Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML

Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen,
Yuanyuan Tian, Douglas R. Burdick, Shivakumar Vaithyanathan

IBM Research – Almaden; San Jose, CA, USA

ABSTRACT

SystemML aims at declarative, large-scale machine learning (ML) on top of MapReduce, where high-level ML scripts with R-like syntax are compiled to programs of MR jobs. The declarative specification of ML algorithms enables—in contrast to existing large-scale machine learning libraries—automatic optimization. SystemML's primary focus is on data parallelism but many ML algorithms inherently exhibit opportunities for task parallelism as well. A major challenge is how to efficiently combine both types of parallelism for arbitrary ML scripts and workloads. In this paper, we present a systematic approach for combining task and data parallelism for large-scale machine learning on top of MapReduce. We employ a generic Parallel FOR construct (`ParFOR`) as known from high performance computing (HPC). Our core contributions are (1) complementary parallelization strategies for exploiting multi-core and cluster parallelism, as well as (2) a novel cost-based optimization framework for automatically creating optimal parallel execution plans. Experiments on a variety of use cases showed that this achieves both efficiency and scalability due to automatic adaptation to ad-hoc workloads and unknown data characteristics.

1. INTRODUCTION

Large-scale data analytics have become an integral part of online services, enterprise data management, system management, and scientific applications in order to gain value from huge amounts of collected data [6, 14]. Finding interesting unknown facts and patterns often requires to analyze the full data set instead of applying sampling techniques [6]. Recent approaches mainly address this challenge by leveraging parallel programming paradigms such as MapReduce (MR) [8], its open-source implementation Hadoop, or more general data flow abstractions [17, 32]. These frameworks enable large-scale, fault-tolerant, and cost-effective parallelization on commodity hardware. Often high-level languages are used in order to overcome the complexity of multiple MR jobs per query. Examples are Jaql [3], Pig [26], and Hive [30], which all compile queries to MR jobs for low programming effort and good out-of-the-box performance.

Besides analyzing big data, the second driving force of large-scale analytics is the increasing need for advanced analytics beyond traditional aggregation queries, in terms of machine learning (ML) and content analysis [6]. These analytics range from descriptive statistics to data mining techniques such as clustering, classification, regression, and association rule mining. Typical applications are log and sales analysis, recommendations, and customer classifications. While existing statistical tools such as R and Matlab provide a rich variety of advanced analysis libraries, they are not designed—except specific packages [9]—for distributed computing on big data. Existing work on large-scale ML ranges from tailor-made distributed ML algorithms [2] to integrating R into higher-level languages [7]. However, these approaches require the user to choose the parallelization strategy—i.e., an appropriate algorithm implementation—upfront, which is often difficult in ad-hoc analysis scenarios.

In contrast to existing work, SystemML [11, 31] and other systems like Cumulon [16] enable *declarative* machine learning. Complex ML algorithms are expressed in a high-level language—including linear algebra—and compiled to programs of MR jobs. This is comparable to languages such as Jaql, Pig, and Hive but domain-specific for machine learning. The major advantages of this high-level language approach are the flexible specification of large-scale ML algorithms and automatic cost-based optimization. However, there are also fundamental challenges.

Challenges: Any ML system on top of MapReduce faces two fundamental problems: First, for small datasets, MR performance is often orders of magnitude worse than parallel *in-memory computation.* The reasons are distributed operator implementations, distributed/local file system I/O, and MR framework overhead. Second, the data-parallel MR paradigm does not inherently support *task parallelism* for compute-intensive workloads. However, there are many use cases such as descriptive statistics, cross-validation, or ensemble learning that would strongly benefit from task parallelism. The major challenge is to provide *efficiency and scalability* for the *full spectrum* from many small to few very large tasks. Interestingly, both problems are interrelated due to memory constraints and—as we will show—can be jointly solved in a dedicated cost-based optimization framework.

Contributions: The primary contribution of this paper is a systematic approach for combining task and data parallelism for large-scale machine learning on top of MapReduce. Our core idea is to employ a dedicated `ParFOR` (Parallel FOR) construct as known from high-performance computing (HPC) and to create optimal parallel execution plans. For high efficiency and scalability on a wide variety of use cases, we introduce (1) complementary `ParFOR` parallelization strategies including novel techniques for access-aware data partitioning and locality, and (2) a novel cost-based optimization framework for task-parallel ML programs. In detail, we make the following technical contributions:

- *Foundations:* After a brief up-to-date background description of SystemML, we introduce a taxonomy of task-parallel ML programs, conceptually formulate the problem, and discuss correctness in Section 2.
- *Runtime:* We further explain local and remote parallelization strategies for exploiting multi-core and cluster parallelism in Section 3. This includes techniques for task partitioning, result aggregation, and novel techniques for data partitioning and co-location.
- *Optimizer:* Subsequently, we introduce the novel cost-based optimization for hierarchies of `ParFOR` loops and parallel instructions in Section 4. We formulate the optimization problem, describe our time- and memory-based cost model, and present our heuristic optimizer.
- *Results:* Finally, we present experimental results and insights into our optimizer in Section 5. We study the effectiveness of parallelization strategies on a variety of use cases, including comparisons to R and Spark.

2. BACKGROUND AND FOUNDATIONS

As foundations for *hybrid* parallelization strategies, we (1) describe the background of SystemML as a data-parallel ML system, (2) introduce the *complementary* design space of task-parallel ML programs via a new taxonomy, and finally, (3) formulate the `ParFOR` optimization problem.

2.1 Background SystemML

In the interest of a clear presentation, we briefly describe the architecture and execution model of SystemML [11, 31].

System Architecture: Figure 1 shows the high-level architecture of SystemML. ML algorithms are expressed in DML (Declarative Machine learning Language) with R-like syntax. DML scripts are parsed to a hierarchical representation of statement blocks and statements, where statement blocks are defined by the program structure. Each statement block is then compiled into DAGs of high-level operators (HOPs), low-level operators (LOPs), and finally to runtime plans of program blocks and instructions. At each compilation step, we apply different optimizations. Examples include constant propagation/folding, common subexpression elimination (CSE), operator ordering, operator selection, recompilation decisions, and piggybacking (packing multiple instructions into a single MR job). At runtime, the control program executes the hierarchy of program blocks and instructions. Instructions are either CP (control program) instructions that are locally executed in-memory of the master process, or MR instructions that are executed as part of generic MR jobs on a Hadoop cluster, submitted via MR-job instructions. MR instructions work on blocks of matrices. The exchange of intermediate results between MR jobs and CP instructions is done via file exchange over the distributed file system (HDFS). A multi-level buffer pool controls this exchange and in-memory matrices. There are again several optimizations including decisions on dense/sparse matrix block representations and dynamic recompilation.

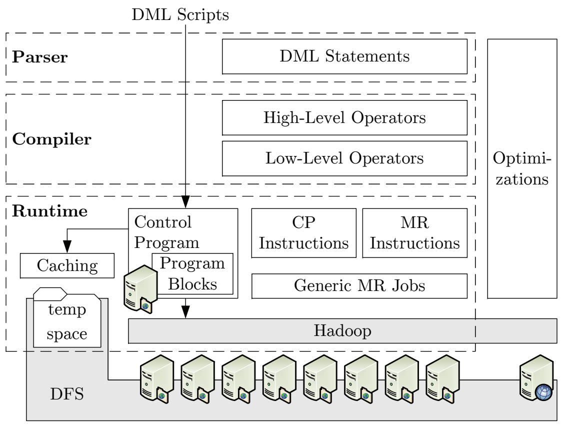


Figure 1: SystemML Architecture.

EXAMPLE 1. ***(Correlation)*** *We compute Pearson's Correlation Coefficient* $r_{X,Y} = \text{cov}(\mathbf{X},\mathbf{Y})/\sigma_X\sigma_Y$ *of two* $m \times 1$ *vectors* $\mathbf{X}$ *and* $\mathbf{Y}$*. For a numerical stable realization in DML, we compute the standard deviations via the second central moment, the covariance, and finally the coefficient* $r_{X,Y}$*:*

```
X = read( "./input/X" ); #data on HDFS
Y = read( "./input/Y" );
m = nrow(X);
sigmaX = sqrt( centralMoment(X,2)*(m/(m-1.0)) );
sigmaY = sqrt( centralMoment(Y,2)*(m/(m-1.0)) );
r = cov(X,Y) / (sigmaX * sigmaY);
write( r, "./output/R" );
```

Since this script is parsed to a single statement block, we create a single HOP DAG as shown in Figure 2. For example, the statement `r = cov(X,Y) / (sigmaX * sigmaY)` is compiled to three binary operators (`b(/)`, `b(cov)`, `b(*)`) as part of this DAG. On LOP level, we then decide on the execution strategy per operator. If $\mathbf{X}$ *and* $\mathbf{Y}$ *fit in memory, those three HOPs can be compiled into a partial CP LOP DAG of `BinaryCP(/)`, `CoVariance`, and `BinaryCP(*)`; otherwise the HOP `b(cov)` would be compiled to an MR LOP chain of `CombineBinary` (aligns blocks of* $\mathbf{X}$ *and* $\mathbf{Y}$*) and `CoVariance` (computes covariance incrementally). Finally, we create a program block of executable CP/MR instructions, where we pack multiple MR instructions into shared MR jobs. At runtime, we sequentially execute this program with materialized intermediates between instructions. Further details on the runtime operators `cm` and `cov` can be found in [31].*

This execution model exploits data parallelism via MR instructions whenever useful but results in a serial execution of independent tasks such as independent iterations. We therefore introduce *hybrid* `ParFOR` parallelization strategies for combining both task and data parallelism because there are many use cases that would strongly benefit.

