Translation of Array-Based Loops to Distributed Data-Parallel Programs

Leonidas Fegaras
University of Texas at Arlington
fegaras@cse.uta.edu

Md Hasanuzzaman Noor
University of Texas at Arlington
mdhasanuzzaman.noor@mavs.uta.edu

ABSTRACT

Large volumes of data generated by scientific experiments and simulations come in the form of arrays, while programs that analyze these data are frequently expressed in terms of array operations in an imperative, loop-based language. But, as datasets grow larger, new frameworks in distributed Big Data analytics have become essential tools to large-scale scientific computing. Scientists, who are typically comfortable with numerical analysis tools but are not familiar with the intricacies of Big Data analytics, must now learn to convert their loop-based programs to distributed data-parallel programs. We present a novel framework for translating programs expressed as array-based loops to distributed data parallel programs that is more general and efficient than related work. Although our translations are over sparse arrays, we extend our framework to handle packed arrays, such as tiled matrices, without sacrificing performance. We report on a prototype implementation on top of Spark and evaluate the performance of our system relative to hand-written programs.

1 INTRODUCTION

Most data used in scientific computing and machine learning come in the form of arrays, such as vectors, matrices and tensors, while programs that analyze these data are frequently expressed in terms of array operations in an imperative, loop-based language. These loops are inherently sequential since they iterate over these collections by accessing their elements randomly, one at a time, using array indexing. Current scientific applications must analyze enormous volumes of array data using complex mathematical data processing methods. As datasets grow larger and data analysis computations become more complex, programs written with array-based loops must now be rewritten to run on parallel or distributed architectures. Most scientists though are comfortable with numerical analysis tools, such as MatLab, and with certain imperative languages, such as FORTRAN and C, to express their array-based computations using algorithms found in standard data analysis textbooks, but are not familiar with the intricacies of parallel and distributed computing. Because of the prevalence of array-based programs, a considerable effort has been made to automatically parallelize these loops. Most automated parallelization methods in High Performance Computing (HPC) exploit loop-level parallelism by using multiple threads to access the indexed data in a loop in parallel. But indexed array values that are updated in one loop step may be used in the next steps, thus creating loop-carried dependencies, called recurrences. The presence of such dependencies complicates the parallelization of a loop. DOALL parallelization [28] identifies and parallelizes loops that do not have any recurrences, that is, when statements within a loop can be executed independently. Although there is a substantial body of work on automated parallelization on shared-memory architectures in HPC, there is very little work done on applying these techniques to the new emerging distributed systems for Big Data analysis (with the notable exceptions of MOLD [37] and Casper [2]).

In recent years, new frameworks in distributed Big Data analytics have become essential tools for large-scale machine learning and scientific discoveries. These systems, which are also known as Data-Intensive Scalable Computing (DISC) systems, have revolutionized our ability to analyze Big Data. Unlike HPC systems, which are mainly for shared-memory architectures, DISC systems are distributed data-parallel systems on clusters of shared-nothing computers connected through a high-speed network. One of the earliest DISC systems is Map-Reduce [12], which was introduced by Google and later became popular as an open-source software with Apache Hadoop [5]. For each Map-Reduce job, one needs to provide two functions: a map and a reduce. The map function specifies how to process a single key-value pair to generate a set of intermediate key-value pairs, while the reduce function specifies how to combine all intermediate values associated with the same key. The Map-Reduce framework uses the map function to process the input key-value pairs in parallel by partitioning the data across a number of compute nodes in a cluster. Then, the map results are shuffled across a number of compute nodes so that values associated with the same key are grouped and processed by the same compute node. Recent DISC systems, such as Apache Spark [6] and Apache Flink [4], go beyond Map-Reduce by maintaining dataset partitions in the memory of the compute nodes. Essentially, in their core, these systems remain Map-Reduce systems but

they provide rich APIs that implement many complex operations used in data analysis and support libraries for graph analysis and machine learning.

The goal of this paper is to design and implement a framework that translates array-based loops to DISC operations. Not only do these generated DISC programs have to be semantically equivalent to their original imperative counterparts, but they must also be nearly as efficient as programs written by hand by an expert in DISC systems. If successful, in addition to parallelizing legacy imperative code, such a translation scheme would offer an alternative and more conventional way of developing new DISC applications.

DISC systems use data shuffling to exchange data among compute nodes, which takes place implicitly between the map and reduce stages in Map-Reduce and during group-bys and joins in Spark and Flink. Essentially, all data exchanges across compute nodes are done in a controlled way using DISC operations, which implement data shuffling by distributing data based on some key, so that data associated with the same key are processed together by the same compute node. Our goal is to leverage this idea of data shuffling by collecting the cumulative effects of updates at each memory location across loop iterations and apply these effects in bulk to all memory locations using DISC operations. This idea was first introduced in MOLD [37], but our goal is to design a general framework to translate loop-based programs using compositional rules that transform programs piece-wise, without having to search for program templates to match (as in MOLD [37]) or having to use a program synthesizer (as in Casper [2]).

Consider, for example, the incremental update $A[e]$ += v in a loop, for a sparse vector A. The cumulative effects of all these updates throughout the loop can be performed in bulk by grouping the values v across all loop iterations by the array index e (that is, by the different destination locations) and by summing up these values for each group. Then the entire vector A can be replaced with these new values. For instance, assuming that the values of C were zero before the loop, the following program

```
for i = 0, 9 do
    C[A[i].K] += A[i].V
```

can be evaluated in bulk by grouping the elements $A[i]$ of the vector A by $A[i].K$ (the group-by key), and summing up all the values $A[i].V$ associated with each different group-by key. Then the resulting key-sum pairs are the new values for the vector C. If the sparse vectors C and A are represented as relational tables with schemas (I, V) and (I, K, V), respectively, then the new values of C can be calculated as follows in SQL:

```
insert into C select A.K as I, sum(A.V) as V
from A group by A.K
```

For example, from A on the left we get C on the right:

$A(I,K,V)$	$C(I,V)$
(3,3,10)	(3,23)
(8,5,25)	(5,25)
(5,3,13)	

These results are consistent with the outcome of the loop, which can be unrolled to the updates C[3]+=10; C[3]+=13; C[5]+=25.

Instead of SQL, our framework uses monoid comprehensions [20], which resemble SQL but have less syntactic sugar and are more concise. Our framework translates the previous loop-based program to the following bulk assignment that calculates all the values of C using a bag comprehension that returns a bag of index-value pairs:

$$C \;:=\; \{\,(k, +/v) \mid (i,k,v) \leftarrow A, \textbf{ group by } k\,\}.$$

A group-by operation in a comprehension lifts each pattern variable defined before the group-by (except the group-by keys) from some type t to a bag of t, indicating that each such variable must now contain all the values associated with the same group-by key value. Consequently, after we group by k, the variable v is lifted to a bag of values, one bag for each different k. In the comprehension result, the aggregation $+/v$ sums up all the values in the bag v, thus deriving the new values of C for each index k.

A more challenging example, which is used as a running example throughout this paper, is the product R of two square matrices M and N such that $R_{ij} = \sum_k M_{ik} * N_{kj}$. It can be expressed as follows in a loop-based language:

```
for i = 0, d-1 do
    for j = 0, d-1 do {
        R[i,j] := 0;
        for k = 0, d-1 do
            R[i,j] += M[i,k]*N[k,j]  }
```

A sparse matrix M can be represented as a bag of tuples (i, j, v) such that $v = M_{ij}$. This program too can be translated to a single assignment that replaces the entire content of the matrix R with a new content, which is calculated using bulk relational operations. More specifically, if a sparse matrix is implemented as a relational table with schema (I,J,V), matrix multiplication between the tables M and N can be expressed as follows in SQL:

```
select M.I, N.J, sum(M.V*N.V) as V
from M join N on M.J=N.I group by M.I, N.J
```

As in the previous example, instead of SQL, our framework uses a comprehension and translates the loop-based program

for matrix multiplication to the following assignment:

$$R := \{\, (i,j,+/v) \mid (i,k,m) \leftarrow M,\ (k',j,n) \leftarrow N,\ k = k',\ \textbf{let}\ v = m * n,\ \textbf{group by}\ (i,j) \,\}.$$

Here, the comprehension retrieves the values $M_{ik} \in M$ and $N_{kj} \in N$ as triples (i,k,m) and (k',j,n) so that $k = k'$, and sets $v = m * n = M_{ik} * N_{kj}$. After we group the values by the matrix indexes i and j, the variable v is lifted to a bag of numerical values $M_{ik} * N_{kj}$, for all k. Hence, the aggregation $+/v$ will sum up all the values in the bag v, deriving $\sum_k M_{ik} * N_{kj}$ for the ij element of the resulting matrix. If we ignore non-shuffling operations, this comprehension is equivalent to a join between M and N followed by a reduceByKey operation in Spark.

1.1 Highlights of our Approach

Our framework translates a loop-based program in pieces, in a bottom-up fashion over the abstract syntax tree (AST) representation of the program, by translating every AST node to a comprehension. Matrix indexing is translated as follows:

$$M[i,j] = \{\, m \mid (I,J,m) \leftarrow M,\ I = i,\ J = j \,\}.$$

If M_{ij} exists, it will return the singleton bag $\{M_{ij}\}$, otherwise, it will return the empty bag. Since any matrix access that normally returns a value of t is lifted to a comprehension that returns a bag of t, every term in the loop-based program must be lifted in the same way. For example, the integer multiplication $A * B$ must be lifted to the comprehension $\{\, a*b \mid a \leftarrow A,\ b \leftarrow B \,\}$ over the two bags A and B (the lifted operands) that returns a bag (the lifted result). Consequently, the term $M[i,k]*N[k,j]$in matrix multiplication is translated to:

$$\{\, a * b \mid a \leftarrow \{\, m \mid (I,J,m) \leftarrow M,\ I = i,\ J = k \,\},\ b \leftarrow \{\, n \mid (I,J,n) \leftarrow N,\ I = k,\ J = j \,\} \,\},$$

which, after unnesting the nested comprehensions and renaming some variables, is normalized to:

$$\{\, m * n \mid (I,J,m) \leftarrow M,\ I = i,\ J = k,\ (I',J',n) \leftarrow N,\ I' = k,\ J' = j \,\},$$

which is equivalent to a join between M and N.

Incremental updates, such as $R[i,j]$ += $M[i,k] * N[k,j]$ in matrix multiplication, accumulate their values across iterations, hence they must be considered in conjunction with iterations. Consider the following loop, where $f(k)$, $g(k)$, and $h(k)$ are terms that may depend on k:

for k = 0, 99 **do** $M[f(k),g(k)]$ += $h(k)$.

