| + |C|)$.

In CPMM, in the first job, mappers read blocks of A and B, and send them to reducers. So, the amount of data shuffled is $|A| + |B|$. The reducers perform cross products for each k and apply a *local aggregator* to partially aggregate the results across different values of k within a reducer. The result size produced by each reducer is bounded by $|C|$. When there are r reducers in the job, the amount of data written to DFS is bounded by $r|C|$. This data is then read into the second MapReduce job, shuffled and then fed into the reducers to produce the final result. So, the total cost of CPMM is bounded by $cost(\mathit{CPMM}) \leq \mathit{shuffle}(|A| + |B| + r|C|) + \mathit{IO}_{dfs}(|A| + |B| + |C| + 2r|C|)$.

For data of the same size, *shuffle* is a more expensive operation than IO_{dfs} as it involves network overhead, local file system IO and external sorting.

The cost models discussed above provide a guideline for choosing the appropriate algorithm for a particular matrix multiplication. When A and B are both very large, CPMM is likely to perform better, since the shuffle overhead of RMM is prohibitive. On the other hand, if one matrix, say A, is small enough to fit in one block ($M_b = K_b = 1$), the cost of RMM becomes $\mathit{shuffle}(N_b|A| + |B|) + \mathit{IO}_{dfs}(|A| + |B| + |C|)$. Essentially, RMM now partitions the large matrix B and broadcasts the small matrix A to every reducer. In this case, RMM is likely to perform better than CPMM. In Section V-C, we will experimentally compare the performance of CPMM and RMM for different input data characteristics.

V. Experiments

The goals of our experimentation are to study scalability under conditions of varying data and Hadoop cluster sizes, and the effectiveness of optimizations in SystemML. For this purpose, we chose GNMF for which similar studies have been conducted recently [10], thereby enabling meaningful comparisons. Since SystemML is architected to enable a large class of ML algorithms, we also study 2 other popular ML algorithms, namely linear regression and PageRank.

A. Experimental Setup

The experiments were conducted with Hadoop 0.20 [9] on two different clusters:

- 40-core cluster: The cluster uses 5 local machines as worker nodes. Each machine has 8 cores with hyper-threading enabled, 32 GB RAM and 500 GB storage. We set each node to run 15 concurrent mappers and 10 concurrent reducers.
- 100-core EC2 cluster: The EC2 cluster has 100 worker nodes. Each node is an EC2 small instance with 1 compute unit, 1.7 GB memory and 160 GB storage. Each node is set to run 2 mappers and 1 reducer concurrently.

The datasets are synthetic, and for given dimensionality and sparsity, the data generator creates random matrices with uniformly distributed non-zero cells. A fixed matrix block size (c.f. Section III-D1) of 1000×1000 is used for all the experiments, except for the matrix blocking experiments in Section V-C. For the local aggregator used in CPMM, we use an in-memory buffer pool of size 900 MB on the 40-core cluster and 500 MB on the 100-core EC2 cluster.

B. Scalability

We use GNMF shown in Script 1 as a running example to demonstrate scalability on both the 40-core cluster and the 100-core cluster.

The input matrix V is a sparse matrix with d rows and w columns. We fix w to be 100,000 and vary d. We set the sparsity of V to be 0.001, thus each row has 100 non-zero entries on average. The goal of GNMF algorithm is to compute dense matrices W of size $d \times t$ and H of size $t \times w$, where $V \approx WH$. t is set to 10 (As described in Section I in the context of topic modeling, t is the number of topics.). Table IV lists the characteristics of V, W and H used in our setup.

Baseline single machine comparison: As a baseline for comparing SystemML, we first run GNMF using 64-bit version of R on a single machine with 64 GB memory. Figure 7(a) shows the execution times for one iteration of the algorithm with increasing sizes of V. For relatively small sizes of V, R runs very efficiently as the data fits in memory. However, when the number of rows in V increases to 10 million (1 billion non-zeros in V), R runs out of memory, while SystemML continues to scale.