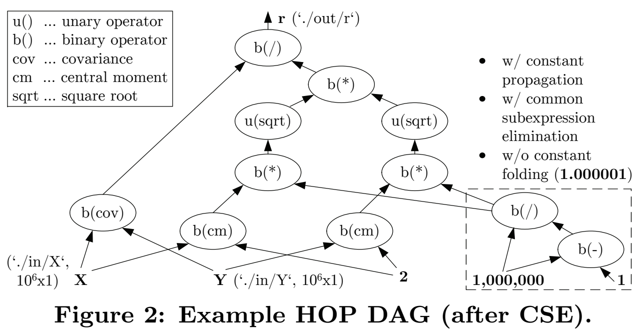


Figure 2: Example HOP DAG (after CSE).

Table 1: Taxonomy of Task-Parallel ML Algorithms.

	Single Model	Multiple Models
Disjoint Data	SQM [5], Data Gen., SGD [10]	Univariate Stats, Indep. Models
Overlap. Data	SQM [5], C. SVM [12], ALS, EM, SGD* [10]	Bivariate Stats, Meta, CV
All Data	Dist.-based, kNN, EL	Meta, EL

2.2 A Taxonomy for ML Task Parallelism

We now introduce a taxonomy of task-parallel ML programs as a basis for reasoning about classes of use cases. In our context, *data parallelism* refers to operator-/DAG-level parallelization, i.e., executing an operation on blocks of matrices in parallel. In contrast, *task parallelism* refers to program-level parallelization, i.e., executing a complex ML program on iterations in parallel. Our taxonomy (see Table 1) employs two perspectives: model- and data-oriented. First, the model-oriented perspective describes the ML-algorithm-specific statistical model we compute or use. *Multiple* (independent) models inherently exhibit large potential for task parallelism. Examples are cross-validation (CV) or ensemble learning (EL). There are also many use cases of *single* composable models, i.e., decomposable problems and aggregation of submodels, that could benefit from task parallelism. Example classes are (1) algorithms in statistical query model (SQM) summation form [5], (2) partitioned matrix factorization via alternating least squares (ALS), expectation maximization (EM), or stochastic gradient decent (SGD) (see [10] for a comprehensive survey), and (3) tree-based algorithms like decision trees or Cascade support vector machines (SVMs) [12]. Second, the data-oriented view describes the data access characteristics of iterations, which may use *disjoint*, *overlapping*, or *all* data. Those data categories define applicable optimizations such as partitioning (*disjoint/overlapping*) and memoization/sharing (*overlapping/all*). The bottom line is, we have a variety of use cases with diverse computation and data access characteristics.

2.3 Problem and Correctness

At the language-level, we provide ParFOR as a high-level primitive for task parallelism. We now introduce our running example, formulate the conceptual problem, and sketch our dependency analysis for ensuring result correctness.

EXAMPLE 2. ***(Pairwise Correlation)*** *We now extend Example 1 to parallel correlation computation for all $n(n-1)/2$ pairs of columns of an $m \times n$ input matrix* $\mathbf{D}$*. This use case is representative for more complex bivariate statistics.*

```
D = read("./input/D");
m = nrow(D);
n = ncol(D);
R = matrix(0, rows=n, cols=n);
parfor( i in 1:(n-1) ) {
   X = D[ ,i];
   m2X = centralMoment(X,2);
   sigmaX = sqrt( m2X*(m/(m-1.0)) );
   parfor( j in (i+1):n ) {
      Y = D[ ,j];
      m2Y = centralMoment(Y,2);
      sigmaY = sqrt( m2Y*(m/(m-1.0)) );
      R[i,j] = cov(X,Y) / (sigmaX*sigmaY);
}}
write(R, "./output/R");
```

The outer ParFOR loop iterates from 1 to $(n-1)$ and computes σ for the first column. Due to symmetry of $r_{X,Y}$, the inner loop only iterates from $(i+1)$ to n in order to compute $r_{X,Y}$ for all pairs of columns. In later discussions, we will refer to R[i,j]=v and v=D[,i] as left and right indexing, respectively. The result is an upper-triangular matrix $\mathbf{R}$.

A Case for Optimization: Given a variety of use cases (see Table 1) and workloads, there is a strong need for different parallel execution strategies. Recall our running example; if we have many small pairs, we aim for distributed in-memory computation, but if we have few very large pairs, we are interested in scalable data-parallel computation. Additional challenges of this example are: (1) a triangular nested loop control structure, (2) a column-wise data access on unordered distributed data, and (3) a bivariate all-to-all data shuffling pattern. Putting it altogether, complex ParFOR programs and ad-hoc data analysis with unknown input characteristics require automatic optimization.

DEFINITION 1. ***(ParFOR Optimization Problem)*** *Given a ParFOR body denoted prog, a ParFOR predicate $p = ([a, b], z)$ with lower bound a, upper bound b and increment z, as well as a cluster configuration cc, find a parallel execution plan that minimizes the execution time T with $\phi_1 : \min T(prog(p))$ s.t. $k \leq ck \wedge m \leq cm$, where k is the degree of parallelism, m is the memory consumption, and ck, cm being constraints. Note that the predicate p defines $N = \lceil (b - a + 1)/z \rceil$ iterations, where a single iteration executes prog exactly once for a specific value of the index variable and prog(p) must create correct results.*

Dependency Analysis: In order to guarantee result correctness for parallel execution, we apply a loop dependency analysis. We employ existing techniques from HPC compilers [19] and extend those slightly. For ensuring *determinism* and *independence*, we disprove the existence of any inter-iteration (loop-carried) dependencies. We use a candidate-based algorithm based on the conceptual framework of linear functions. First, we collect dependency candidates $\mathcal{C}$, where a candidate $c \in \mathcal{C}$ is defined as a write to a non-local variable. Second, each candidate $c \in \mathcal{C}$ is checked via a sequence of tests (scalar, constant, equals, GCD/Banerjee [19]) against all written and read variables of the ParFOR body. If we cannot prove independence, we add c to $\mathcal{C}'$. For range/set indexing, we introduce artificial index variables and bounds according to the given range. Third, if $\mathcal{C}' = \emptyset$, there is no loop-carried dependency and the test succeeds; otherwise, we raise an error. Finally, note that we do not aim for automatic parallelization of all for loops because this would hide potential false positive dependencies.

3. PARALLELIZATION STRATEGIES

In order to support the spectrum of use cases, we provide complementary ParFOR parallelization strategies. They all adhere to a conceptual *master/worker pattern*: iterations are logically grouped to tasks $\mathcal{W}$, k parallel workers execute those tasks, and finally worker results are merged. Hence, there are three major aspects to consider: task partitioning, parallel execution, and result merge. Accordingly, we now discuss complementary techniques for task partitioning, two physical realizations of parallel workers, hybrid parallelization, result merging, and runtime optimizations for large inputs. Finally, our cost-based optimizer exploits these strategies in order to generate efficient parallel execution plans.

3.1 Task Partitioning

Task partitioning groups iterations to tasks with two major but contradictory goals: (1) low task communication overhead (via few tasks) and (2) good load balance (via many tasks). Both aspects have high impact. For example, on MapReduce, task setup can take seconds. However, we need many tasks to exploit large clusters and load balance is crucial because the most time-consuming worker determines the overall execution time. We model a task $w_i \in \mathcal{W}$ as a logical group of one or many (sequentially executed) iterations with task size $l_i = |w_i|$. Additionally, $\mathcal{W}$ is defined as a set of disjoint tasks that exactly cover predicate p.

Fixed-Size Schemes: Fixed-size task partitioning creates tasks with constant size $l_i = cl$ which trades communication overhead and load balance. One extreme is *naïve* task partitioning with minimal task sizes of $l_i = 1$ that leads to very good load balance but high communication overhead. Another extreme is *static* task partitioning with maximal task sizes of $l_i = \lceil N/k \rceil$ that leads to very low communication overhead but potentially poor load balance.

Self-Scheduling Schemes: Additionally, we apply the Factoring[1] self-scheduling algorithm [18] from the area of HPC, as a simple yet very effective scheme. The basic idea is to use waves of exponentially decaying task sizes in order to achieve low communication overhead via large tasks at the beginning but good load balance via few small tasks at the end. *Factoring* computes the task size l_i for the next wave of k tasks, based on remaining iterations R_i, as follows:

$$\begin{aligned} R_0 &= N, \\ R_{i+1} &= R_i - k \cdot l_i, \end{aligned} \qquad l_i = \left\lceil \frac{R_i}{x_i \cdot k} \right\rceil = \left\lceil \left(\frac{1}{x_i}\right)^{i+1} \frac{N}{k} \right\rceil, \quad (1)$$

with $x_i = 2$ as suggested for unknown variability [18]. As an example, $N = 101$ and $k = 4$ gives us a sequence of $(13, 13, 13, 13, 7, 7, 7, 7, 3, 3, 3, 3, 2, 2, 2, 2, 1)$ iterations. For specific scenarios, we slightly extended this to *constrained* C$^-$/C$^+$Factoring that additionally imposes either a minimum constraint $l'_i = \max(l_i, cmin)$ (e.g., for reduced communication overhead) or a maximum constraint $l'_i = \min(l_i, cmax)$ (e.g., for upper-bounded memory usage). Regarding communication overhead it is noteworthy that $|\mathcal{W}|$ increases only logarithmically in N with $\mathcal{O}(k \log_x N/k)$.