Suppose now that there are two values, k_1 and $k_2 \neq k_1$, that have the same image under both f and g, that is, when $f(k_1) = f(k_2)$ and $g(k_1) = g(k_2)$. Then, $h(k_1)$ and $h(k_2)$ should be aggregated together. In general, we need to bring together all values $h(k)$ that have the same values for $f(k)$ and $g(k)$. That is, we need to group by $f(k)$ and $g(k)$ and sum up all $h(k)$ in each group. This is accomplished by the comprehension:

$$\{\, (i,\ j,\ +/v) \mid k \leftarrow \text{range}(0,99),\ v \leftarrow h(k),\ \textbf{let}\ i = f(k),\ \textbf{let}\ j = g(k),\ \textbf{group by}\ (i,j) \,\},$$

where $k \leftarrow \text{range}(0,99)$ is an iterator that corresponds to the for-loop and the summation $+/v$ sums up all $h(k)$ that correspond to the same indexes $f(k)$ and $g(k)$.

If we apply this method to $R[i,j]$ += $M[i,k]*N[k,j]$, which is embedded in a triple-nested loop, we derive:

$$\{\, (i,\ j,\ +/v) \mid i \leftarrow \text{range}(0,d-1),\ j \leftarrow \text{range}(0,d-1),\ k \leftarrow \text{range}(0,d-1),\ v \leftarrow M[i,k] * N[k,j],\ \textbf{group by}\ (i,j) \,\}.$$

After replacing $M[i,k] * N[k,j]$ and unnesting the nested comprehensions, we get:

$$\{\, (i,\ j,\ +/v) \mid i \leftarrow \text{range}(0,d-1),\ j \leftarrow \text{range}(0,d-1),\ k \leftarrow \text{range}(0,d-1),\ (I,J,m) \leftarrow M,\ I = i,\ J = k,\ (I',J',n) \leftarrow N,\ I' = k,\ J' = j,\ \textbf{let}\ v = m * n,\ \textbf{group by}\ (i,j) \,\}.$$

Joins between a for-loop and a matrix traversal, such as

$$i \leftarrow \text{range}(0,d-1),\ (I,J,m) \leftarrow M,\ I = i,$$

can be optimized to a matrix traversal, such as

$$(i,J,m) \leftarrow M,\ \text{inRange}(i,0,d-1),$$

where the predicate $\text{inRange}(i,0,d-1)$ returns true if $0 \leq i \leq d-1$. Based on this optimization, the previous comprehension becomes:

$$\{\, (i,\ j,\ +/v) \mid (i,k,m) \leftarrow M,\ \text{inRange}(i,0,d-1),\ (k',j,n) \leftarrow N,\ k = k',\ \textbf{let}\ v = m * n,\ \textbf{group by}\ (i,j) \,\},$$

which is the desired translation of matrix multiplication.

We present a novel framework for translating array-based loops to DISC programs using simple compositional rules that translate these loops piece-wise. Our framework translates an array-based loop to a semantically equivalent DISC program as long as this loop satisfies some simple syntactic restrictions, which are more permissive than the recurrence restrictions imposed by many current systems and can be statically checked at compile-time. For a loop to be parallelizable, many systems require that an array should not be both read and updated in the same loop. For example, they reject the update $V[i] := (V[i-1] + V[i+1])/2$ inside a loop over i because V is read and updated in the same loop. But they also reject incremental updates, such as $V[i]$ += 1, because such an update reads from and writes to the same vector V. Our framework relaxes these restrictions by accepting incremental updates of the form $V[e_1] \oplus= e_2$ in a loop, for some commutative operation $\oplus$ and for some terms e_1 and e_2

that may contain arbitrary array operations, as long as there are no other recurrences present. It translates such an incremental update to a group-by over e_1, followed by a reduction of the e_2 values in each group using the operation $\oplus$. Operation $\oplus$ is required to be commutative because a group-by in a DISC system uses data shuffling across the computing nodes to bring the data that belong to the same group together, which may not preserve the original order of the data. Therefore, a non-commutative reduction may give results that are different from those of the original loop. We have proved the soundness of our framework by showing that our translation rules are meaning preserving for all loop-based programs that satisfy our restrictions. Given that our translation scheme generates DISC operations, this proof implies that loop-based programs that satisfy our restrictions are parallelizable. Furthermore, the class of loop-based programs that can be handled by our framework is equal to the class of programs expressed in our target language, which consists of comprehensions (i.e., basic SQL), while-loops, and assignments to variables. Some real-world programs that contain irregular loops, such as bubble-sort which requires swapping vector elements, are rejected.

Compared to related work (MOLD [37] and Casper [2]): **1)** Our translation scheme is complete under the given restrictions as it can translate correctly any program that does not violate such restrictions, while the related work is very limited and can work on simple loops only. For example, neither of the related systems can translate PageRank or Matrix Factorization. **2)** Our translator is faster than related systems by orders of magnitude in some cases, since it uses compositional transformations without having to search for templates to apply (as in [37]) or use a program synthesizer to explore the space of valid programs (as in [2]). **3)** Our translations have been formally verified, while Casper needs to call an expensive program validator after each program synthesis. Our system, called *DIABLO* (a Data-Intensive Array-Based Loop Optimizer), is implemented on top of DIQL [13, 22], which is a query optimization framework for DISC systems that optimizes SQL-like queries and translates them to Java byte code at compile-time. Currently, DIABLO has been tested on Spark [6], Flink [4], and Scala's Parallel Collections.

Although our translations are over sparse arrays, our framework can easily handle packed arrays, such as tiled matrices, without any fundamental extension. Essentially, the unpack and pack functions that convert dense array structures to sparse arrays and vice versa, are expressed as comprehensions that can be fused with those generated by our framework, thus producing programs that directly access the packed structures without converting them to sparse arrays first. This fusion is hard to achieve in template-based translation systems, such as MOLD [37], which may require different templates for different storage structures. The contributions of this paper are summarized as follows:

- We present a novel framework for translating array-based loops to distributed data parallel programs that is more general and efficient than related work.
- We provide simple rules for dependence analysis that detect recurrences across loops that cannot be handled by our framework.
- We describe how our framework can be extended to handle packed arrays, such as tiled matrices, which can potentially result to a better performance.
- We evaluate the performance of our system relative to hand-written programs on a variety of data analysis and machine learning programs.

This paper is organized as follows. Section 3 describes our framework in detail. Section 4 lists some optimizations on comprehensions that are necessary for good performance. Section 5 explains how our framework can be used on densely packed arrays, such as tiled matrices. Finally, Section 6 gives some performance results for some well-known data analysis programs.

2 RELATED WORK

Most work on automated parallelization in HPC is focused on parallelizing loops that contain array scans without recurrences (DOALL loops) and total reductions (aggregations) [23, 29]. As a generalization of these methods, DOACROSS parallelization [28] separates the loop computations that have no recurrences from the rest of the loop and executes them in parallel, while the rest of the loop is executed sequentially. Other methods that parallelize loops with recurrences simply handle these loops as DOALL computations but they perform a run-time dependency analysis to keep track of the dynamic dependencies, and sequentialize some computations if necessary [43]. Recently, the work by Farzan and Nicolet [16, 17] describes loop-to-loop transformations that augment the loop body with extra computations to facilitate parallelization. Data parallelism is an effective technique for high-level parallel programming in which the same computation is applied to all the elements of a dataset in parallel. Most data parallel languages limit their support to flat data parallelism, which is not well suited to irregular parallel computations. In flat data-parallel languages, the function applied over the elements of a dataset in parallel must be itself sequential, while in nested data-parallel languages this function too can be parallel. Blelloch and Sabot [8] developed a framework to support nested data parallelism using flattening, which is a technique for converting irregular nested computations into regular computations on flat arrays. These techniques have been extended and implemented in various systems, such as Proteus [35]. DISC-based systems do not support nested

parallelism because it is hard to implement in a distributed setting. Spark, for example, does not allow nested RDDs and will raise a run-time error if the function of an RDD operation accesses an RDD. The DIQL and DIABLO translators, on the other hand, allow nested data parallel computations in any form, by translating them to flat-parallel DISC operations by flattening comprehensions and by translating nested comprehensions to DISC joins [22].

The closest work to ours is MOLD [37]. To the best of our knowledge, this was the first work to identify the importance of group-by in parallelizing loops with recurrences in a DISC platform. Like our work, MOLD can handle complex indirect array accesses simply using a group-by operation. But, unlike our work, MOLD uses a rewrite system to identify certain code patterns in a loop and translate them to DISC operations. This means that such a system is as good as its rewrite rules and the heuristic search it uses to apply the rules. Given that the correctness of its translations depends on the correctness of each rewrite rule, each such rule must be written and formally validated by an expert. Another similar system is Casper [2], which translates sequential Java code into semantically equivalent Map-Reduce programs. It uses a program synthesizer to search over the space of sequential program summaries, expressed as IRs. Unlike MOLD, Casper uses a theorem prover based on Hoare logic to prove that the derived Map-Reduce programs are equivalent to the original sequential programs. Our system differs from both MOLD and Casper as it translates loops directly to parallel programs using simple meaning preserving transformations, without having to search for rules to apply. The actual rule-based optimization of our translations is done at a second stage using a small set of rewrite rules, thus separating meaning-preserving translation from optimization.

Another related work on automated parallelization for DISC systems is Map-Reduce program synthesis from input-output examples [39], which is based on recent advances in example-directed program synthesis. One important theorem for parallelizing sequential scans is the third homomorphism theorem, which indicates that any homomorphism (ie, a parallelizable computation) can be derived from two sequential scans; a foldl that scans the sequence from left to right and a foldr that scans it from right to left. This theorem has been used to parallelize sequential programs expressed as folds [34] by heuristically synthesizing a foldr from a foldl first. Along these lines is GRAPE [15], which requires three sequential incremental programs to derive one parallel graph analysis program, although these programs can be quite similar. Lara [32] is a declarative domain-specific language for collections and matrices that allows linear algebra operations on matrices to be mixed with for-comprehensions for collection processing. This deep embedding of matrix and collection operations with the host programming language facilitates better optimization. Although Lara addresses matrix inter-operation optimization, unlike DIABLO, it does not support imperative loops with random matrix indexing. Another area related to automated parallelization for DISC systems is deriving SQL queries from imperative code [14]. Unlike our work, this work addresses aggregates, inserts, and appends to lists but does not address array updates. Finally, our bulk processing of loop updates resembles the framework described in [27], which rewrites a stored procedure to accept a batch of bindings, instead of a single binding. That way, multiple calls to a query under different parameters become a single call to a modified query that processes all parameters in bulk. Unlike our work, which translates imperative loop-based programs on arrays, this framework modifies existing SQL queries and updates.