Comparison against best known published result: [10] introduces a hand-coded MapReduce implementation of GNMF. We use this MapReduce implementation as a baseline to evaluate the efficiency of the execution plan generated by SystemML as well as study the performance overhead of our

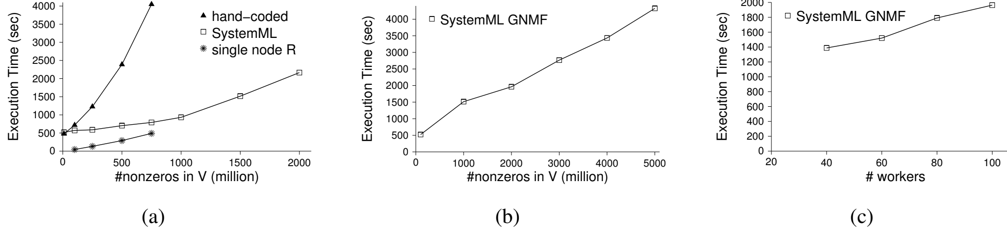


Fig. 7. Scalability of GNMF: (a) increasing data size on 40-core cluster, (b) increasing data size on 100-core cluster, (c) increasing data size and cluster size

generic runtime[10]. For a fair comparison, we re-implemented the algorithm as described in the paper and ran it on the same 40-core cluster as the SystemML generated plan. The hand-coded algorithm contains 8 full MapReduce jobs and 2 map-only jobs, while the execution plan generated by SystemML consists of 10 full MapReduce jobs. For the hand-coded algorithm, the matrices are all prepared in the required formats: V is in cell representation, W is in a row-wise representation and H is in a column-wise representation. For the SystemML plan, the input matrices are all in block representation with block size 1000×1000. Figure 7(a) shows the performance comparison of SystemML with the hand-coded implementation. Surprisingly, the performance of SystemML is significantly better than the hand-coded implementation. As the number of non-zeros increases from 10 million to 750 million, execution time on SystemML increases steadily from 519 seconds to around 800 seconds, while execution time for the hand-coded plan increases dramatically from 477 seconds to 4048 seconds! There are two main reasons for this difference. First, SystemML uses the block representation for V, W, and H, while in the hand-coded implementation, the largest matrix V is in cell representation. As discussed in Section III-D1 and to be demonstrated in Section V-C, the block representation provides significant performance advantages over the cell representation. Second, the hand-coded implementation employs an approach very similar to CPMM for the two most expensive matrix multiplications in GNMF: `t(W)%*%V` and `V%*%t(H)`, but without the local aggregator (see Section IV-B). As will be shown in Section V-C, CPMM with local aggregation significantly outperforms CPMM without local aggregation.

Scalability on 100-core EC2 cluster: To test SystemML on a large cluster, we ran GNMF on a 100-core EC2 cluster. In the first experiment, we fixed the number of nodes in the cluster to be 100, and ran GNMF by varying the number of non-zero values from 100 million to 5 billion. Figure 7(b) demonstrates the scalability of SystemML for one iteration of GNMF. In the second experiment (shown in Figure 7(c)), we varied the number of worker nodes from 40 to 100 and scaled the problem size proportionally from 800 million non-zero values to 2 billion non-zeros. The ideal scale-out behavior would depict a flat line in the chart. However, it is impossible to realize this ideal scale-out behavior due to many factors such as network overheads. Nevertheless, Figure 7(c) presents a steady increase in execution time with the growth in data and cluster size.

Besides scalability, DML improves productivity and reduces development time of ML algorithms significantly. For example, GNMF is implemented in 11 lines of DML script, but requires more than 1500 lines of Java code in the hand-coded implementation. Similar observations have been made in [18] regarding the power of declarative languages in substantially simplifying distributed systems programming.

C. Optimizations

RMM vs CPMM: We now analyze the performance differences between alternative execution plans for matrix multiplication, RMM and CPMM. We consider three examples from GNMF (Script 1): `V%*%t(H)`, `W%*%(H%*%t(H))`, and `t(W)%*%W`. To focus on matrix multiplication, we set `H'=t(H)`, `S=H%*%t(H)`, and `W'=t(W)`. Then the three multiplications are defined as: `V%*%H'`, `W%*%S` and `W'%*%W`. The inputs of these three multiplications have very distinct characteristics as shown in Table IV. With d taking values in millions, V is a very large matrix; H' is a medium sized matrix; W' and W are very tall and skinny matrices; and S is a tiny matrix. We compare execution times for the two alternative algorithms for the three matrix multiplications in Figures 8(a), 8(b) and 8(c).

Note that neither of the algorithms always outperforms the other with their relative performance depending on the data characteristics as described below.