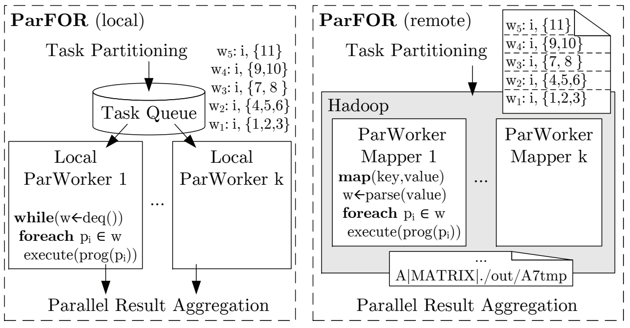
Task Encoding: For task communication, we use set and range tasks. Set tasks contain one or many values of the index variable, while range tasks encode a sequence of values via a (from, to, increment)-triple for compression if $l_i > 3$.

3.2 Local Parallelism

The variety of `ParFOR` use cases and workloads led to one of our major design goals: *generality* in terms of task parallelism for arbitrary body programs and arbitrary data sizes. As a first runtime strategy, we now explain `LOCAL ParFOR` (`ParFOR-L`) execution. The basic concept is to exploit multi-core parallelism by executing tasks concurrently as local threads within the JVM of SystemML's control program. This enables parallel execution within a single node with very low overhead and its generality allows arbitrary instructions and nested parallelism in the `ParFOR` body.

Runtime Architecture Overview: Figure 3(a) shows the runtime architecture of `ParFOR-L`. First, we initialize k parallel workers, create a task queue, and start the workers as threads that continuously dequeue and execute tasks until no more tasks are available. Second, we do task partitioning and enqueue tasks to the task queue. Using streaming task creation allows us to upper-bound the memory consumption of this task queue. Third, we wait for finished `ParFOR` execution by joining all worker threads. Fourth, we aggregate all worker results. In the following, we describe selected details.

[1]Factoring differs from guided self-scheduling [27], as used in OpenMP, in executing waves of tasks with equal size, which is more robust and naturally fits the MR execution model.



(a) Local Parallelism (b) Remote Parallelism

Figure 3: Architecture `ParFOR` Execution Strategies.

Local Parallel Workers are continuously running threads that execute one task—and internally one iteration—at-a-time. Each worker gets a deep copy of the `ParFOR` body, i.e., program blocks and instructions with unique filenames, and a shallow copy of the symbol table in order to ensure isolation. Due to shared address space and copy-on-write semantics, a shallow copy is sufficient, i.e., we do not need to copy input matrices.

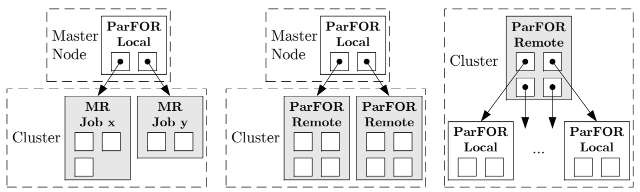
Task Scheduling assigns tasks to workers, where the single task queue is a self-scheduling approach. Since workers dequeue the next task whenever they finished a task, temporal gaps between task execution are very small. This also leads to a good load balance, which still depends on task partitioning because it determines the scheduling granularity. Finally, this approach reduces the communication overhead to a single synchronized dequeue operation per task.

Dynamic Recompilation in SystemML re-optimizes HOP DAGs during runtime according to the actual matrix characteristics. This is important for initial unknowns. `ParFOR-L` requires two extensions: First, we evenly divide the context memory budget among the k worker threads. Second, there is the danger of lock contention on the single HOP DAG. Hence, we create deep copies of relevant DAGs for each worker and thus enable concurrent recompilation.

3.3 Remote Parallelism

In order to complement the generality of local parallelism (`ParFOR-L`) with distributed in-memory computation, we provide a second runtime strategy: `REMOTE ParFOR` (`ParFOR-R`). The basic concept is to execute `ParFOR` itself as a single MR job and to execute its body as in-memory CP instructions, distributed on all nodes of the cluster. This ensures scalability for large or compute-intensive problems.

Runtime Architecture Overview: The runtime architecture of `ParFOR-R` is shown in Figure 3(b). First, we do task partitioning and serialize the task sequence into a task file on HDFS. Second, we export all dirty—i.e., updated, in-memory—input matrices to HDFS. Third, we serialize the `ParFOR` program body, i.e., program blocks, instructions, and referenced DML/external functions, a shallow copy of the symbol table, as well as internal configurations and store them in the MR job configuration. Fourth, we submit the



(a) L-MR (b) L-R-CP (c) R-L-CP

Figure 4: Examples Hybrid Parallelism.

MR job and wait for its completion. Result matrices of individual tasks are directly written to HDFS but (varname, filename)-tuples are collected in the job output. This ensures output flexibility and yet fault tolerance. Fifth, we aggregate results. We now describe again selected details.

MR Job Configuration: `ParFOR-R` is a map-only MR job whose input is the task file with one `ParFOR` task per line and we use the `NLineInputFormat` in order to initiate one map task per `ParFOR` task. The number of map tasks is therefore directly controlled via task partitioning, where k is equal to the number of map slots in the cluster.

Remote Parallel Workers behave like local workers, except for realizing the MR mapper interface. We initialize the worker by parsing the serialized program and creating program blocks, instructions, and a symbol table with unique file names. On each map, we parse the given task, execute the program for all task iterations, and write result variables to HDFS. In case of JVM reuse, we also reuse workers in order to exploit cached inputs and pre-aggregate results. Finally, we do not allow MR-job instructions (nested MR jobs) inside a remote worker because this incurs the danger of deadlocks if all map slots are occupied by `ParFOR`.

Task Scheduling is handed over to the Hadoop scheduler, which is important for several reasons. First, it provides global scheduling for (1) our task- and data-parallel jobs, as well as (2) other MR-based applications on a shared cluster. Second, we get MR functionality such as fault tolerance, data-locality, and an existing ecosystem. Finally, since Hadoop executes input splits by decreasing size, we ensure the `ParFOR` task order by padding with leading zeros.

3.4 Hybrid Parallelism

The *generality* of `ParFOR` allows us to combine parallel execution models as needed. We now exemplify hybrid parallelization strategies, where *hybrid* refers to combing (1) task and data parallelism, (2) in-memory and MR computation, as well as (3) multi-core and cluster parallelism. Finally, hybrid strategies give us great flexibility of creating efficient execution plans for complex ML algorithms.

Parallel MR Jobs (Figure 4(a)): If the `ParFOR` body contains operations on large data, we cannot run in-memory operations via `ParFOR-R`. However, `ParFOR-L` exploits multi-core parallelism for CP and MR-job instructions, and hence can run parallel MR jobs. This is beneficial for latency hiding and full resource exploitation. For instance, consider our running example with a $10^9 \times 5$ matrix $\mathbf{D}$, i.e., 10 pairs of $2 \cdot 8\,\text{GB}$ each. We would run two nested `ParFOR-L` and thus, MR jobs for indexing, covariance and central moment of all pairs in parallel. The best MR job configuration (e.g., number of reducers) depends on the `ParFOR` degree of parallelism k and we might get piggybacking potential across iterations.

Mixed Nested Parallelism (Figures 4(b)/4(c)): In case of nested `ParFOR`—as used in our running example—where only the outer contains an MR-job instruction, we can use `ParFOR-R` for the inner. This leads to parallel `ParFOR` MR jobs, in parallel to the MR jobs from the outer. If there are only CP instructions, we can use a `ParFOR-R` for the outer. Within those map tasks, we can additionally use multi-threaded `ParFOR-L` for the inner to exploit all resources.

3.5 Parallel Result Aggregation

After local or remote execution, we automatically consolidate all worker results, which is crucial for usability and performance. There are two important observations from dependency analysis. First, result variables are the dependency candidates $\mathcal{C}$, i.e., updated, non-local matrices. Second, independence implies that worker results are disjoint. Conceptually, we distinguish two scenarios, in both of which the original result matrix $\mathbf{R}$ still exists due to copy-on-write. First, if $\mathbf{R}$ is empty, we simply need to copy all non-zero values from all workers into the final result. An instance of this scenario is our running example. Second, if $\mathbf{R}$ is non-empty, we need to copy all (zero and non-zero) values that *differ* from the original ones. An example is distributed matrix factorization, where we iteratively modify subblocks in parallel. We call those two cases *with* and *without compare*.

We support three strategies: (1) local in-memory, (2) local file-based, and (3) parallel remote result aggregation. *Local in-memory* pins $\mathbf{R}$ into memory, creates the compare matrix if necessary, and merges all worker results one-at-a-time. *Local file-based* uses a virtual staging file of indexed blocks, which can be viewed as a radix sort of blocks. It maps worker result and compare blocks, and merges one result block at-a-time. *Parallel remote* uses a dedicated MR job whose inputs are the worker results and if necessary the compare matrix. Mappers then tag blocks as data or compare, while reducers get the compare block and then merge one block at-a-time.