Many scientific data generated by scientific experiments and simulations come in the form of arrays, such as the results from high-energy physics, cosmology, and climate modeling. Many of these arrays are stored in scientific file formats that are based on array structures, such as, CDF (Common Data Format), FITS (Flexible Image Transport System), GRIB (GRid In Binary), NetCDF (Network Common Data Format), and various extensions to HDF (Hierarchical Data Format), such as HDF5 and HDF-EOS (Earth Observing System). Many array-processing systems use special storage techniques, such as regular tiling, to achieve better performance on certain array computations. TileDB [36] is an array data storage management system that performs complex analytics on scientific data. It organizes array elements into ordered collections called fragments, where each fragment is dense or sparse, and groups contiguous array elements into data tiles of fixed capacity. Unlike our work, the focus of TileDB is the I/O optimization of array operations by using small block updates to update the array stores. SciDB [38, 41] is a large-scale data management system for scientific analysis based on an array data model with implicit ordering. The SciDB storage manager decomposes arrays into a number of equal sized and potentially overlapping chunks, in a way that allows parallel and pipeline processing of array data. Like SciDB, ArrayStore [40] stores arrays into chunks, which are typically the size of a storage block. One of their most effective storage method is a two-level chunking strategy with regular chunks and regular tiles. SystemML [26] is an array-based declarative language to express large-scale machine learning algorithms, implemented on top of Hadoop. It supports many array operations, such as matrix multiplication, and provides alternative implementations to each of them. SciHadoop [9] is a Hadoop plugin that allows scientists to specify logical queries over arrays stored in the NetCDF file format. Their chunking strategy, which is called the Baseline partitioning strategy, subdivides the logical input into a set of partitions (sub-arrays), one for each physical block of the

input file. SciHive [25] is a scalable array-based query system that enables scientists to process raw array datasets in parallel with a SQL-like query language. SciHive maps array datasets in NetCDF files to Hive tables and executes queries via Map-Reduce. Based on the mapping of array variables to Hive tables, SQL-like queries on arrays are translated to HiveQL queries on tables and then optimized by the Hive query optimizer. SciMATE [44] extends the Map-Reduce API to support the processing of the NetCDF and HDF5 scientific formats, in addition to flat-files. SciMATE supports various optimizations specific to scientific applications by selecting a small number of attributes used by an application and perform data partition based on these attributes. TensorFlow [1] is a dataflow language for machine learning that supports data parallelism on multi-core machines and GPUs but has limited support for distributed computing. Finally, MLlib [33] is a machine learning library built on top of Spark and includes algorithms for fast matrix manipulation based on native (C++ based) linear algebra libraries. Furthermore, MLlib provides a uniform rigid set of high-level APIs that consists of several statistical, optimization, and linear algebra primitives that can be used as building blocks for data analysis applications.

3 OUR FRAMEWORK

3.1 Syntax of the Loop-Based Language

The syntax of the loop-based language is given in Figure 1. This is a proof-of-concept loop-based language; many other languages, such as Java or C, can be used instead. Types of values include parametric types for various kinds of collections, such as vectors, matrices, key-value maps, bags, lists, etc. To simplify our translation rules and examples in this section, we do not allow nested arrays, such as vectors of vectors. There are two kinds of assignments, an incremental update $d \oplus= e$ for some commutative operation $\oplus$, which is equivalent to the update $d := d \oplus e$, and all other assignments $d := e$. To simplify translation, variable declarations, **var** $v : t = e$, cannot appear inside for-loops. There are two kinds of for-loops that can be parallelized: a for-loop in which an index variable iterates over a range of integers, and a for-loop in which a variable iterates over the elements of a collection, such as the values of an array. Our current framework generates sequential code from a while-loop. Furthermore, if a for-loop contains a while-loop in its body, then this for-loop too becomes sequential and it is treated as a while-loop. Finally, a statement block contains a sequence of statements.

3.2 Restrictions for Parallelization

Our framework can translate for-loops to equivalent DISC programs when these loops satisfy certain restrictions described in this section. In Appendix A, we provide a proof that, under these restrictions, our transformation rules to be presented in Section 3.8 are meaning preserving, that is, the programs generated by our translator are equivalent to the original loop-based programs. In other words, since our target language is translated to DISC operations, the loop-based programs that satisfy our restrictions are parallelizable.

Our restrictions use the following definitions. For any statement s in a loop-based program, we define the following three sets of L-values (destinations): the readers $\mathcal{R}[\![s]\!]$, the writers $\mathcal{W}[\![s]\!]$, and the aggregators $\mathcal{A}[\![s]\!]$. The **readers** are the L-values read in s, the **writers** are the L-values written (but not incremented) in s, and the **aggregators** are the L-values incremented in s. For example, for the following statement:

$$V[W[i]] \mathrel{+}= n * C[i] * C[i+1],$$

where i is a loop index, the aggregators are $\mathcal{A}[\![s]\!] = \{V[W[i]]\}$, the readers are $\mathcal{R}[\![s]\!] = \{W[i], n, C[i], C[i+1]\}$, and the writers are $\mathcal{W}[\![s]\!] = \emptyset$. Two L-values d_1 and d_2 **overlap**, denoted by overlap(d_1, d_2), if they are the same variable, or they are equal to the projections $d_1'.A$ and $d_2'.A$ with overlap(d_1', d_2'), or they are array accesses over the same array name. The **context** of a statement s, context(s), is the set of outer loop indexes for all loops that enclose s. Note that, each for-loop must have a distinct loop index variable; if not, the duplicate loop index is replaced with a fresh variable. For an L-value d, indexes(d) is the set of loop indexes used in d.

An **affine** expression [3] takes the form

$$c_0 + c_1 * i_1 + \cdots + c_k * i_k,$$

where $i_1, \ldots, i_k$ are loop indexes and $c_0, \ldots, c_k$ are constants. For an L-value d in a statement s, affine(d, s) is true if d is a variable, or a projection $d'.A$ with affine(d', s), or an array indexing $v[e_1, \ldots, e_n]$, where each index e_i is an affine expression and all loop indexes in context(s) are used in d. In other words, if affine(d, s) is true, then d is stored at different locations for different values of the loop indexes in context(s).

Definition 3.1 (Affine For-Loop). *A for-loop statement s is affine if s satisfies the following properties:*

1. *for any update $d := e$ in s,* affine(d, s);
2. *there are no dependencies between any two statements s_1 and s_2 in s, that is, if there are no L-values $d_1 \in (\mathcal{A}[\![s_1]\!] \cup \mathcal{W}[\![s_1]\!])$ and $d_2 \in \mathcal{R}[\![s_2]\!]$ such that* overlap(d_1, d_2), *with the following exceptions:*
   (a) *if $d_1 \in \mathcal{W}[\![s_1]\!]$, $d_1 = d_2$, and s_1 precedes s_2;*

Type:

t	::=	v	basic type (int, float, ...)
	|	$v[t]$	parametric type
	|	$(t_1, \ldots, t_n)$	tuple type
	|	$\langle A_1 : t_1, \ldots, A_n : t_n \rangle$	record type

Expression:

e	::=	d	a destination (L-value)
	|	$e_1 \star e_2$	any binary operation $\star$
	|	$(e_1, \ldots, e_n)$	tuple construction
	|	$\langle A_1 = e_1, \ldots, A_n = e_n \rangle$	record construction
	|	*const*	constant (int, float, ...)

Destination (L-value):

d	::=	v	variable
	|	$d.A$	record projection
	|	$v[e_1, \ldots, e_n]$	array indexing

Statement:

s	::=	$d \oplus= e$	incremental update
	|	$d := e$	assignment
	|	**var** $v : t = e$	declaration
	|	**for** $v = e_1, e_2$ **do** s	iteration
	|	**for** v **in** e **do** s	traversal
	|	**while** (e) s	loop
	|	**if** (e) s_1 [**else** s_2]	conditional
	|	$\{ s_1; \ldots; s_n \}$	statement block

Figure 1: Syntax of loop-based programs

(b) *if* $d_1 \in \mathcal{A}[\![s_1]\!]$, $d_1 = d_2$, affine(d_2, s_2), s_1 *precedes* s_2, *and* context$(s_1) \cap$ context$(s_2) =$ indexes(d_1).

Restriction 1 indicates that the destination of any non-incre-mental update must be a different location at each loop iteration. If the update destination is an array access, the array indexes must be affine and completely cover all surrounding loop indexes. This restriction does not hold for incremental updates, which allow arbitrary array indexes in a destination as long as the array is not read in the same loop. Restriction 2 combined with exception (a) rejects any read and write on the same array in a loop except when the read is after the write and the read and write are at the same location ($d_1 = d_2$), which, based on Restriction 1, is a different location at each loop iteration. Exception (b) indicates that if we first increment and then read the same location, then these two operations must not be inside a for-loop whose loop index is not used in the destination. This is because the increment of the destination is done within the for-loops whose loop indexes are used in the destination and across the rest of the surrounding for-loops. For example, the following loop:

for $i = \ldots$ **do** { **for** $j = \ldots$ **do** { $V[i]$ += 1 }; $W[i] := V[i]$ },

increments and reads $V[i]$. The contexts of the first and second updates are $\{i, j\}$ and $\{i\}$, respectively, and their intersection gives $\{i\}$, which is equal to the indexes of $V[i]$. If there were another statement $M[i, j] := V[i]$ inside the inner loop, this would violate Exception (b) since their context intersection would have been $\{i, j\}$, which is not equal to the indexes of $V[i]$.

An affine for-loop satisfies the following theorem, which is proved in Appendix A. It is used as the basis of our program translations.

Theorem 3.1. *An affine for-loop satisfies:*

$$\begin{aligned} &\textbf{\textit{for}}\ i = \ldots\ \textbf{\textit{do}}\ \{\, s_1;\ s_2 \,\} \\ &\quad = \{\, \textbf{\textit{for}}\ i = \ldots\ \textbf{\textit{do}}\ s_1;\ \textbf{\textit{for}}\ i = \ldots\ \textbf{\textit{do}}\ s_2 \,\}. \end{aligned} \tag{1}$$

In fact, our restrictions in Definition 3.1 were designed in such a way that all affine for-loops satisfy this theorem and at the same time are inclusive enough to accept as many common loop-based programs as possible. In Appendix A, we prove that our program translations, to be described in Section 3.8, under the restrictions in Definition 3.1 are meaning preserving, which implies that all affine for-loops are parallelizable since the target of our translations is DISC operations.