For `V%*%H'`, due to the large sizes of both V and H', CPMM is the preferred approach over RMM, because the shuffling cost in RMM increases dramatically with the number of rows in V.

For `W%*%S`, RMM is preferred over CPMM, as S is small enough to fit in one block, and RMM essentially partitions W and broadcasts S to perform the matrix multiplication.

For `W'%*%W`, the cost for RMM is *shuffle*$(|W'| + |W|) + IO_{dfs}(|W'| + |W| + |S|)$ with a degree of parallelism of only 1, while the cost of CPMM is roughly *shuffle*$(|W'| + |W| + r|S|) + IO_{dfs}(2r|S| + |W'| + |W| + |S|)$. For CPMM, the degree of parallelization is $d/1000$, which ranges from 1000 to 50000 as d increases from 1 million to 50 million. When d is relatively small, even though the degree of parallelization is only 1, the advantage of the low shuffle cost makes RMM perform better than CPMM. However, as d increases, CPMM's

[10] Through personal contact with the authors of [10], we were informed that all the scalability experiments for the hand-coded GNMF algorithm were conducted on a proprietary SCOPE cluster with thousands of nodes, and the actual number of nodes scheduled for each execution was not known.

TABLE IV
CHARACTERISTICS OF MATRICES.

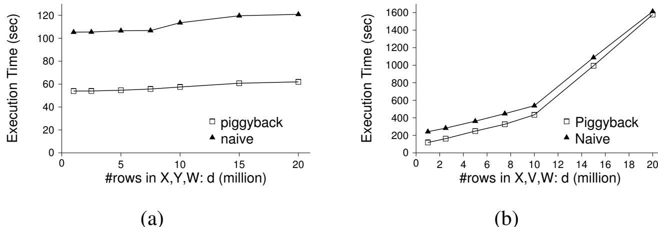
Matrix	X,Y,W	H'	V	W'	S	H
Dimension	$d \times 10$	$100,000 \times 10$	$d \times 100,000$	$10 \times d$	10×10	$10 \times 100,000$
Sparsity	1	1	0.001	1	1	1
#non zeros	$10d$	1 million	$100d$	$10d$	100	1 million

TABLE V
FILE SIZES OF MATRICES FOR DIFFERENT d (BLOCK SIZE IS 1000X1000)

	d (million)	1	2.5	5	7.5	10	15	20	30	40	50
V	# non zero (million)	100	250	500	750	1000	1500	2000	3000	4000	5000
	Size (GB)	1.5	3.7	7.5	11.2	14.9	22.4	29.9	44.9	59.8	74.8
X,Y,W,W'	# non zero (million)	10	25	50	75	100	150	200	300	400	500
	Size (GB)	0.077	0.191	0.382	0.573	0.764	1.1	1.5	2.2	3.0	3.7

higher degree of parallelism makes it outperform RMM. Overall, CPMM performs very stably with increasing sizes of W' and W.

Piggybacking: To analyze the impact of piggybacking several lops into a single MapReduce job, we compare piggybacking to a naive approach, where each lop is evaluated in a separate MapReduce job. Depending on whether a single lop dominates the cost of evaluating a LOP-Dag, the piggybacking optimization may or may not be significant. To demonstrate this, we first consider the expression `W*(Y/X)` with `X=W%*%H%*%t(H)` and `Y=V%*%t(H)`. The matrix characteristics for X, Y, and W are listed in Tables IV and V. Piggybacking reduces the number of MapReduce jobs from 2 to 1 resulting in a factor of 2 speed-up as shown in Figure 9(a). On the other hand, consider the expression `W*(V%*%t(H)/X)` from the GNMF algorithm (step 8), where `X=W%*%H%*%t(H)`. While piggybacking reduces the number of MapReduce jobs from 5 to 2, the associated performance gains are small as shown in Figure 9(b).



(a) (b)

Fig. 9. Piggybacking or not: (a) `W*(Y/X)`, (b) `W*(V%*%t(H)/X)`

Matrix Blocking: Table VI shows the effect of matrix blocking on storage and computational efficiency (time) using the expression `V%*%H'`. As a baseline, the table also includes the corresponding numbers for the cell representation. The matrix characteristics for V with d=1 million rows and H are listed in Table IV. The execution time for the expression improves by orders of magnitude from hours for the cell representation to minutes for the block representation.