3.6 Runtime Optimizations

With the aim of efficiency and scalability for workloads with large input matrices, we introduce additional dedicated runtime optimizations. At a high level, this includes novel techniques for (1) access-aware data partitioning, which applies to *disjoint/overlapping* access and (2) access-aware data locality, which applies to *disjoint/overlapping/all* access. Additionally, we also change the replication factor for *overlapping/all* access, but this is a pure optimizer decision. Access-awareness over the entire `ParFOR` body program is crucial for cost estimation and correct plan generation.

Data Partitioning: For large matrices, i.e., if the memory consumption of an operation exceeds the memory budget, we execute that operation as an MR instruction in order to ensure robustness. This has serious implications because we cannot execute `ParFOR-R` and we execute at least one MR job per iteration and thus, potentially many MR jobs that repeatedly scan the entire data. A key observation is that we often have very large input matrices but individual `ParFOR` iterations work only on rows, columns or blocks of moderate size. In those scenarios, the MR job for repeated indexed access is one of the most expensive operations. Our basic idea is (1) to transparently partition the input matrix according to the access pattern and (2) to rewrite the body to direct indexed access of partitions. We apply data partitioning only for read-only matrices and pure row- or column-wise access pattern. This ensures that no operation other than indexed access is affected. For each input matrix $\mathbf{D}$, we recursively analyze all accesses in the `ParFOR`

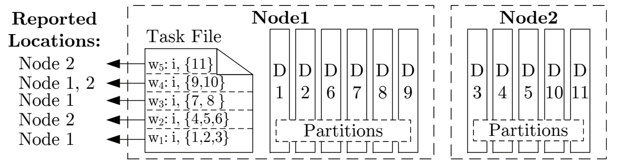


Figure 5: Example Locality Reporting for `X=D[,i]`.

body; if there is a common row- or column-wise pattern, this becomes the partitioning scheme. We then accordingly partition **D** into directly accessible files and recompile indexed accesses with forced execution type depending on the partition size. We provide two realizations: (1) *local file-based*, and (2) *parallel remote* partitioning both of which create one `SequenceFile` per partition. *Local file-based* is a two-phase out-of-core algorithm that reads the input, appends blocks to a partitioned staging file, and finally creates a partitioned matrix. *Parallel remote* is a dedicated MR job, where mappers do the partitioning on a block level and reducers write partitions. For high performance of partitioning and read, we also support block-wise partitioning (groups of rows or columns) with a block size close to the HDFS block size.

Data Locality: Since `ParFOR-R` uses a task file as the input, Hadoop cannot co-locate relevant map tasks to the input matrices or partitions. This leads to unnecessary data transfer because—especially on large clusters—data-local access becomes unlikely. Our basic idea is to explicitly provide location information of relevant matrices and partitions per logical task in order to enable data locality. We do this for the largest input matrix. We use a dedicated input format (specialized `NLineInputFormat`) and input split (specialized `FileSplit`). Whenever splits are requested from this input format, we analyze the matrix partitioning information and create our wrapper split around each original file split. Instead of reporting the location of the small task file splits, we (1) parse our logical task (one per split), (2) get the locations of all related partition files, (3) count frequencies, and finally (4) report the top-k frequent nodes as locations. Since, Hadoop treats all reported locations equally, we report only the hosts with top-1 frequency, as shown in Figure 5 for one index access of our running example. For range tasks, i.e., (from, to, increment)-triples, we heuristically analyze only locations of the first and last iteration because locality is examined serially before job submission.

4. OPTIMIZATION FRAMEWORK

Hybrid parallelization strategies give us great opportunities. However, finding the optimal parallel execution plan is challenging because local decisions affect each other due to shared resources and data-flow properties, which span a huge search space. In the following, we present a tailor-made optimization framework including (1) the problem formulation, (2) a cost model and statistics estimation, as well as (3) plan rewrites and an optimization algorithm. The generality of this framework allows to reason about hybrid task and data parallelism of arbitrary complex ML programs.

4.1 Problem Formulation

During initial compilation, important matrix characteristics and most importantly the `ParFOR` problem size N might be unknown. This led to the major design decision to apply `ParFOR` optimization as a second optimization phase—as done in parallel DBMSs such as XPRS [15]—at runtime for each top-level `ParFOR`. As a foundation, we now define the plan representation and related optimization problem.

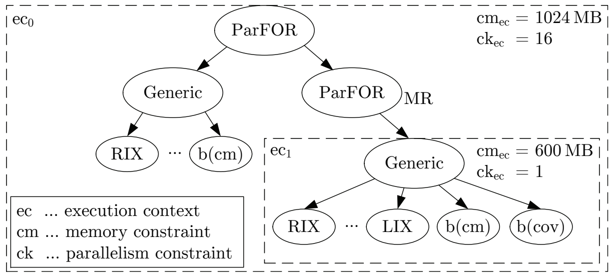


Figure 6: Plan Tree of Running Example.

DEFINITION 2. ***(Plan Tree)*** *P is a tree of nodes $n \in \mathcal{N}_P$ with height h, modeling* `ParFOR` *and its body prog. Inner nodes are program blocks and refer to an unordered set of child nodes $c(n)$, where edges represent containment relationships. Leaf nodes are operations. Nodes have a node type nt, an execution type et, a degree of parallelism k, and specific attributes $\mathcal{A}$. P spans a non-empty set of execution contexts $\mathcal{EC}_P$, where the root node $r(P)$ and specific et define a context with memory cm_{ec} and parallelism ck_{ec} constraints. Shared resources are global constraints.*

Example Plan Tree: Figure 6 shows the plan tree P of our running example. Inner nodes refer to `ParFOR` and `generic` program blocks; leaf nodes refer to operations (HOPs in this case). The root node defines the context of the master process with its constraints (e.g., the max JVM size). Since the nested `ParFOR` has $et = \text{MR}$, there is a second context of the map task process. Note there is a mapping from nodes in P to HOP DAGs and runtime plans. Based on the plan tree (Def. 2) and our overall objective (Def. 1), we now can define the plan tree optimization problem.

DEFINITION 3. ***(Plan Tree Optimization Problem)*** *Given an initial plan tree P, which is assumed to be the optimal sequential plan per program block, transform P into a semantically equivalent plan tree P' that is optimal w.r.t.*

$$\begin{aligned} \phi_2: \quad & \min \quad \hat{T}(r(P)) \\ & s.t. \quad \forall ec \in \mathcal{EC}_P : \hat{M}(r(ec)) \le cm_{ec} \ \wedge \\ & \qquad\qquad K(r(ec)) \le ck_{ec}. \end{aligned} \tag{2}$$

Thus, the goal is to minimize the execution time of the plan tree's root node $\hat{T}(r(P))$ under the hard constraints of maximum total memory consumption $\hat{M}(r(ec))$ and maximum total parallelism $K(r(ec))$ per execution context ec. Valid transformations are node operator selection (et), node configuration changes (k, $\mathcal{A}$), and structural changes of P.

In contrast to traditional query optimization, P covers (1) a complex ML program with control flow, linear algebra operators, task and data parallelism, which require dedicated rewrites and cost modeling, and (2) hard constraints, which require dedicated search strategies and cost estimation.

Complexity: The plan tree optimization problem is a *multiple knapsack problem with multiple constraints*, which is known to be $\mathcal{NP}$-hard. Its specific properties are a *variable hierarchy* of items and knapsacks, as well as multiple *variable* capacity constraints. In detail, n is an item and $\hat{T}(n)$ is the item value. Each context ec defines a knapsack with constraints cm_{ec} and ck_{ec}. P with $\mathcal{EC}_P$ defines the item and knapsack hierarchies, where multiple knapsacks share common constraints (e.g., cluster parallelism).

4.2 Cost Model

As a fundamental precondition for optimization, we need a cost model and accurate estimates. According to objective ϕ_2, there are different requirements on those estimates. We are interested in the expected execution time since this is our soft constraint but we need *worst-case* estimates for memory and parallelism in order to guarantee those hard constraints. This is important for preventing out-of-memory situations. Furthermore, we need an *analytical* cost model that allows us to cost arbitrary plan alternatives. We now describe details of (1) estimating memory and execution time for leaf nodes, and (2) aggregating statistics over plan trees.

4.2.1 Worst-Case Memory Estimates

Memory estimation for leaf nodes of a plan tree works on HOP DAGs. We are interested in estimates for CP operations but it also applies to block computations in MR. For each DAG, we use a bottom-up approach of recursively propagating matrix characteristics and estimating memory.

Matrix Characteristics: Important matrix characteristics for memory estimates are the matrix dimensions d_1, d_2 and the matrix sparsity d_s. For worst-case memory estimates, we also need worst-case estimates of those characteristics. Fortunately, many operators of linear algebra (e.g., matrix multiplication) allow to exactly infer their output dimensions. Inferring the sparsity is more difficult due to potential skew but there are, for example, sparsity preserving operations like $s \cdot \mathbf{X}$ ($s \notin \{0, \text{NaN}, \infty\}$). For unknown characteristics, we assume $d_1 = \infty$, $d_2 = \infty$ and $d_s = 1$.