For example, the incremental update:

$$\textbf{for}\ i = \ldots\ \textbf{do}\ C[V[i].K] \mathrel{+}= V[i].D,$$

which counts all $V[i].D$ in groups that have the same key $V[i].K$, satisfies our restrictions since it increments but does not read C. On the other hand, some non-incremental updates may outright be rejected. For example, the loop:

$$\textbf{for}\ i = \ldots\ \textbf{do}\ V[i] := (V[i-1] + V[i+1])/2$$

will be rejected by Restriction 2 because V is both a reader and a writer. To alleviate this problem, one may rewrite this loop as follows:

$$\begin{aligned} &\textbf{for}\ i = \ldots\ \textbf{do}\ V'[i] := V[i]; \\ &\textbf{for}\ i = \ldots\ \textbf{do}\ V[i] := (V'[i-1] + V'[i+1])/2, \end{aligned}$$

which first stores V to V' and then reads V' to compute V. This program satisfies our restrictions but is not equivalent to the original program because it uses the previous values of V to compute the new ones. Another example is:

$$\textbf{for}\ i = \ldots\ \textbf{do}\ \{\, n := V[i];\ W[i] := f(n) \,\},$$

which is also rejected because n is not affine as it does not cover the loop indexes (namely, i). To fix this problem, one may redefine n as a vector and rewrite the loop as:

$$\textbf{for } i = \ldots \textbf{ do } \{\, n[i] := V[i];\ W[i] := f(n[i]) \,\}.$$

Redefining variables by adding to them more array dimensions is currently done manually by a programmer, but we believe that it can be automated when a variable that violates our restrictions is detected.

A more complex example is matrix factorization using gradient descent [30]. The goal of matrix factorization is to split a matrix R of dimension $n \times m$ into two low-rank matrices P and Q of dimensions $n \times l$ and $l \times m$, for small l, such that the error between the predicted and the original matrix $R - P \times Q$ is below some threshold. One step of matrix factorization that computes the new values P and Q from the previous values P' and Q' can be implemented using the following loop-based program:

```
for i = 0, n−1 do
    for j = 0, m−1 do {
        pq := 0.0;
        for k = 0, l−1 do
            pq += P'[i,k]*Q'[k,j];
        error := R[i,j]−pq;
        for k = 0, l−1 do {
            P[i,k] += a*(2*error*Q'[k,j]−b*P'[i,k]);
            Q[k,j] += a*(2*error*P'[i,k]−b*Q'[k,j]);  }}
```

where a is the learning rate and b is the normalization factor used in avoiding overfitting. This program first computes pq, which is the i, j element of $P' \times Q'$, and error, which is the i, j element of $R - P' \times Q'$. Then, it uses error to improve P and Q. This program is rejected because the destinations of the assignments pq := 0.0 and error := R[i,j]-pq do not cover all loop indexes, and the read of pq violates exception (b) (since the intersection of the contexts of pq += P'[i,k]*Q'[k,j] and error := R[i,j]-pq is {i,j}, which is not equal to the indexes of pq). To rectify these problems, we can convert the variables pq and error to matrices, so that, instead of pq and error, we use pq[i,j] and error[i,j].

3.3 Monoid Comprehensions

The target of our translations consists of monoid comprehensions, which are equivalent to the SQL select-from-where-group-by-having syntax. Monoid comprehensions were first introduced and used in the 90's as a formal basis for ODMG OQL [21]. They were recently used as the formal calculus for the DISC query languages MRQL [20] and DIQL [22]. The formal semantics of monoid comprehensions, the query optimization framework, and the translation of comprehensions to a DISC algebra, are given in our earlier work [20, 22]. Here, we describe the syntax only.

A monoid comprehension has the following syntax:

$$\{\, e \mid q_1,\ \ldots,\ q_n \,\},$$

where the expression e is the comprehension head and a qualifier q_i is defined as follows:

Qualifier:

q	::=	$p \leftarrow e$	generator
	|	**let** $p = e$	let-binding
	|	e	condition
	|	**group by** $p\,[:e\,]$	group-by

Pattern:

p	::=	v	pattern variable
	|	$(p_1, \ldots, p_n)$	tuple pattern.

The domain e of a generator $p \leftarrow e$ must be a bag. This generator draws elements from this bag and, each time, it binds the pattern p to an element. A condition qualifier e is an expression of type boolean. It is used for filtering out elements drawn by the generators. A let-binding **let** $p = e$ binds the pattern p to the result of e. A group-by qualifier uses a pattern p and an optional expression e. If e is missing, it is taken to be p. The group-by operation groups all the pattern variables in the same comprehension that are defined before the group-by (except the variables in p) by the value of e (the group-by key), so that all variable bindings that result to the same key value are grouped together. After the group-by, p is bound to a group-by key and each one of these pattern variables is lifted to a bag of values. The result of a comprehension $\{\, e \mid q_1,\ \ldots,\ q_n \,\}$ is a bag that contains all values of e derived from the variable bindings in the qualifiers.

Comprehensions can be translated to algebraic operations that resemble the bulk operations supported by many DISC systems, such as groupBy, join, map, and flatMap. We use $\overline{q}$ to represent the sequence of qualifiers $q_1,\ \ldots,\ q_n$, for $n \geq 0$. To translate a comprehension $\{\, e \mid \overline{q} \,\}$ to the algebra, the group-by qualifiers are first translated to groupBy operations from left to right. Given a bag X of type $\{(K, V)\}$, groupBy(X) groups the elements of X by their first component of type K (the group-by key) and returns a bag of type $\{(K, \{V\})\}$. Let $v_1, \ldots, v_n$ be the pattern variables in the sequence of qualifiers $\overline{q_1}$ that do not appear in the group-by pattern p, then we have:

$$\begin{aligned}&\{\, e' \mid \overline{q_1},\ \textbf{group by}\, p : e,\ \overline{q_2} \,\}\\ &= \{\, e' \mid (p, s) \leftarrow \text{groupBy}(\{\, (e, (v_1, \ldots, v_n)) \mid \overline{q_1} \,\}),\\ &\qquad \forall i :\ \textbf{let}\ v_i = \{\, v_i \mid (v_1, \ldots, v_n) \leftarrow s \,\},\ \overline{q_2} \,\}.\end{aligned}$$

That is, for each pattern variable v_i, this rule embeds a let-binding so that this variable is lifted to a bag that contains all

v_i values in the current group. Then, comprehensions without any group-by are translated to the algebra by translating the qualifiers from left to right:

$$\begin{aligned}\{\, e' \mid p \leftarrow e,\, \overline{q}\,\} &= \text{flatMap}(\lambda p.\, \{\, e' \mid \overline{q}\,\},\, e)\\ \{\, e' \mid \textbf{let } p = e,\, \overline{q}\,\} &= \textbf{let } p = e \textbf{ in } \{\, e' \mid \overline{q}\,\}\\ \{\, e' \mid e,\, \overline{q}\,\} &= \textbf{if } e \textbf{ then } \{\, e' \mid \overline{q}\,\} \textbf{ else } \emptyset\\ \{\, e' \mid \,\} &= \{e'\}.\end{aligned}$$

Given a function f that maps an element of type T to a bag of type $\{S\}$ and a bag X of type $\{T\}$, the operation flatMap(f, X) maps the bag X to a bag of type $\{S\}$ by applying the function f to each element of X and unioning together the results. Although this translation generates nested flatMaps from join-like comprehensions, there is a general method for identifying all possible equi-joins from nested flatMaps, including joins across deeply nested comprehensions, and translating them to joins and coGroups [20].

Finally, nested comprehensions can be unnested by the following rule:

$$\begin{aligned}&\{\, e_1 \mid \overline{q_1},\, p \leftarrow \{\, e_2 \mid \overline{q_3}\,\},\, \overline{q_2}\,\}\\ &\qquad = \{\, e_1 \mid \overline{q_1},\, \overline{q_3},\, \textbf{let } p = e_2,\, \overline{q_2}\,\} \qquad (2)\end{aligned}$$

for any sequence of qualifiers $\overline{q_1}$, $\overline{q_2}$, and $\overline{q_3}$. This rule can only apply if there is no group-by qualifier in $\overline{q_3}$ or when $\overline{q_1}$ is empty. It may require renaming the variables in $\{\, e_2 \mid \overline{q_3}\,\}$ to prevent variable capture.

3.4 Array Representation

In our framework, a sparse array, such as a sparse vector or a matrix, is represented as a key-value map (also known as an indexed set), which is a bag of type $\{(K, T)\}$, where K is the array index type and T is the array value type. More specifically, a sparse vector of type vector[T] is captured as a key-value map of type $\{(\text{long}, T)\}$, while a sparse matrix of type matrix[T] is captured as a key-value map of type $\{((\text{long}, \text{long}), T)\}$.

Merging two compatible arrays is done with the array merging operation $\triangleleft$, defined as follows:

$$\begin{aligned}X \triangleleft Y \;&= \{\, (k, b) \mid (k, a) \leftarrow X,\, (k', b) \in Y,\, k = k'\,\}\\ &\uplus \{\, (k, a) \mid (k, a) \leftarrow X,\, k \notin \Pi_1(Y)\,\}\\ &\uplus \{\, (k, b) \mid (k, b) \leftarrow Y,\, k \notin \Pi_1(X)\,\},\end{aligned}$$

where $\Pi_1(X)$ returns the keys of X. That is, $X \triangleleft Y$ is the union of X and Y, except when there is $(k, x) \in X$ and $(k, y) \in Y$, in which case it chooses the latter value, (k, y). For example, $\{(3, 10), (1, 20)\} \triangleleft \{(1, 30), (4, 40)\}$ is equal to $\{(3, 10), (1, 30), (4, 40)\}$. On Spark, the $\triangleleft$ operation can be implemented as a coGroup.

An update to a vector $V[e_1] := e_2$ is equivalent to the assignment $V := V \triangleleft \{(e_1, e_2)\}$. That is, the new value of V is the current vector V but with the value associated with the index e_1 (if any) replaced with e_2. Similarly, an update to a matrix $M[e_1, e_2] := e_3$ is equivalent to the assignment $M := M \triangleleft \{((e_1, e_2), e_3)\}$.