The impact of block size on storage requirements varies for sparse and dense matrices. For dense matrix H', blocking significantly reduces the storage requirements compared to the cell representation. On the other hand, for sparse matrix V, small block sizes can increase the storage requirement compared to the cell representation, since only a small fraction of the cells are non-zero per block and the per block metadata space requirements are relatively high.

TABLE VI
COMPARISON OF DIFFERENT BLOCK SIZES

Block Size	1000x1000	100x100	10x10	cell
Execution time	117sec	136sec	3hr	>5hr
Size of V (GB)	1.5	1.9	4.8	3.0
Size of H' (MB)	7.8	7.9	8.1	31.0

Figure 10(a) shows the performance comparison for different block sizes with increasing matrix sizes[11]. This graph shows that the performance benefit of using a larger block size increases as the size of V increases.

Local Aggregator for CPMM: To understand the performance advantages of using a local aggregator (Section IV-B), consider the evaluation of `V%*%H'` (V is $d \times w$ matrix, and H' is $w \times t$ matrix). The matrix characteristics for V and H' can be found in Tables IV and V. We first set $w = 100,000$ and $t = 10$. In this configuration, each reducer performs 2 cross products on average, and the ideal performance gain through local aggregation is a factor of 2. Figure 10(b) shows the benefit of using the local aggregator. As d increases from 1 million to 20 million, the speedup ranges from 1.2 to 2.

We next study the effect of w, by fixing d at 1 million and varying w from 100,000 to 300,000. The number of cross products performed in each reducer increases as w increases. Consequently, as shown in Figure 11(a), the intermediate result of *mmcj* increases linearly with w when a local aggregator is not deployed. On the other hand, when a local aggregator is applied, the size of the intermediate result stays constant as shown in the figure. Therefore, the running time with a local aggregator increases very slowly while without an aggregator the running time increases more rapidly (see Figure 11(b)).

D. Additional Algorithms

In this section, we showcase another two classic algorithms written in DML: Linear Regression and PageRank [19].

Linear Regression: Script 2 is an implementation of a conjugate gradient solver for large, sparse, regularized linear

[11] Smaller block sizes were ignored in this experiment since they took hours even for 1 million rows in V.

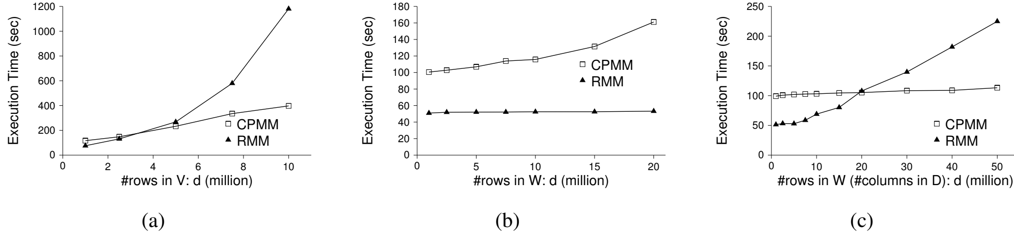


Fig. 8. Comparing two alternatives of matrix multiplication: (a) `V%*%H'`, (b) `W%*%S`, (c) `W'%*%W`

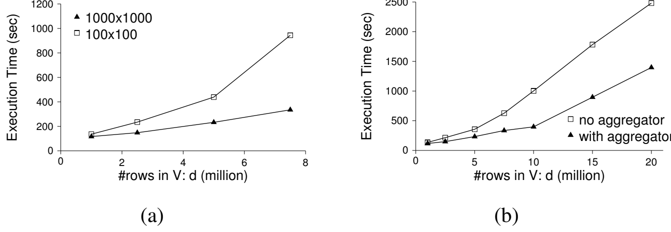


Fig. 10. (a) Execution time with different block sizes, (b) Advantage of local aggregator with increasing d

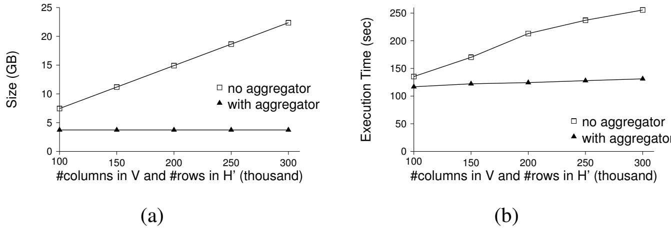


Fig. 11. CPMM with increasing w: (a) intermediate result size, (b) execution time

regression problems. In the script below, V is a data matrix (sparsity 0.001) whose rows correspond to training data points in a high-dimensional, sparse feature space. The vector b is a dense vector of regression targets. The output vector w has the learnt parameters of the model that can be used to make predictions on new data points.