Memory Estimates: Based on worst-case matrix characteristics, which have been propagated through the DAG, we estimate per operation its output memory $\hat{M}(out(n))$ and operation memory $\hat{M}(n)$ as follows (simplified):

$$\hat{M}(out(n)) = \begin{cases} d_1(116\,\text{B} + 12\,\text{B} \cdot d_2 d_s) & \text{sparse} \\ 8\,\text{B} \cdot d_1 d_2 & \text{dense} \end{cases}$$
$$\hat{M}(n) = \sum_{\forall c_i \in in(n)} \hat{M}(out(c_i)) + \hat{M}(n,k) + \hat{M}(out(n)), \tag{3}$$

where dense matrices are double arrays and sparse matrices use compressed rows of (column index, value)-pairs. This estimate reflects our runtime model of CP instructions that pin all inputs $in(n)$ and the output $out(n)$ into memory. Hence, the memory estimate is the sum of input, intermediate (depending on internal parallelism k), and output sizes.

Example Memory Estimates: Figure 7 depicts part of the HOP DAG of our running example's inner `ParFOR`. Assume a $10^6 \times 10$ input matrix $\mathbf{D}$ with sparsity $d_s = 1$. Accordingly, we estimate the output size of right indexing (`RIX`) as 8 MB and the total operation memory as 88 MB.

The worst-case memory estimate of a leaf node of P is then defined as the operation memory estimate $\hat{M}(n)$ of its mapped HOP if $et = \text{CP}$ and as a constant M^C if $et = \text{MR}$ because then it is executed in a separate context.

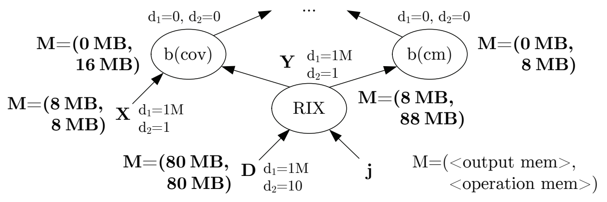
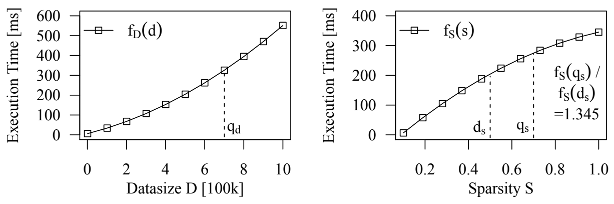


Figure 7: Example Memory Estimates.



(a) Cost Function Datasize (b) Cost Function Sparsity

Figure 8: Example Time Estimates `t(X)%*%X`.

4.2.2 Time Estimates

Estimating time for leaf nodes of a plan tree is more challenging than memory. For accurate estimates, we need to take runtime properties of operators into account. Our basic idea therefore relies on *offline performance profiling* of *runtime instructions*, done once for a cluster configuration.

Performance profiling measures T of relevant instructions Op for variables $\mathcal{V}$, varying one $v \in \mathcal{V}$ at-a-time. Different execution types and matrix representations are modeled as different instructions. We then create polynomial regression models as cost functions $C_{T,Op}(v)$ for each (v, Op)-combination. The profile is the set of $C_{T,Op}(v)$ for all $v \in \mathcal{V}$.

Cost Function Scaling then estimates $\hat{T}$ via this cost profile as follows for a given request $\mathcal{Q}$ with $\forall v \in \mathcal{V} : \exists p(v) \in \mathcal{Q}$, where $f_x(q_x)$ is a shorthand for $C_{T,Op}(x = q_x)$:

$$\hat{T}(\mathcal{Q}, \mathcal{A}) = f_d(q_d, \mathcal{A}) \cdot \prod_{\forall x \in (\mathcal{Q} - d)} \frac{f_x(q_x, \mathcal{A})}{f_x(d_x, \mathcal{A})} \cdot corr(\mathcal{Q}). \tag{4}$$

Scaling one-dimensional cost functions makes a fundamental independence assumption, which is important for efficient profiling but can lead to low accuracy. We therefore use correction terms, based on ratios of number of floating point operations, e.g., for matrix shape adjustments. This correction is crucial for high accuracy due to a shape-dependent asymptotic behavior. For example, consider a matrix multiplication $\mathbf{AB}$, where each matrix has 10^6 cells, i.e., 8 MB. Multiplying two $1{,}000 \times 1{,}000$ matrices requires 2 GFlop, while a dot product of $10^6 \times 1$ vectors requires only 2 MFlop, i.e., a relative difference of 10^3. To summarize, scaled cost functions allow us to accurately estimate time, even for different behavior of dense/sparse operations.

Example Time Estimates: Assume a query $\mathcal{Q}$: $q_d = 700{,}000$, $q_{d1} = 1{,}000$, $q_{d2} = 700$, and $q_s = 0.7$ for CP, dense, transpose-self matrix multiplication $\mathbf{X}^\top\mathbf{X}$. Further, assume cost functions for datasize $f_D(d)$ and sparsity $f_S(s)$, created with squared matrices and defaults $d_d = 500{,}000$ and $d_s = 0.5$. We pick f_D as the leading dimension and get $f_D(q_d) = 325$ ms. Then, we scale it to $\hat{T}(q_d, q_s) = \hat{T}(q_d) \cdot f_S(q_s)/f_S(d_s) = 438$ ms as shown in Figure 8. Last, we do the correction $\hat{T} = \hat{T}(q_d, q_s) \cdot corr(\mathcal{Q})$ and get $\hat{T} = 366$ ms.

Finally, we assign the time estimates from mapped HOPs and instructions to plan tree leaf nodes again.

4.2.3 Plan Tree Aggregate Statistics

Memory and time estimates for arbitrary complex programs are then aggregates of leaf node estimates.

Memory Estimates: The worst-case estimate of memory consumption for a `ParFOR` node is computed with

$$\hat{M}(n) = \begin{cases} k \cdot \max\limits_{\forall c_i \in c(n)} \hat{M}(c_i) & et = \text{CP} \\ M^C & et = \text{MR}, \end{cases} \tag{5}$$

as the number of workers times the most-memory consuming operation since those operations are executed sequentially.

The memory estimate for all other inner-nodes (`for`, `while`, `if`, `func`, and `generic`) is $\hat{M}(n) = \max_{\forall c_i \in c(n)} \hat{M}(c_i)$. One challenge is to incorporate shared reads into memory estimates in order to prevent large overestimation. This is done by splitting $\hat{M}$ into shared $\hat{M}^+$ and non-shared $\hat{M}^-$ parts and scaling $\hat{M}^+$ by the number of consumers.

Time Estimates: Average-case time estimates are the sum of child node estimates due to sequential execution:

$$\hat{T}(n) = w_n \sum_{\forall c_i \in c(n)} \hat{T}(c_i), \;\; w_n = \begin{cases} \lceil \hat{N}/k \rceil & \texttt{parfor} \\ \hat{N} & \texttt{for,while} \\ 1/|c(n)| & \texttt{if} \\ 1 & \text{otherwise.} \end{cases} \quad (6)$$

Since, we cannot determine $\hat{N}$ for `while` and unknown `for`/`parfor`, we estimate it as a constant $\hat{N} = N^c$ there. This reflects at least that the body is likely executed multiple times. Furthermore, the time estimate of an `if` is a weighted sum because only one branch is executed at-a-time.

Parallelism: Finally, we also aggregate total parallelism with $K(n) = k \cdot \max_{\forall c_i \in c(n)} K(c_i)$. For excluding remote parallelism we use $K(n) = 1$, if $et =$ MR.

4.3 Optimization Algorithm

We now discuss an algorithm for finding optimal parallel execution plans. Due to the huge search space, we support a spectrum of optimizers with different complexity. Each optimizer is characterized by: (1) the used cost model, (2) the rewrites that define the search space, and (3) the search strategy. Here, we describe our default heuristic optimizer.

Heuristic Optimizer Overview: Our *heuristic* optimizer uses (1) a *time- and memory-based* cost model without shared reads, (2) heuristic high-impact rewrites, and (3) a transformation-based search strategy with global optimization scope. The time-based cost model enables accurate estimates but requires a pre-created profile. If no profile exists, we use a *memory-based* cost model and—instead of time estimates—additional heuristics that local, in-memory computations require less time than their MR alternatives.

Rewrites: The search space is defined by a variety of `ParFOR`-specific heuristic rewrites. This comprises (1) rewrites regarding `ParFOR` parallelization strategies and (2) rewrites exploiting repeated, parallel iteration execution. First, examples for `ParFOR`-centric rewrites are *operator selection* such as selecting the `ParFOR` execution type *et* (CP/MR, i.e., `ParFOR-L`/`ParFOR-R`), task partitioning and result aggregation methods. Example *configuration changes* include choosing the degree of parallelism k and task sizes. *Structural plan changes* are, for example, artificial nested `ParFOR` for multi-threaded map tasks, unfolding `ParFOR` in recursive functions, and changing `ParFOR` to `for`. Second, examples for iteration-aware rewrites include again *operator selection* like data partitioning, where we choose the partitioning scheme, partitioning method, and execution type *et* of left and right indexing. Example *configuration changes* are choosing matrices for co-location, and changing the partition replication factor r. Most of these rewrites need to take the entire plan tree P into account.