Array indexing though is a little bit more complex because the indexed element may not exist in the sparse array. Instead of a value of type T, indexing over an array of T should return a bag of type $\{T\}$, which can be $\{v\}$ for some value v of type T, if the value exists, or $\emptyset$, if the value does not exist. Then, the vector indexing $V[e]$ is $\{\, v \mid (i, v) \leftarrow V,\, i = e\,\}$, which returns a bag of type $\{T\}$. Similarly, the matrix indexing $M[e_1, e_2]$ is $\{\, v \mid ((i, j), v) \leftarrow M,\, i = e_1,\, j = e_2\,\}$.

We are now ready to express any assignment that involves vectors and matrices. For example, consider the matrices R, M, and N of type matrix[float]. The assignment:

$$R[i, j] := M[i, k] * N[k, j] \qquad (3)$$

is translated to the assignment:

$$\begin{aligned}R := R \triangleleft \{\, ((i, j), m * n) \mid &((i, k), m) \leftarrow M, \qquad (4)\\ &((k', j), n) \leftarrow N,\, k = k'\,\},\end{aligned}$$

which uses a bag comprehension equivalent to a join between the matrices M and N. This assignment can be derived from assignment (3) using simple transformations. To understand these transformations, consider the product $X * Y$. Since both X and Y have been lifted to bags, because they may contain array accesses, this product must also be lifted to a comprehension that extracts the values of X and Y, if any, and returns their product:

$$X * Y = \{\, x * y \mid x \leftarrow X,\, y \leftarrow Y\,\}.$$

Given that matrix accesses are expressed as:

$$\begin{aligned}M[i, k] &= \{\, m \mid ((I, J), m) \leftarrow M,\, I = i,\, J = k\,\}\\ N[k, j] &= \{\, n \mid ((I, J), n) \leftarrow N,\, I = k,\, J = j\,\},\end{aligned}$$

the product $M[i, k] * N[k, j]$ is equal to:

$$\begin{aligned}\{\, x * y \mid\, &x \leftarrow \{\, m \mid ((I, J), m) \leftarrow M,\, I = i,\, J = k\,\},\\ &y \leftarrow \{\, n \mid ((I, J), n) \leftarrow N,\, I = k,\, J = j\,\}\,\},\end{aligned}$$

which is normalized as follows using Rule (2), after some variable renaming:

$$\begin{aligned}&\{\, x * y \mid ((I, J), m) \leftarrow M,\, I = i,\, J = k,\, \textbf{let } x = m,\\ &\qquad ((I', J'), n) \leftarrow N,\, I' = k,\, J' = j,\, \textbf{let } y = n\,\}\\ &= \{\, m * n \mid ((I, J), m) \leftarrow M,\, I = i,\, J = k,\\ &\qquad\qquad ((I', J'), n) \leftarrow N,\, I' = k,\, J' = j\,\}.\end{aligned}$$

Lastly, since the value of e in the assignment $R[i, j] := e$ is lifted to a bag, this assignment is translated to $R := R \triangleleft \{\, ((i, j), v) \mid v \leftarrow e\,\}$, that is, R is augmented with an indexed set that results from accessing the lifted value of e. If e contains a value, the comprehension will return a singleton bag, which will replace $R[i, j]$ with that value. After substituting

the value e with the term derived for $M[i,k] * N[k,j]$, we get an assignment equivalent to the assignment (4).

3.5 Handling Array Updates in a Loop

We now address the problem of translating array updates in a loop. We classify updates into two categories:

(1) Incremental updates of the form $d := d \oplus e$, for some commutative operation $\oplus$, where d is an update destination, which is also repeated as the left operand of $\oplus$. It can also be written as $d \oplus= e$. For example, $V[i] \mathrel{+}= 1$ increments $V[i]$ by 1.
(2) All other updates of the form $d := e$.

Consider the following loop with a non-incremental update:

$$\textbf{for } i = 1, N \textbf{ do } V[g(i)] := W[f(i)] \tag{5}$$

for some vectors V and W, and some terms $f(i)$ and $g(i)$ that depend on the index i. Our framework translates this loop to an update to the vector V, where all the elements of V are updated at once, in a parallel fashion:

$$V := V \lhd \{\, (g(i), v) \mid i \leftarrow \text{range}(1, N), \\ (k, v) \leftarrow V,\ k = f(i) \,\}. \tag{6}$$

But this expression may not produce the same vector V as the original loop if there are recurrences in the loop, such as, when the loop body is $V[i] := V[i-1]$. Furthermore, the join between range$(1, N)$ and W in (6) looks unnecessary. We will transform such joins to array traversals in Section 3.6.

In our framework, for-loops are embedded as generators inside the comprehensions that are associated with the loop assignments. Consider, for example, matrix copying:

$$\textbf{for } i = 1, 10 \textbf{ do for } j = 1, 20 \textbf{ do } M[i,j] := N[i,j].$$

Using the translation of the assignment $M[i,j] := N[i,j]$, the loop becomes:

$$\textbf{for } i = 1, 10 \textbf{ do} \\ \quad \textbf{for } j = 1, 20 \textbf{ do} \\ \quad\quad M := M \lhd \{\, ((i,j), n) \mid ((I,J), n) \leftarrow N, \\ \quad\quad\quad I = i,\ J = j \,\}. \tag{7}$$

To parallelize this loop, we embed the for-loops inside the comprehension as generators:

$$M := M \lhd \{\, ((i,j), n) \mid i \leftarrow \text{range}(1, 10), \\ j \leftarrow \text{range}(1, 20), \\ ((I,J), n) \leftarrow N,\ I = i,\ J = j \,\}. \tag{8}$$

Notice the difference between the loop (7) and the assignment (8). The former will do 10*20 updates to M while the latter will only do one bulk update that will replace all $M[i,j]$ with $N[i,j]$ at once. This transformation can only apply when there are no recurrences across iterations.

3.6 Eliminating Loop Iterations

Before we present the details of program translation, we address the problem of eliminating index iterations, such as range(1, N) in assignment (6), and range(1, 10) and range(1, 20) in assignment (8). If there is a right inverse F of f such that $f(F(k)) = k$, then the assignment (6) is optimized to:

$$V := V \lhd \{\, (g(F(k)), v) \mid (k, v) \leftarrow W, \\ \text{inRange}(F(k), 1, N) \,\}, \tag{9}$$

where the predicate inRange$(F(k), 1, N)$ returns true if $F(k)$ is within the range $[1, N]$. Given that the right-hand side of an update may involve multiple array accesses, we can choose one whose index term can be inverted. For example, for $V[i-1]$, the inverse of $k = i - 1$ is $i = k + 1$. In the case where no such inverse can be derived, the range iteration simply remains as is. One such example is the loop $\textbf{for } i = 1, N \textbf{ do } V[i] := 0$, which is translated to $V := V \lhd \{\, (i, 0) \mid i \leftarrow \text{range}(1, N) \,\}$.

3.7 Handling Incremental Updates

There is an important class of recurrences in loops that can be parallelized using group-by and aggregation. Consider, for example, the following loop with an incremental update:

$$\textbf{for } i = 1, N \textbf{ do } V[g(i)] \mathrel{+}= W[i]. \tag{10}$$

Let's say, for example, that there are 3 indexes overall, i_1, i_2, and i_3, that have the same image under g, ie, $k = g(i_1) = g(i_2) = g(i_3)$. Then, $V[k]$ must be set to $V[k] + W[i_1] + W[i_2] + W[i_3]$. In general, we need to bring together all values of W whose indexes have the same image under g. That is, we need to group by $g(i)$. Hence, the loop can be translated to a comprehension with a group-by:

$$V := V \lhd \{\, (k, v + (+/w)) \mid (i, w) \leftarrow W,\ \text{inRange}(i, 1, N), \\ \textbf{group by } k : g(i),\ (j, v) \leftarrow V,\ j = k \,\},$$

which groups W by the destination index $g(i)$ and, for each group, it calculates the aggregation $+/w$ of all values $w = W[i]$ with the same $g(i)$ value, but also adds the original value $v = V[g(i)]$ before the group-by.

If the destination of the incremental update is a variable, such as in $n \mathrel{+}= W[i]$, then the group-by is over $()$, since there are no indexes used in n:

$$n := \{\, n + (+/w) \mid (i, w) \leftarrow W,\ \text{inRange}(i, 1, N), \\ \textbf{group by } k : () \,\}.$$

This group-by can be eliminated because it forms a single group; in which case the variable w is lifted to a bag that contains all the values of W:

$$n := \{\, n + (+/\{\, w \mid (i, w) \leftarrow W,\ \text{inRange}(i, 1, N) \,\}) \,\}.$$

We will discuss optimizations like this in Section 4.

$\mathcal{E}[\![e]\!]$: Translate the expression e to a comprehension term

$$\mathcal{E}[\![V]\!] = \{\, V \,\} \tag{11a}$$
$$\mathcal{E}[\![e.A]\!] = \{\, v.A \mid v \leftarrow \mathcal{E}[\![e]\!] \,\} \tag{11b}$$
$$\mathcal{E}[\![V[e_1,\ldots,e_n]]\!] = \{\, v \mid k_1 \leftarrow \mathcal{E}[\![e_1]\!], \ldots, k_n \leftarrow \mathcal{E}[\![e_n]\!], ((i_1,\ldots,i_n),v) \leftarrow V,\ i_1 = k_1, \ldots, i_n = k_n \,\} \tag{11c}$$
$$\mathcal{E}[\![e_1 \star e_2]\!] = \{\, v_1 \star v_2 \mid v_1 \leftarrow \mathcal{E}[\![e_1]\!],\ v_2 \leftarrow \mathcal{E}[\![e_2]\!] \,\} \tag{11d}$$
$$\mathcal{E}[\![(e_1,\ldots,e_n)]\!] = \{\, (v_1,\ldots,v_n) \mid v_1 \leftarrow \mathcal{E}[\![e_1]\!],\ \ldots,\ v_n \leftarrow \mathcal{E}[\![e_n]\!] \,\} \tag{11e}$$
$$\mathcal{E}[\![\langle A_1 = e_1, \ldots, A_n = e_n \rangle]\!] = \{\, \langle A_1 = v_1, \ldots, A_n = v_n \rangle \mid v_1 \leftarrow \mathcal{E}[\![e_1]\!],\ \ldots,\ v_n \leftarrow \mathcal{E}[\![e_n]\!] \,\} \tag{11f}$$
$$\mathcal{E}[\![const]\!] = \{\, const \,\} \tag{11g}$$