Script 2: Linear Regression

```
1: V=readMM("in/V", rows=1e8, cols=1e5, nnzs=1e10);
2: y=readMM("in/y", rows=1e8, cols=1);
3: lambda = 1e-6; // regularization parameter
4: r=-(t(V) %*% y) ;
5: p=-r ;
6: norm_r2=sum(r*r);
7: max_iteration=20;
8: i=0;
9: while(i<max_iteration){
10:  q=((t(V) %*% (V %*% p)) + lambda*p)
11:  alpha= norm_r2/(t(p)%*%q);
12:  w=w+alpha*p;
13:  old_norm_r2=norm_r2;
14:  r=r+alpha*q;
15:  norm_r2=sum(r*r);
16:  beta=norm_r2/old_norm_r2;
17:  p=-r+beta*p;
18:  i=i+1;}
19:writeMM(w, "out/w");
```

PageRank: Script 3 shows the DML script for the PageRank algorithm. In this algorithm, G is a row-normalized adjacency matrix (sparsity 0.001) of a directed graph. The procedure uses power iterations to compute the PageRank of every node in the graph.

Figures 12(a) and 12(b) show the scalability of SystemML for linear regression and PageRank, respectively. For linear regression, as the number of rows increases from 1 million to 20 million (non-zeros ranging from 100 million to 2 billion), the execution time increases steadily. The PageRank algorithm also scales nicely with increasing number of rows from 100 thousand to 1.5 million (non-zeros ranging from 100 million to 2.25 billion).

Script 3: PageRank

```
1: G=readMM("in/G", rows=1e6, cols=1e6, nnzs=1e9);
//p: initial uniform pagerank
2: p=readMM("in/p", rows=1e6, cols=1);
//e: all-ones vector
3: e=readMM("in/e", rows=1e6, cols=1);
//ut: personalization
4: ut=readMM("in/ut", rows=1, cols=1e6);
5: alpha=0.85; //teleport probability
6: max_iteration=20;
7: i=0;
8: while(i<max_iteration){
9:  p=alpha*(G%*%p)+(1-alpha)*(e%*%ut%*%p);
10: i=i+1;}
11:writeMM(p, "out/p");
```

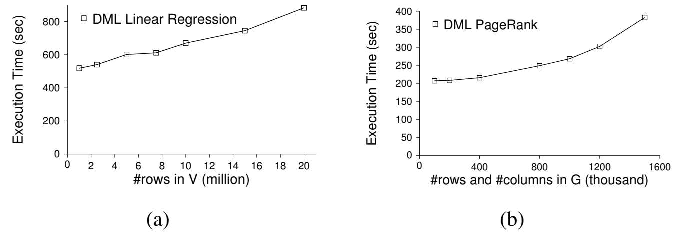


Fig. 12. (a) Execution of Linear Regression with increasing data size on 40-core cluster, (b) Execution of PageRank with increasing data size on 40-core cluster

VI. Related Work

The increasing demand for massive-scale analytics has recently spurred many efforts to design systems that enable distributed machine learning. DryadLINQ [20] is a compiler which translates LINQ programs into a set of jobs that can be executed on the Microsoft Dryad platform. LINQ is a .NET extension that provides declarative programming for data manipulation. The DryadLINQ set of language extensions is supported in C# and facilitates the generation and optimization of distributed executions plans for the specified portions of the C# program. The Large Vector Library built on top of DryadLINQ provides simple mathematical primitives and datatypes using which machine learning algorithms can be

implemented in C#. However, unlike SystemML, the onus of identifying the data parallel components of an algorithm and expressing them in DryadLINQ expressions is still left to the programmer.

Apache Mahout [13] provides a library of ML algorithms written in Hadoop. Compared to SystemML's declarative approach, Mahout requires detailed implementation for new algorithms and change of existing code for performance tuning.

Pegasus [21] is a Hadoop-based library that implements a class of graph mining algorithms that can be expressed via repeated matrix-vector multiplications. The generic versions of the key Pegasus primitive is subsumed by matrix multiplication operators in SystemML. Unlike Pegasus, SystemML does not consider a single primitive in isolation but attempts to optimize a sequence of general linear algebra computations.