Search Strategy: We use a transformation-based, top-down search strategy that transforms P and its mapped program into P'. This follows the fundamental heuristic to apply available parallelism as high as possible in the plan tree to cover more operations and reduce synchronization. Our simple yet very effective approach uses a well-defined rewrite order. First, we apply data-flow rewrites that change the memory estimates of P. This includes data and result partitioning because we potentially recompile related indexed reads/writes to in-memory operations. Second, we recursively decide—starting at the root of P—on `ParFOR` execution type and degree of parallelism. Based on memory constraints and estimates, we can directly compute the maximum parallelism to apply per level. Third, for all subtrees rooted by `ParFOR`, we apply execution-type-specific rewrites. For `ParFOR-L` this includes task partitioner and recompilation budget, while for `ParFOR-R` this includes data co-location, replication factors, nested `ParFOR`, and task partitioner. Fourth, and finally, we decide on result merge strategies, handle recursive functions, and recompile unnecessary `ParFOR` to `for`. The majority of rewrites has a complexity of $\mathcal{O}(|\mathcal{N}_P|)$ with exceptions of up to $\mathcal{O}(|\mathcal{N}_P|^2)$. This strategy finds the optimal plan according to the heuristically restricted search space but guarantees all constraints of ϕ_2.

5. EXPERIMENTAL EVALUATION

The aim of our evaluation is to study `ParFOR` parallelization strategies and its optimization for a variety of use cases. To summarize, the major findings and insights are:

Hybrid Parallelization: ML use cases exhibit diverse workload characteristics. Complementary parallelization strategies and optimization are key to achieve high performance. The R comparisons confirmed this since each use case required a different hand-chosen strategy. In contrast, our optimizer generated good plans with negligible overhead.

High-Impact Rewrites: Depending on the use case, different rewrites matter the most. For large inputs with indexed access it is data partitioning, locality and replication; for large compute it is `ParFOR-R`, i.e., distributed in-memory operations; for many iterations it is task partitioning; and for large outputs it is result partitioning and merging.

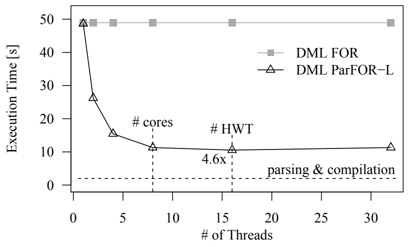
R and Spark Comparison: Our optimizer achieved performance comparable to pure in-memory computations (local and distributed) on small problems, significant improvements on larger problems, and very good scalability. Furthermore, both R and Spark required careful prevention of out-of-memory situations (since they are pure runtimes), while our optimizer guarantees memory constraints.

5.1 Experimental Setting

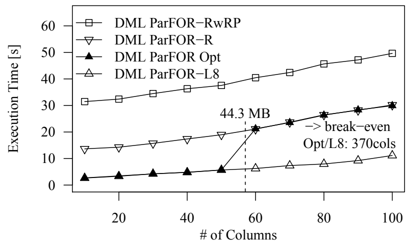
Setup: We ran our experiments on a 5 node cluster, i.e., 3 nodes of 2x4 X5550 @ 2.67GHz and 2 nodes of 2x4 X5560 @ 2.80GHz; each node with hyper-threading enabled, 64 GB RAM, 1.5 TB storage (6 disks in RAID-0), 1 Gb Ethernet, and SUSE Linux Enterprise Server 11 64bit. For reproducible results, all experiments measured *total* execution times from outside SystemML (as of 07/2013), which conservatively assumes a shared cluster and uses the heuristic `ParFOR` optimizer without pre-created profile. The default number of reducers per MR job was set to 10. We used IBM Hadoop Cluster 1.1.1 and IBM JDK 1.6.0 64bit. Our Hadoop cluster[2] was configured with 80/40 map/reduce slots, without JVM reuse, and an HDFS block size of 128 MB. All experiments used initial/max JVM sizes of 1 GB for the master and map/reduce tasks, and a memory budget ratio of 0.7.

Data and Use Cases: For investigating different data characteristics, we use synthetic data, created with

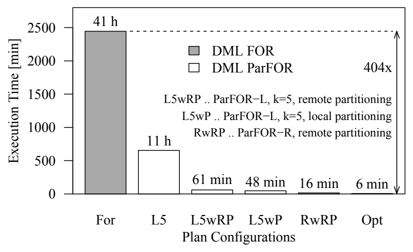
[2]We measured the singlenode HDFS throughput with DFS-IO (1 file, 1 GB) as up to 456 MB/s read and 107 MB/s write.



(a) Scenario S: Local Parallelism



(b) Scenario S: Plan Comparison



(c) Scenario M: Plan Comparison

Scenario	DML (Opt)	R (1x1)	R `doMC` (1x8)	R `doSNOW` (5x1/5x8)	Spark Data-Par. mem/mem+disk	Spark Hybrid mem/mem+disk	Spark Task-Par. task/broadcast/(partition)
XS ($10^3 \times 100$)	3	3	**3**	5 / 19	176 / 180	86 / 89	7 / 6 / (11)
S ($10^5 \times 100$)	30	21	**6**	13 / 45	319 / 323	249 / 263	– / 13 / (20)
M ($10^7 \times 100$)	**363**	2,109	751*	838 / 870*	7,875 / 8,256	11,140 / 11,458	– / – / (776)
L ($10^7 \times 1{,}000$)	**17,321**	–	–	– / –	7.4E7** / 5.5E6**	7.0E7** / 5.7E6**	– / – / (–)

(d) R and Spark Comparison [s] (*reduced degree of local parallelism $k = 3$, **estimate based on 1% sample of iterations)

Figure 9: Results of Pairwise Correlation (Descriptive Statistics).

algorithm-specific DML data generators. Furthermore, we use a spectrum of three use cases ranging from data- to compute-intensive. For each use case, we investigate scenarios of small (S), medium (M), and large (L) input data. All use cases are based on real-world problems, where experiments on the real data showed similar results.

Baseline Comparisons: The baselines are (1) SystemML's serial `FOR`, (2) R 2.15.1 64bit [9] (unlimited memory), incl. the parallel R packages `doMC` (multi-core, 1 node $\times$ 8 cores) and `doSNOW` (cluster, type socket, 5 nodes $\times$ 1/8 cores), as well as (3) Spark 0.8.0 [32] (5×16 workers, 16 GB memory per node, 10 reducers, standalone on top of our HDFS). For a fair comparison, both SystemML and R use (1) binary inputs/outputs, and (2) equivalent script-level algorithms, while for Spark, we ported our runtime operators and hand-tuned alternative runtime plans. To the best of our knowledge, there is no publicly available MR-based ML system on an abstraction level comparable to SystemML.

5.2 Use Case 1: Descriptive Statistics

Our first use case is the data-intensive pairwise correlation (Corr) from our running example (Ex. 2), representing more complex real-world use cases of bivariate statistics from finance and log analysis. Its characteristics are (1) large, dense, partitionable input, (2) small output, (3) few iterations, and (4) a simple algorithm but nested `ParFOR`.

Scenario S: The small scenario uses a dense $10^5 \times 100$ input matrix, i.e., 80 MB in dense binary format and 4,950 pairs. Figure 9(a) compares `FOR` and `ParFOR-L` for increasing numbers of threads k. For $k = 1$, we see that the `ParFOR` overhead is negligible. We also observe good speedups with more threads up to 4.6x at $k = 16$. The best speedup is below 8x because this includes parsing/compilation (2 s), read/write, and JIT compile. Figure 9(b) compares the plan alternatives `ParFOR-L` with $k = 8$, `ParFOR-R` w/ and w/o remote partitioning (2 and 1 MR jobs), and the choice of our optimizer (`Opt`) for increasing numbers of columns n. The time differences are mainly due to MR job latency as well as more I/O and JIT compile. Our optimizer picks `ParFOR-L` up to $n = 56$ and then switches to `ParFOR-R` because it tries to exploit $k = 16$ threads and hence, estimates the memory as 730 MB, i.e., greater than the budget of 717 MB.

Scenario M: Figure 9(c) shows the results of the medium scenario, where we increased the input size to $10^7 \times 100$, i.e., 8 GB. Since this exceeds our memory budget, right indexing is initially compiled to MR. We again compare several plan alternatives, where we set $k = 5$ for `ParFOR-L` to meet the JVM constraints. First, we observe a large execution time of 41 h for serial `FOR` because it uses 5,049 MR jobs for column indexing. Second, although we already exploit 64 of 80 map slots for a single MR job on 8 GB, `ParFOR-L` improves the runtime by almost 4x due to latency hiding, full utilization of map/reduce slots, and parallel CP instructions. Third, partitioning (remote and local) has a huge impact on `ParFOR-L` because it replaces the 5,049 indexing jobs with one partitioning operation. Local partitioning shows better performance due to partition locality at the master node. Fourth, partitioning is also an enabler for `ParFOR-R` that led to additional improvements. Finally, our optimizer produces the best plan—via additional rewrites (e.g., replication) as detailed in Subsection 5.4—for a speedup of 404x.