$\mathcal{K}[\![d]\!]$: Derive the destination index from d

$$\mathcal{K}[\![V]\!] = \{\, () \,\} \tag{12a}$$
$$\mathcal{K}[\![d.A_i]\!] = \mathcal{K}[\![d]\!] \tag{12b}$$
$$\mathcal{K}[\![V[e_1,\ldots,e_n]]\!] = \mathcal{E}[\![(e_1,\ldots,e_n)]\!] \tag{12c}$$

$\mathcal{D}[\![d]\!](k)$: Derive d from the destination index k

$$\mathcal{D}[\![V]\!](k) = \{\, V \,\} \tag{13a}$$
$$\mathcal{D}[\![d.A_i]\!](k) = \{\, v.A_i \mid v \leftarrow \mathcal{D}[\![d]\!](k) \,\} \tag{13b}$$
$$\mathcal{D}[\![V[e_1,\ldots,e_n]]\!](k) = \{\, v \mid ((i_1,\ldots,i_n),v) \leftarrow V,\ (i_1,\ldots,i_n) = k \,\} \tag{13c}$$

$\mathcal{U}[\![d]\!](x)$: Update the destination d with the value x

$$\mathcal{U}[\![V]\!](x) = [\, V := \{\, v \mid (k,v) \leftarrow x \,\} \,] \tag{14a}$$
$$\mathcal{U}[\![d.A_i]\!](x) = \mathcal{U}[\![d]\!](\{\, (k, \langle A_1 = w.A_1, \ldots, A_i = v, \ldots, A_n = w.A_n \rangle) \mid (k,v) \leftarrow x,\ w \leftarrow \mathcal{D}[\![d]\!](k) \,\}) \tag{14b}$$
$$\mathcal{U}[\![V[e_1,\ldots,e_n]]\!](x) = [\, V := V \triangleleft x \,] \tag{14c}$$

$\mathcal{S}[\![s]\!](\overline{q})$: Translate the statement s to a target code block using the list of for-loop qualifiers $\overline{q}$

$$\mathcal{S}[\![d \oplus= e]\!](\overline{q}) = \mathcal{U}[\![d]\!](\{\, (k, w \oplus (\oplus/v)) \mid \overline{q},\ v \leftarrow \mathcal{E}[\![e]\!],\ k \leftarrow \mathcal{K}[\![d]\!],\ \textbf{group by}\ k,\ w \leftarrow \mathcal{D}[\![d]\!](k) \,\}) \tag{15a}$$
$$\mathcal{S}[\![d := e]\!](\overline{q}) = \mathcal{U}[\![d]\!](\{\, (k,v) \mid \overline{q},\ v \leftarrow \mathcal{E}[\![e]\!],\ k \leftarrow \mathcal{K}[\![d]\!] \,\}) \tag{15b}$$
$$\mathcal{S}[\![\textbf{var}\ V : t = e]\!](\overline{q}) = \mathcal{S}[\![V := e]\!](\overline{q}) \tag{15c}$$
$$\mathcal{S}[\![\textbf{for}\ v = e_1, e_2\ \textbf{do}\ s]\!](\overline{q}) = \mathcal{S}[\![s]\!](\overline{q} +\!\!+ [\, v_1 \leftarrow \mathcal{E}[\![e_1]\!],\ v_2 \leftarrow \mathcal{E}[\![e_2]\!],\ v \leftarrow \text{range}(v_1,v_2) \,]) \tag{15d}$$
$$\mathcal{S}[\![\textbf{for}\ v\ \textbf{in}\ e\ \textbf{do}\ s]\!](\overline{q}) = \mathcal{S}[\![s]\!](\overline{q} +\!\!+ [\, A \leftarrow \mathcal{E}[\![e]\!],\ (i,v) \leftarrow A \,]) \tag{15e}$$
$$\mathcal{S}[\![\textbf{while}\ (e)\ s]\!](\overline{q}) = [\, \text{while}(\mathcal{E}[\![e]\!], \mathcal{S}[\![s]\!](\overline{q})) \,] \tag{15f}$$
$$\mathcal{S}[\![\textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2]\!](\overline{q}) = \mathcal{S}[\![s_1]\!](\overline{q} +\!\!+ [\, p \leftarrow \mathcal{E}[\![e]\!],\ p \,]) +\!\!+ \mathcal{S}[\![s_2]\!](\overline{q} +\!\!+ [\, p \leftarrow \mathcal{E}[\![e]\!],\ !p \,]) \tag{15g}$$
$$\mathcal{S}[\![\{\, s_1; \ldots; s_n \,\}]\!](\overline{q}) = \mathcal{S}[\![s_1]\!](\overline{q}) +\!\!+ \cdots +\!\!+ \mathcal{S}[\![s_n]\!](\overline{q}) \tag{15h}$$

Figure 2: Rules for translating loop-based programs to target code

3.8 Program Translation

The target of our translations is a list of statements, where a statement c has the following syntax:

Target Code:

c	::=	$v := e$	assignment
	|	$\text{while}(e, c)$	loop
	|	$[\, c_1, \ldots, c_n \,]$	code block

In the target code, an assignment to a variable v of type t gets a value e of type $\{t\}$. An assignment to an array is done in bulk, by replacing the entire array with a new one. The while-loop corresponds to the while statement in Figure 1; it repeats the code c in its body while the condition e is true. Finally, a code block is like a block of statements that need to be evaluated in order.

The rules for translating loop-based programs to the target code are given in Figure 2. They are mainly given in terms

of the semantic functions $\mathcal{E}$ and $\mathcal{S}$ that translate expressions and statements, respectively. The syntactic brackets $[\![\ldots]\!]$ enclose syntactic elements, as defined in Figure 1. The rules for $\mathcal{E}[\![e]\!]$, given in Equations (11a)-(11g), translate an expression e of type t to a comprehension term of type $\{t\}$. For example, using Equation (11c), $M[1,2]$ is translated to:

$$\{\, v \mid k \leftarrow \{\,1\,\},\ l \leftarrow \{\,2\,\},\ ((i,j),v) \leftarrow M,\ i = k,\ j = l \,\} = \{\, v \mid ((i,j),v) \leftarrow M,\ i = 1,\ j = 2 \,\}.$$

The rules for $\mathcal{S}[\![s]\!](\overline{q})$, given in Equations (15a)-(15h), translate a statement s to a list of target code statements. $\mathcal{S}[\![s]\!](\overline{q})$ is parameterized by a list of qualifiers $\overline{q}$ that correspond to the for-loop iterations, to be embedded in the comprehensions derived from the assignments in the loop body. This is always possible because of Theorem 3.1. That is, the for-loops in Equations (15d) and (15e) become qualifiers, which are propagated to the translation of their body s along with the current $\overline{q}$ (where $+\!\!+$ is list concatenation). While-loops, on the other hand, are translated to while-loop target statements in Equation (15f) because they are not parallelized. The qualifiers $\overline{q}$ are propagated to every statement in a block, as shown in Equation (15h). Equations (15a) and (15b) translate assignments. An incremental update $d \oplus\!= e$, equal to $d := d \oplus e$, is translated by Equation (15a). All other assignments are translated by Equation (15b). Both Equations (15a) and (15b) use the semantic function $\mathcal{K}$ that derives the destination indexes of the assignment, and the semantic function $\mathcal{U}$ that generates the update associated with the assignment. More specifically, $\mathcal{U}[\![d]\!](x)$ replaces the destination d with the value x by reconstructing the destination variable from its components, replacing the components reachable from d with x. For example, $\mathcal{U}[\![V[1]]\!](\{(1,10)\})$, which is equal to $[\, V := V \triangleleft \{(1,10)\} \,]$, updates V to be equal to V but with $V[1]$ replaced with 10. The incremental update $d \oplus\!= e$ is translated by Equation (15a) to a comprehension with a group-by over the destination index d and an aggregation $\oplus/v$ of all e values associated with the same group-by key. The value w added to the aggregation is the initial value of d before the loop. This value cannot be computed from $\mathcal{E}[\![d]\!]$ because it is correlated to the destination index k. Instead, it is derived from k using the semantic function $\mathcal{D}$.

Theorem A.1 in Appendix A proves that the transformation rules in Figure 2 under the restrictions in Definition 3.1 are meaning preserving.

3.9 Examples of Program Translation

First consider the following statement s that consists of a non-incremental update in a for-loop:

$$\textbf{for } i = 1,10 \textbf{ do } V[i] := W[i].$$

It is translated as follows:

$$\begin{array}{l}
\mathcal{S}[\![s]\!]([\ ]) \\
\quad \textit{(from Equation (15d))} \\
= \mathcal{S}[\![V[i] := W[i]]\!]([\, v_1 \leftarrow \mathcal{E}[\![1]\!],\ v_2 \leftarrow \mathcal{E}[\![10]\!], \\
\qquad\qquad i \leftarrow \text{range}(v_1, v_2)\,]) \\
\quad \textit{(using Equation (11g) and after normalization)} \\
= \mathcal{S}[\![V[i] := W[i]]\!]([\, i \leftarrow \text{range}(1,10)\,]) \\
\quad \textit{(from Equation (15b))} \\
= \mathcal{U}[\![V[i]]\!](\{\, (k,v) \mid i \leftarrow \text{range}(1,10),\ v \leftarrow \mathcal{E}[\![W[i]]\!], \\
\qquad\qquad k \leftarrow \mathcal{K}[\![V[i]]\!]\,\}) \\
\quad \textit{(from Equations (11c) and (12c))} \\
= \mathcal{U}[\![V[i]]\!](\{\, (k,v) \mid i \leftarrow \text{range}(1,10), \\
\qquad\qquad v \leftarrow \{\, w \mid (j,w) \leftarrow W,\ j = i \,\},\ k \leftarrow \{\, i \,\}\,\}) \\
\quad \textit{(after normalization)} \\
= \mathcal{U}[\![V[i]]\!](\{\, (i,w) \mid i \leftarrow \text{range}(1,10),\ (j,w) \leftarrow W,\ j = i \,\}) \\
\quad \textit{(from Equation (14c))} \\
= [\, V := V \triangleleft \{\, (i,w) \mid i \leftarrow \text{range}(1,10),\ (j,w) \leftarrow W,\ j = i \,\}\,] \\
\quad \textit{(after eliminating the loop iteration)} \\
= [\, V := V \triangleleft \{\, (i,w) \mid (i,w) \leftarrow W,\ \text{inRange}(i,1,10) \,\}\,].
\end{array}$$