Spark [22] is a cluster computing framework that allows users to define distributed datasets that can be cached in memory across machines for applications that require frequent passes through them. Likewise, parallel computing toolboxes in Matlab and R allow distributed objects to be created and message-passing functions to be defined to implement data and task parallel algorithms. By contrast, SystemML is designed to process massive datasets that may not fit in distributed memory and need to reside on disk. R packages like rmpi, rpvm expose the programmer directly to message-passing systems with significant expectation of parallel programming expertise. Other popular packages such as SNOW [23] that are built on top of MPI packages are relatively more convenient but still low-level (e.g., requiring explicit distribution of data across worker nodes) in comparison to the rapid prototyping environment enabled by SystemML. Ricardo [24] is another R-Hadoop system where large-scale computations are expressed in JAQL, a high level query interface on top of Hadoop, while R is called for smaller-scale single-node statistical tasks. This requires the programmer to identify scalability of different components of an algorithm, and re-express large-scale matrix operations in terms of JAQL queries. Similar to Ricardo, the RHIPE [25] package provides the capability to run an instance of R on each Hadoop node, with the programmer needing to directly write Map and Reduce functions.

VII. Conclusions

In this paper we have presented SystemML - a system that enables the development of large-scale machine learning algorithms. SystemML applies a sequence of transformations to translate DML scripts, the language in which machine learning algorithms are expressed, into highly optimized execution plans over MapReduce. Starting with program analysis, these transformations include evaluation and selection of alternative execution plans and final assembly of lower-level operators into small number of MapReduce jobs. Systematic empirical results in this paper have shown the benefit of a number of optimization strategies such as blocking, piggybacking and local aggregation, and the applicability of SystemML to scale-up a diverse set of machine learning algorithms. Ongoing work includes system-wide cost-based optimizations, e.g., evaluation and selection of multiple matrix blocking strategies, development of additional constructs to support machine learning meta-tasks such as model selection, and enabling a large class of algorithms to be probed at an unprecedented data scale.

References

[1] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *ICML*, 2008.

[2] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, 2006, pp. 281–288.

[3] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, "Parallel support vector machines: The cascade svm," in *NIPS*, 2004.

[4] E. Y. Chang, H. Bai, and K. Zhu, "Parallel algorithms for mining large-scale rich-media data," in *MM '09: Proceedings of the seventeen ACM international conference on Multimedia*, 2009.

[5] S. A. Robila and L. G. Maciak, "A parallel unmixing algorithm for hyperspectral images," in *Intelligent Robots and Computer Vision*, 2006.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *CACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[8] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, 2008.

[9] Apache, "Apache hadoop," http://hadoop.apache.org/.

[10] C. Liu, H. chih Yang, J. Fan, L.-W. He, and Y.-M. Wang, "Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce," in *WWW*, 2010, pp. 681–690.

[11] A. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *WWW*, 2007, pp. 271–280.

[12] B. Panda, J. Herbach, S. Basu, and R. J. Bayardo, "Planet: Massively parallel learning of tree ensembles with mapreduce," *PVLDB*, vol. 2, no. 2, pp. 1426–1437, 2009.

[13] Apache, "Apache mahout," http://mahout.apache.org/.

[14] D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," 2001.

[15] Mathworks, "Matlab," http://www.mathworks.com.

[16] R. team, "The r project for statistical computing," http://www.r-project.org.

[17] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proc. VLDB Endow.*, vol. 2, pp. 1414–1425, 2009.

[18] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears, "Boom analytics: exploring data-centric, declarative programming for the cloud," in *EuroSys*, 2010, pp. 223–236.

[19] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford University, Tech. Rep., 1999.

[20] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system - implementation and observations," in *ICDM*, 2009.

[22] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-53, May 2010.

[23] L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova, "Snow: Simple network of workstations," http://cran.r-project.org/web/packages/snow.

[24] S. Das, Y. Sismanis, K. Beyer, R. Gemulla, P. Haas, and J. McPherson, "Ricardo: Integrating r and hadoop," in *SIGMOD*, 2010.

[25] S. Guha, "Rhipe: R and hadoop integrated processing environment," http://ml.stat.purdue.edu/rhipe.