R Comparison: Figure 9(d) shows the R results. The R scripts implement this use case via `moment` and `cov`, while scripts with the dedicated `cor` were only up to 9% faster. For `doMC` and `doSNOW` we returned index-value list results due to their output restrictions. For scenario S, R provides very good performance, where `doMC` performed best because the broadcast overhead of `doSNOW` was not amortized. Although our optimizer switched too early to `ParFOR-R` and despite our numerical stable operators [31], our runtime is comparable. For an even smaller scenario XS, there is essentially no difference between DML (Opt) and `doMC`. On scenario M, serial R and `doMC` scale linearly, while `doSNOW` scales even better because its overheads have less impact. Note that we had to reduce the degree of parallelism for `doMC` and `doSNOW` to 3 in order to prevent swapping because they create multiple copies of the entire dataset. Although, `doSNOW` shows the better relative behavior, `doMC` still performs best of R. However, our optimized plan was 2x faster. We also conducted a large experiment with $10^7 \times 1{,}000$ input (Scenario L), which means 80 GB and 499,500 pairs. `ParFOR-R` alone read 40 TB from HDFS. However, we see linear scalability and an aggregated throughput of $2.3\,\mathrm{GB/s}$. This scenario cannot be executed in R because **D** does not fit in-memory and exceeds R's constraint of 2^{31} elements per matrix.

Spark Comparison: Figure 9(d) also shows the Spark results. We implemented three categories of plans via Sparks Java API (`JavaPairRDD`) and SystemML's binary format/operations, w/o buffer pool: (1) *data-parallel* (parallel indexing, cm, cov), (2) *hybrid* (parallel indexing, local cm, cov), and (3) *task-parallel* (parallel outer loop). For data-parallel and hybrid, we investigate different RDD storage levels (memory-only, memory-and-disk) for **D** (and **X**). For task-parallel plans, we investigate different input transfers (task serialization, broadcast variables, and HDFS partitions via our data partitioning, replication 1). First, Spark shows impressive performance for data-parallel and hybrid plans. Compare Scenario M, Hybrid against SystemML `FOR` from Figure 9(c) (equivalent plans): the 13x improvement is reasoned by distributed in-memory data and fast job/task scheduling. Second, despite fast jobs, data-parallel/hybrid plans do not perform well for this task-parallel use case due to many jobs (15,048/10,098 for scenarios XS, S, M). Third, task-parallel plans with Spark's mechanisms quickly run out of memory. With our partitioning, Spark performs very well on Scenario M, which is comparable to `ParFOR-R` w/ remote partitioning (RwRP, Figure 9(c)). On Scenario L, even this plan fails because column partitioning runs out of memory at `groupByKey()`. However, conceptually all presented techniques could be transferred to Spark as well.

Larger Clusters: We also conducted experiments on two larger clusters of (a) 1+15 nodes (2x4 E5440 @ 2.83GHz, 4 disks), 120/60 map/reduce and (b) 1+18 nodes (1x6 E5-2430 @ 2.20GHz, 11 disks), 216/108 map/reduce. We see consistent results for Scenario L of 13,009 s and 5,581 s. Compared to the result of 17,321 s with 80 mappers, this means linear scaling with increasing degree of parallelism. This very good scalability allows us to effectively address larger problems with additional hardware resources.

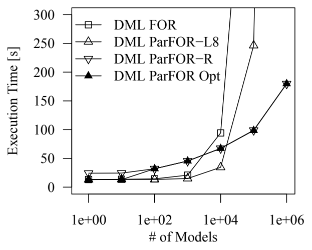
5.3 Use Case 2: Meta Learning

As a second use case, we use compute-intensive meta learning approaches: *linear regression feature subsampling* (LinReg) and *logistic regression parameter search* (Logistic).

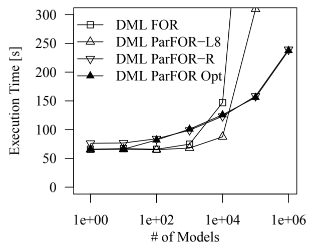
5.3.1 Linear Regression, Feature Subsampling

This linear regression example is based on a real use case from an insurance company. We have an $m \times n$ input matrix $\mathbf{X}$, a vector $\mathbf{y}$, and we want to compute N ordinary least square models of z randomly chosen features. We compute the normal equations once with $\mathbf{A} = \mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I_n}$ and $\mathbf{b} = \mathbf{X}^\top \mathbf{y}$, where $\lambda \mathbf{I_n}$ is the regularization. Each `ParFOR` iteration then picks $z = 15$ random features out of $[1, n]$, projects $\mathbf{A}$ and $\mathbf{b}$, solves the system of equations with an external function calling JLAPACK, and writes the vector of coefficients to a result variable. This use case is characterized as (1) small, dense `ParFOR` input, (2) moderate-size output, (3) many iterations, and (4) a simple algorithm but potentially large matrix multiplications before `ParFOR`.

Scenario S: The small scenario has a dense $10^4 \times 10^3$ input matrix $\mathbf{X}$, i.e., 80 MB. Figure 10(a) shows the runtime with log-scaled increasing N. For one model, we see the time of initial matrix multiplications. Serial `FOR` and `ParFOR-L` perform well up to 1,000 models, while `ParFOR-R` exhibits additional job setup latency. Our optimizer switches too early—at 100 models—from `ParFOR-L` to `ParFOR-R`. One key observation is, however, that as N further increases the runtime for serial increases dramatically. The reason is copy-on-write for left indexing which copies $zN/2$ non-zero values per iteration and dominates costs for large N. This effect occurs time-delayed for `ParFOR-L` due to a natural result partitioning across k workers. `ParFOR-R` and our optimized



(a) Scenario S

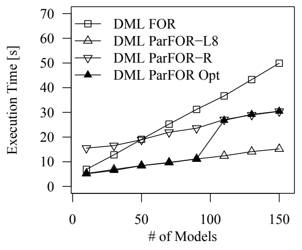


(b) Scenario M

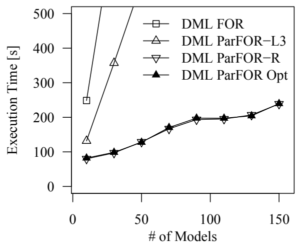
Scenario	DML	R	R `doMC`	R `doSNOW`
S ($10^4 \times 10^3$)	**180**	239	1,872	4,616 / 2,599
M ($10^6 \times 10^3$)	**238**	1,631	3,034	5,737 / 3,739
L ($10^7 \times 10^3$)	**574**	–	–	– / –

(c) R Comparison [s]

Figure 10: Results of Linear Regression.



(a) Scenario S



(b) Scenario M

Scenario	DML	R	R `doMC`	R `doSNOW`
S ($10^4 \times 10^3$)	**31**	370	60	82 / 35
M ($10^6 \times 10^3$)	**239**	19,869	4,621	4,041 / 1,097
L ($10^6 \times 10^5$)	**619**	49,860	12,355	10,492 / 2,550

(c) R Comparison [s]

Figure 11: Results of Logistic Regression.

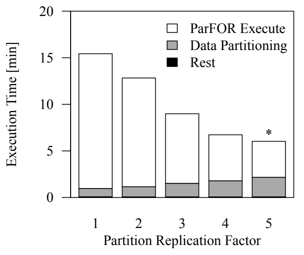
plan do not have this problem due to result partitioning *per task*. For even larger N, our optimizer would control result partitioning via C$^+$Factoring according to the result size.

Scenario M: The medium scenario increases the data size to $10^6 \times 10^3$, i.e., 8 GB. The behavior (see Figure 10(b)) is the same as in Scenario S, except for a larger offset for initial matrix multiplications. This makes a case for hybrid runtime plans where we exploit data parallelism for matrix multiplications and task parallelism for model training.

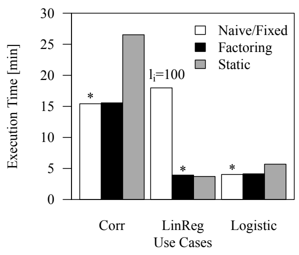
R Comparison: Figure 10(c) shows the R baselines, implemented via matrix multiplications and `solve` that also calls LAPACK. For Scenario S, DML is almost 1.5x faster than R serial, while interestingly, both parallel R strategies are significantly slower. Serial R uses in-place updates here, while we do copy-on-write. However, we gain from parallelism and result partitioning. Both `doMC` and `doSNOW` suffer from expensive result merge. We tuned the parameters of `doMC` and `doSNOW`; increasing `.maxcombine` from 100 to 500, gave a 2x improvement which supports the argument of result merge overhead. For Scenario M, DML is 7x faster than R due to data-parallel matrix multiplications. The large scenario increases the data size to $10^7 \times 10^3$, i.e., 80 GB, where we see very good scalability of data-parallel matrix multiplications. Again, R cannot execute Scenario L.

5.3.2 Logistic Regression, Parameter Search

As a more complex example—based on a real use case from an automotive manufacturer—we do parameter search on a trust region Newton method for logistic regression [22]. This algorithm exposes a regularization parameter C, which



(a) Replication Factor



(b) Task Partitioner

Figure 12: Selected Optimizer Decisions (*).

we investigate via a `ParFOR` parameter grid search over $\mathbf{C} = \theta_1((\mathbf{y} = 1) + \theta_2(\mathbf{y} = -1))$ with $\theta_1 \in \{1, .1, .01, .001, .0001\}$ and $\theta_2 \in [5/N, 1]$ for $N \geq 5$. This use case is characterized as (1) moderate-size, sparse input, (2) small output, (3) few iterations, and (4) a complex iterative algorithm.