Note that, the assignment $V := V \triangleleft \ldots$ is done in parallel, such as replacing an RDD with another RDD in Spark. Consider the following loop s with an incremental update:

$$\textbf{for } i = 1,10 \textbf{ do } W[K[i]] \mathrel{+}= V[i].$$

Then, from Equation (15d), $\mathcal{S}[\![s]\!]([\ ])$ is equal to:

$$\begin{array}{l}
\mathcal{S}[\![W[K[i]] \mathrel{+}= V[i]]\!]([\, v_1 \leftarrow \mathcal{E}[\![1]\!],\ v_2 \leftarrow \mathcal{E}[\![10]\!], \\
\qquad\qquad i \leftarrow \text{range}(v_1, v_2)\,]) \\
= \mathcal{S}[\![W[K[i]] \mathrel{+}= V[i]]\!]([\, i \leftarrow \text{range}(1,10)\,]).
\end{array}$$

To translate $W[K[i]] \mathrel{+}= V[i]$ using Equation (15a), we need to derive the destination index using Equation (12c):

$$\mathcal{K}[\![W[K[i]]]\!] = \mathcal{E}[\![K[i]]\!] = \{\, a \mid (m,a) \leftarrow K,\ m = i \,\}$$

and the destination value from the destination index using Equation (13c):

$$\mathcal{D}[\![W[K[i]]]\!](k) = \{\, v \mid (i,v) \leftarrow W,\ i = k \,\}.$$

Hence, the loop translation is:

$$\begin{array}{l}
\mathcal{S}[\![W[K[i]] \mathrel{+}= V[i]]\!]([\, i \leftarrow \text{range}(1,10)\,]) \\
\quad \textit{(from Equation (15a))} \\
= \mathcal{U}[\![W[K[i]]]\!](\{\, (k, w + (+/v)) \mid i \leftarrow \text{range}(1,10), \\
\quad (l,v) \leftarrow V,\ l = i,\ k \leftarrow \mathcal{K}[\![W[K[i]]]\!], \\
\quad \textbf{group by } k,\ w \leftarrow \mathcal{D}[\![W[K[i]]]\!](k)\,\})
\end{array}$$

$$
\begin{aligned}
&= \mathcal{U}[\![W[K[i]]]\!](\{\,(k, w + (+/v)) \mid i \leftarrow \text{range}(1,10),\\
&\quad (l, v) \leftarrow V,\ l = i,\ k \leftarrow \{\,a \mid (m, a) \leftarrow K,\ m = i\,\},\\
&\quad \textbf{group by}\ k,\ w \leftarrow \{\,v \mid (i, v) \leftarrow W,\ i = k\,\}\,\})\\
&= \mathcal{U}[\![W[K[i]]]\!](\{\,(k, w + (+/v)) \mid i \leftarrow \text{range}(1,10),\\
&\quad (l, v) \leftarrow V,\ l = i,\ (m, a) \leftarrow K,\ m = i,\\
&\quad \textbf{group by}\ a,\ (j, w) \leftarrow W,\ j = a\,\})\\
&\quad \textit{(from Equation (14c))}\\
&= [\,W := W \triangleleft \{\,w + (+/v) \mid i \leftarrow \text{range}(1,10),\\
&\qquad (l, v) \leftarrow V,\ l = i,\ (m, a) \leftarrow K,\ m = i,\\
&\qquad \textbf{group by}\ a,\ (j, w) \leftarrow W,\ j = a\,\}\,],
\end{aligned}
$$

which is optimized to the following target code after removing the loop iteration:

$$
\begin{aligned}
[\,W := W \triangleleft \{\,&w + (+/v) \mid (i, v) \leftarrow V,\\
&\text{inRange}(i, 1, 10),\ (m, a) \leftarrow K,\ m = i,\\
&\textbf{group by}\ a,\ (j, w) \leftarrow W,\ j = a\,\}\,].
\end{aligned}
$$

4 OPTIMIZATIONS

As discussed in Section 3, incremental updates on variables of a basic type, such as n += $W[i]$, can be translated to total aggregations. This translation is actually an optimization of the default translation. The optimization rule, for a constant group-by key c, is:

$$
\begin{aligned}
&\{\,e \mid \overline{q_1},\ \textbf{group by}\ p : c,\ \overline{q_2}\,\} \qquad (16)\\
&\quad \rightarrow\ \{\,e \mid \textbf{let}\ p = c,\ \forall v_i : \textbf{let}\ v_i = \{\,v_i \mid \overline{q_1}\,\},\ \overline{q_2}\,\},
\end{aligned}
$$

where v_i are the pattern variables in $\overline{q_1}$. For example, consider the assignment n += $W[i]$, which is translated to:

$$
n := \{\,n + (+/w) \mid (i, w) \leftarrow W,\ \textbf{group by}\ k : ()\,\}.
$$

The right-hand side of this assignment is optimized to:

$$
\begin{aligned}
&\{\,n + (+/w) \mid \textbf{let}\ k = (),\ \textbf{let}\ w = \{\,w \mid (i, w) \leftarrow W\,\}\,\}\\
&\quad =\ \{\,n + (+/\{\,w \mid (i, w) \leftarrow W\,\})\,\},
\end{aligned}
$$

which is more efficient because it does not use a group-by. The same happens when indexes in the destination are constants, such as in $M[1,2]$ += 1. Then, the group-by on $(1,2)$ can be removed using Rule (16):

$$
\begin{aligned}
M \triangleleft \{\,&(k, v + (+/c) \mid \textbf{let}\ c = 1,\ \textbf{group by}\ k : (1,2),\\
&\qquad ((i,j), v) \leftarrow M,\ i = 1,\ j = 2\,\}\\
=\ M \triangleleft \{\,&(k, v + (+/c) \mid \textbf{let}\ k = (1,2),\\
&\qquad \textbf{let}\ c = \{\,c \mid \textbf{let}\ c = 1\,\},\\
&\qquad ((i,j), v) \leftarrow M,\ i = 1,\ j = 2\,\}\\
=\ M \triangleleft \{\,&((1,2), v + 1) \mid ((i,j), v) \leftarrow M,\ i = 1,\ j = 2\,\}.
\end{aligned}
$$

Another optimization is when the group-by key is unique, that is, when the group-by function is injective. In that case, each group is a singleton bag. The group-by can be eliminated using the following rule:

$$
\begin{aligned}
&\{\,e \mid \overline{q_1},\ \textbf{group by}\ p : k,\ \overline{q_2}\,\} \qquad (17)\\
&\quad \rightarrow\ \{\,e \mid \overline{q_1},\ \textbf{let}\ p = k,\ \forall v_i : \textbf{let}\ v_i = \{v_i\},\ \overline{q_2}\,\}.
\end{aligned}
$$

That is, the group-by is removed and every pattern variable v_i in $\overline{q_1}$ is lifted to a singleton bag that represents the group, that is, it contains v_i only. For example, the loop:

$$
\textbf{for}\ i = 1, 10\ \textbf{do}\ V[i] \mathrel{+}= W[i]
$$

has a default translation, after removing the for-loop:

$$
\begin{aligned}
V \triangleleft \{\,&(k, v + (+/w)) \mid (i, w) \leftarrow W,\ \text{inRange}(i, 1, 10),\\
&\textbf{group by}\ k : i,\ (j, v) \leftarrow V,\ j = k\,\}.
\end{aligned}
$$

Here, the group-by key is unique since it is the index of W. Based on Rule (17), this term is optimized to:

$$
\begin{aligned}
V \triangleleft \{\,&(k, v + (+/w)) \mid (i, w) \leftarrow W,\ \text{inRange}(i, 1, 10),\\
&\textbf{let}\ k = i,\ \textbf{let}\ w = \{w\},\ (j, v) \leftarrow V,\ j = k\,\}\\
= V \triangleleft \{\,&(i, v + w) \mid (i, w) \leftarrow W,\ \text{inRange}(i, 1, 10),\\
&(j, v) \leftarrow V,\ j = i\,\}.
\end{aligned}
$$

Inferring whether a group-by key is unique is similar to inferring whether an assignment destination is affine (Section 3.2). A generator $(i, w) \leftarrow W$ for an array W indicates that i is unique. If the group-by key is an affine term that consists of all array indexes in the generators before the group-by, then it is a unique key.

5 PACKING/UNPACKING ARRAYS

In our framework, sparse arrays are an abstract representation of real arrays that may have been stored and partitioned into various custom dense storage structures. This separation of representation from implementation simplifies the language semantics by abstracting the implementation details from programs. More importantly, it makes easier to change the implementation without changing the programs. But this separation may introduce one more level of interpretation needed for restructuring data when loading storage structures into sparse arrays (unpacking) and storing sparse arrays to storage structures (packing). Our framework though can remove this extra layer of interpretation without any fundamental extension to the framework. In this paper, we discuss only matrices.

In our framework, a sparse matrix of type matrix[T] is represented as {((long, long), T)}, which contains the matrix elements in sparse form. One example of a concrete implementation of a matrix is organizing the matrix elements into equal sized chunks, called tiles [36, 38, 40], where each tile is a dense array of elements. This is called a tiled matrix. One possible implementation of a tiled matrix is {((long, long), Array[T])}, which is a bag of tiles where each

tile has an upper-left coordinate index and a Array[T] which is a dense vector that contains the matrix elements that belong to this tile. The translation from dense to sparse vector, and vice versa, can be defined as follows in Scala:

```
def scan(V) = V.zipWithIndex.map(_.swap)
def form(L,n) = { val a = new Array(n)
                  for ( (i,v) <- L
                        if i>=0 && i<=n ) a(i)=v
                  a }
```

where scan(V) converts the dense vector $V = [\, v_1, \ldots, v_n \,]$ into the sparse vector $\{(0, v_1), \ldots, (n-1, v_n)\}$ and form(L, n) converts the sparse vector L to a dense vector of size n. A tile is the unit of distributed processing. If, in addition, we use a Scala parallel collection, such as ParArray, to store the dense vector in a tile, then we would have thread-level data parallelism along with distributed data parallelism.