Scenario S: The inputs of the small scenario are a sparse $10^4 \times 10^3$ matrix $\mathbf{X}$ ($d_s = 0.01$) and a dense 10^4 vector $\mathbf{y}$, i.e., 1.3 MB. We set outer/inner max iterations (model/step size convergence) to 1,000. Figure 11(a) shows the results for increasing number of models N, where the maximum number of observed outer iterations was 13. `ParFOR-L` shows a good speedup due to the compute-intensive use case, while `ParFOR-R` has again latency overhead but scales better for larger N. Our optimizer picks `ParFOR-L` up to $N = 90$ but switched too early (at $N = 100$) to the robust `ParFOR-R`.

Scenario M: The medium scenario increases the input size to $10^6 \times 10^3$, i.e., 130 MB. Figure 11(b) shows the results, where the maximum number of observed outer iterations was 18. The moderate speedup of `ParFOR-L` is reasoned by a reduced degree of parallelism of $k = 3$ to meet the JVM constraints. Due to the compute-intensive workload, `ParFOR-R` shows the best performance and scalability, where for $N \leq 80$, we do not fully exploit the cluster and are affected by load imbalance. Finally, our optimizer uses `ParFOR-R` right from $N = 10$ due to memory pressure.

R Comparison: Figure 11(c) shows the R baselines. Due to the compute-intensive workload our optimized plan is already 12x faster than serial R, 2x faster than R `doMC`, and level with `doSNOW`. For Scenario M, the behavior is similar but the relative speedup of DML and `doSNOW` increased due to amortized distribution costs. DML scales better and is now 4.6x faster than `doSNOW` due to a higher degree of parallelism and faster serial execution. Scenario L further investigates sparse inputs by increasing the data size to $10^6 \times 10^5$ but decreasing the sparsity to $d_s = 0.0001$, i.e., still 130 MB. The maximum number of observed iterations was 22. R was able to execute Scenario L due to sparse input. However, DML scales better than the R alternatives and R serial is already 81x slower which makes it infeasible.

5.4 Optimizer Deep-Dive

Finally, we take a closer look at selected optimizer decisions and the actual optimization overhead itself. The use cases are Corr, LinReg, and Logistic as introduced before.

Replication Factor: Setting the partition replication factor r had large impact for the Corr use case. Figure 12(a) shows the time breakdown for `ParFOR-R` on Scenario M with increasing r. Partitioning time increases sublinearly due to asynchronous replication but the execution time decreases significantly due to nested, i.e., the quadratic number of, partition reads. Note that together with data locality the trend is similar but damped. Hence, our optimizer set $r = 5$.

Task Partitioner: Task partitioning plays an important role for low communication overhead but good load balance. Figure 12(b) compares different task partitioner (Naïve/Fixed-Size, Factoring, Static) for `ParFOR-R` on Scenario M of all use cases. For few iterations (Corr and Logistic, 100/150) both Naïve and Factoring used minimal task sizes of $l_i = 1$ that led to very good load balance at low overhead and hence performed best. For many iterations (LinReg, 10^6), even Fixed-size with a task size of $l_i = 100$ (i.e., 10,000 tasks) led to large communication overhead, while Naïve was infeasible. In contrast, Factoring showed low communication overhead due to a logarithmic number of tasks (1,120). Static showed—despite the lowest communication overhead—suboptimal performance in cases with time variability per iteration (Corr, Logistic) due to load imbalance. Hence, our optimizer applied Naïve/Factoring.

Optimization Overhead: Dependency analysis is negligible because it took less than 0.1 ms on all use cases, even for Logistic with more than 150 lines of DML. The actual optimization time including plan tree creation was also very low. For use cases Corr and LinReg—with plan tree sizes of $|\mathcal{N}_P| = 21$ and $|\mathcal{N}_P| = 11$, respectively—it was always below 85 ms. The more complex Logistic had a plan size of $|\mathcal{N}_P| = 238$ and $h = 10$ but also required less than 110 ms.

6. RELATED WORK

Large-Scale ML: With regard to declarative ML, we classify existing systems according to its abstraction level. First, at the lowest level, there are many tailor-made, large-scale ML algorithms. Second, frameworks like Mahout [2] and MADlib [29], or platforms like R [9] and Matlab [28] provide dedicated packages for large-scale computation. Third, there is a variety of *distributed* ML systems. Spark [32], ScalOps [4], and Revolution R's `RMR` [9] provide script-level map and reduce functions. Twitter's ML integration [23] uses Pig storage functions. R [9] and Matlab [28] provide—similar to `ParFOR`—parallel for loops. Systems like Ricardo [7] execute R/Jaql queries and R scripts on distributed data partitions. Alternatives include distributed graph processing systems like Distributed GraphLab [24] but those require a vertex-centric view. In contrast, we do not require to explicitly specify the parallelization strategy. Fourth, in SystemML [11, 31] we aim for *declarative* large-scale ML and optimization for runtime. Users write high-level ML algorithms and our optimizer compiles hybrid—potentially distributed—plans. Cumulon [16] introduced several improvements and optimizes for monetary costs under time constraints but does not support task parallelism. Hence, it could benefit from `ParFOR` as well. Fifth, at an even higher level, systems like MLbase [20] specify ML tasks and optimize for runtime and accuracy. This is challenging but orthogonal because our optimizations can be used there as well for realizing logical learning plans.

MR Program Optimization: Cost-based optimization for large-scale ML did not receive much attention yet. However, recent work already laid foundations for MR program and query optimization. This includes static code analysis for operator reordering [17], profiling/cost modeling of black box programs [13], configuration optimization [13, 21], workflow optimization via pickybacking [21, 25] and dynamic re-optimization [1]. These approaches are also important steps towards optimizing ML programs. However, in that context they fall short for two reasons: First, most existing work focus on optimizing pure data flows of relational or black-box operators, which stands in contrast to complex ML programs. RIOT [33] already observed that mapping ML programs to SQL queries does not always exploit the full optimization potential. Accordingly, Zhang et al. optimized I/O sharing for complex programs [34] but for singlenode execution only. Second, due to the focus on black-box operators, there is a lack of analytical cost models for plan comparisons. Cumulon [16] already presented time estimation but for data-parallel ML operations only. Our optimization framework is a first step towards *both* challenges of optimizing *task-parallel* ML programs.

7. CONCLUSIONS

To summarize, we introduced a systematic approach of combined task and data parallelism for large-scale ML. In detail, we presented complementary `ParFOR` parallelization strategies and the related optimization framework. The experiments showed that our optimizer achieves out-of-the-box (1) performance comparable to pure in-memory computations on small problems, (2) significant improvements on larger problems, and (3) very good scalability. In conclusion, users can now specify task- and data-parallel ML algorithms in an easy and flexible way via a high-level primitive, without hand-tuning the execution plan. Most importantly, the combined task and data parallelism on top of MapReduce allows—in contrast to custom parallelization schemes—to share cluster resources with other MR-based systems because the MR scheduler provides global scheduling. Finally, there are many directions for future work. First, this includes advanced runtime and optimization strategies. Second, Hadoop Next Generation (YARN) allows to specify context constraints via so-called resource containers, which gives us even more optimization opportunities.

8. REFERENCES

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-Parallel Computing. In *NSDI*, 2012.

[2] Apache. *Mahout*. `mahout.apache.org`.

[3] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.

[4] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative Systems for Large-Scale Machine Learning. *IEEE Data Eng. Bull.*, 35(2):24–32, 2012.

[5] 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.

[6] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.

[7] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[9] Dirk Eddelbuettel. *CRAN Task View: High-Performance and Parallel Computing with R.* R Project, 2013. `cran.r-project.org/web/views/HighPerformanceComputing.html`.

[10] R. Gemulla, P. J. Haas, E. Nijkamp, and Y. Sismanis. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. IBM Research Report RJ10481, March 2011, Revised February 2013.

[11] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[12] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik. Parallel Support Vector Machines: The Cascade SVM. In *NIPS*, 2004.

[13] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4(11):1111–1122, 2011.

[14] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, 2011.

[15] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *PDIS*, 1991.

[16] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.

[17] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 5(11):1256–1267, 2012.

[18] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Practical and Robust Method for Scheduling Parallel Loops. In *SC*, 1991.

[19] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers Inc., 2002.

[20] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[21] H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for MapReduce Workflows. *PVLDB*, 5(11):1196–1207, 2012.

[22] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust Region Newton Method for Logistic Regression. *Journal of Machine Learning Research*, 9:627–650, 2008.

[23] J. Lin and A. Kolcz. Large-Scale Machine Learning at Twitter. In *SIGMOD*, 2012.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 5(8):716–727, 2012.

[25] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1):494–505, 2010.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[27] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Computers*, 36(12):1425–1439, 1987.

[28] G. Sharma and J. Martin. MATLAB®: A Language for Parallel Computing. *International Journal of Parallel Programming*, 37(1):3–36, 2009.

[29] The MADlib Analytics Library. `madlib.net`.

[30] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[31] Y. Tian, S. Tatikonda, and B. Reinwald. Scalable and Numerically Stable Descriptive Statistics in SystemML. In *ICDE*, 2012.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[33] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR*, 2009.

[34] Y. Zhang and J. Yang. Optimizing I/O for Big Array Analytics. *PVLDB*, 5(8):764–775, 2012.