Suppose that the tiles are of size $n * m$. We can map a tiled matrix N to a sparse matrix using the function unpack(N):

$$\{\, ((I + k/m,\ J + k\%m),\ v) \mid ((I, J), L) \leftarrow N, \\ (k, v) \leftarrow \text{scan}(L) \,\}$$

We can map a sparse matrix M to a tiled matrix using the function pack(M):

$$\{\, ((I * n,\ J * m),\ \text{form}(z, n * m)) \mid ((i, j), v) \leftarrow M, \\ \textbf{let } z = (i + j * n, v),\ \textbf{group by } (I : i/n, J : j/m) \,\}$$

Under these mappings, $N[i, j]$ is translated to:

$$\{\, v \mid ((I, J), v) \leftarrow \text{unpack}(N),\ I = i,\ J = j \,\} \\ = \{\, v \mid ((I, J), L) \leftarrow N,\ (k, v) \leftarrow \text{scan}(L),\ I = i,\ J = j \,\}$$

which directly traverses the data in the matrix tiles. Assignments to a matrix N, which are normally translated to $N := N \lhd x$ by Equation (14c), are now translated to $N := \text{pack}(\text{unpack}(N) \lhd x)$. This expensive unpacking and packing of N can be removed by transforming this assignment to $N := N \lhd' \text{pack}(x)$, where $\lhd'$ merges two tiled matrices. The tile merging in $N \lhd' \text{pack}(x)$ can be implemented without shuffling if we keep every matrix partitioned by the tile coordinates and set the group-by partitioner in pack(x) to be equal to the N partitioner. Then the tile merging can be implemented using zipPartitions in Spark, which does not require any shuffling.

6 PERFORMANCE EVALUATION

DIABLO is implemented on top of DIQL [13, 22], which is a query optimization framework that optimizes and compiles queries to Java byte code at compile-time. DIQL can run on Apache Spark, Apache Flink, Cascading/Scalding, and Scala Parallel collections. DIABLO compiles loop-based programs to monoid comprehensions, which in turn are translated to byte code by the DIQL compiler. DIABLO is currently implemented on Spark, Flink, and on Scala's Parallel Collections. The translator from loop-based programs to monoid comprehensions is less than 500 lines of Scala code, while the normalizer and optimizer of comprehensions are also less than 500 lines total. The DIABLO code is available as part of the DIQL source code on GitHub [13]. The subdirectory `benchmarks/diablo` in the source code contains all the benchmark programs, the scripts, and the detailed execution logs with all the measurements derived from our performance evaluations (the file README.md explains how to repeat these experiments).

Our translation scheme is general, since it can translate any loop-based program that satisfies our restrictions, and efficient, since it uses simple program transformations, instead of searching to match specific program templates. We first evaluated the translator efficiency of DIABLO relative to MOLD [37] and Casper [2] (Table 1). The programs used in these evaluations are described next in this section. The translation times for MOLD were taken directly from the MOLD paper [37] but are not verified, because at the time of writing, we could not install MOLD due to software dependency issues. In addition, although both the binaries and source code of Casper are available at [10], we were not able to validate some of the results reported in [2]. More specifically, based on our communication with the main developer of Casper, we tried many configurations and libraries, but were not able to compile some of the test files provided with the source code. The results reported here were run on Casper 0.1.1, with Sketch 1.7.5 and Dafny 1.9.7. These experiments were done on a 2.7 GHz Intel Core i5 with 8GB RAM. Each program was run 4 times. Casper was able to synthesize code for Histogram but its validator failed to validate the code. For Linear Regression, Casper was taking too long so we had to abort it after 19 hours. The fail entries in Table 1 are failures to synthesize code for the test files; these errors were reported by the Dafny program synthesizer. We can see that the DIABLO translator is far more efficient than both MOLD and Casper and, unlike these systems, can translate complex programs. In fact, Casper can only translate trivial flat loops.

Although the focus of our work is on distributed processing, not shared-memory data parallelism, our second set of experiments was to evaluate a variety of loop-based programs in two ways: in parallel using Scala's parallel collections and sequentially using regular lists. That is, each one of these loop-based programs was compiled to parallel and to sequential Scala programs, and these two programs were evaluated over the same data. Scala uses thread-level shared-memory data parallelism on a multi-core computer to process parallel collections. For these evaluations, we used one server with Xeon E5-2680v3 at 2.5GHz, with 24 cores and 128GB RAM. The results, shown in Table 2, are based on

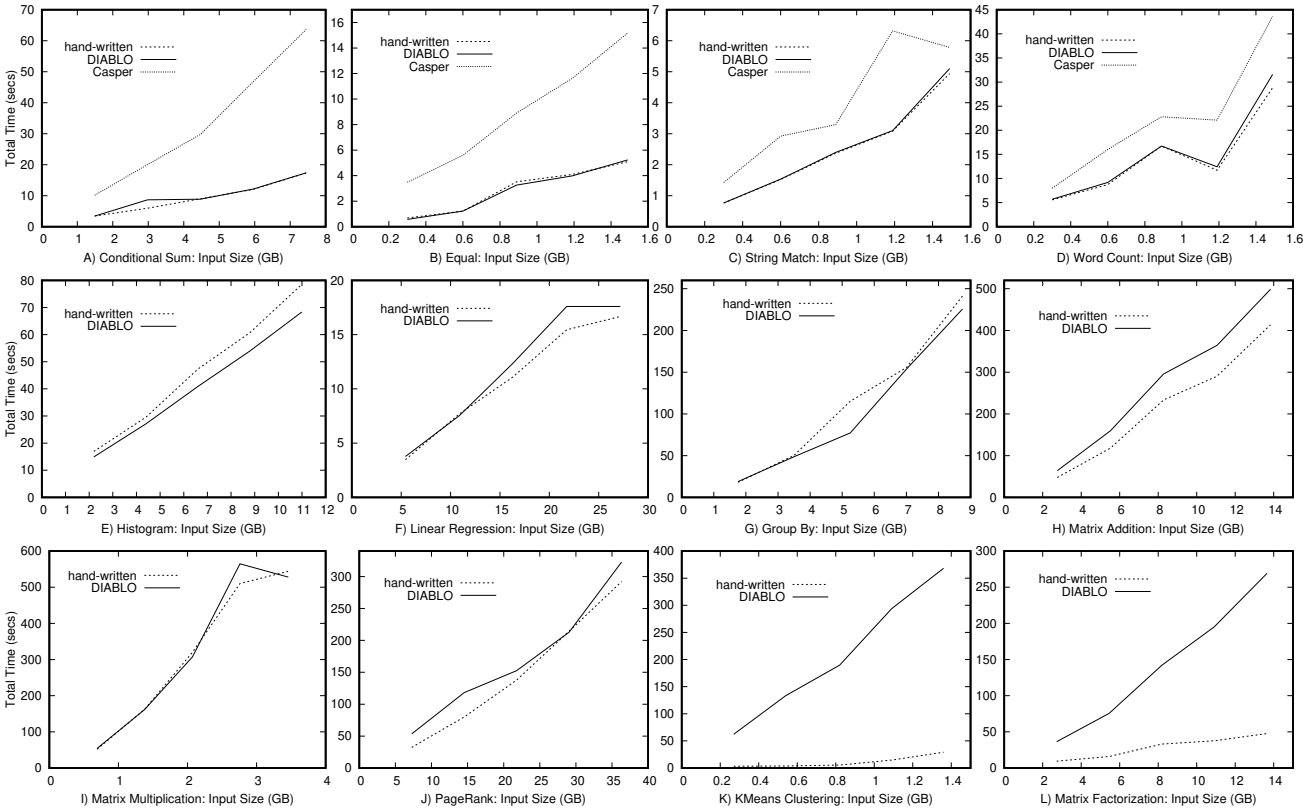

Figure 3: Performance of DIABLO relative to hand-written Spark code

Table 1: Compilation time in seconds

test program	MOLD	Casper	DIABLO
Average		172.25	5.75
Conditional Count		20.25	5.75
Conditional Sum		18.75	5.25
Count		9.75	5.75
Equal		11.25	5.75
Equal Frequency		778.00	5.75
String Match	68	806.00	8.50
Sum		10.25	5.00
Word Count	11	102.25	6.50
Histogram	233	10272.00	9.00
Matrix Multiplication	40	fail	8.25
Linear Regression	28	>19 hours	8.75
KMeans	340	fail	9.75
PCA	66	fail	13.25
PageRank			9.50
Matrix Factorization			14.50

Table 2: Parallel (par) vs Sequential (seq) evaluation time in seconds

test program	count	size (MB)	par	seq
Conditional Sum	10^9	61035	9.4	39.2
Equal	5×10^8	20504	8.3	36.2
String Match	5×10^8	20504	6.9	31.9
Word Count	5×10^7	2050	29.5	97.7
Histogram	5×10^7	3338	7.1	31.5
Linear Regression	10^8	13924	8.6	25.2
Group-By	5×10^7	2766	30.6	73.8
Matrix Addition	3500×3500	2710	28.9	154.0
Matrix Multiplication	420×420	39	19.7	179.3
PageRank	1500000	279	9.5	37.1
KMeans	500000	70	27.2	30.5
Matrix Factorization	980×980	210	9.6	24.3

the programs and data described next in this section. Each experiment was evaluated 4 times and the mean value was used. We can see that all DIABLO parallel programs are faster than their sequential counterparts.

To evaluate the quality of our generated code on a distributed platform, we have tested our system on 12 programs and compared their evaluation against efficient hand-written programs on Spark. The platform used for our evaluations was a small cluster of 10 nodes built on the XSEDE Comet cloud computing infrastructure at SDSC (San Diego Supercomputer Center). Each Comet node has one Xeon E5-2680v3 at 2.5GHz, with 24 cores, 128GB RAM, and 320GB SSD. For our experiments, we used Apache Spark 2.2.0 running on Apache Hadoop 2.6.0. All experiments were done on random data, stored in Spark RDDs. Each Spark executor was configured to have 4 cores and 23 GB RAM. Consequently,

there were 5 executors per node, giving a total of 50 executors, from which 2 were reserved for other tasks. Each program was evaluated over 5 datasets and each evaluation was repeated 4 times, the first of which was discarded to make sure that the JVM/JIT warm-up time does not skew the results. Hence, each data point in the plots in Figure 3 represents the mean value of these 3 evaluations. The dataset size was calculated by multiplying the dataset length by the size of a dataset element when is serialized to bytes using Java standard serialization. For example, a sparse matrix of type RDD[((Long,Long),Double)] with 1000 elements has size $1000 * 234$ bytes because ((Long,Long),Double) is serialized to 234 bytes. Each program was evaluated in 3 different ways: as a loop-based program translated by DIABLO to the Spark Core API (RDDs) (the lines tagged "DIABLO"), as an equivalent efficient program in Spark Core written by us (the lines tagged "hand-written"), and as a loop-based program translated by Casper to Spark Core, when such translation is possible (the lines tagged "Casper"). The programs are available on GitHub [13] and are given in Appendix B.

Conditional Sum filters a dataset V of type RDD[Double] that contains random data and aggregates the result. The Spark code is:

```
V. filter (_ < 100). reduce(_+_).
```

The largest dataset used had 10^9 elements and size 7.45 GB. Equal, String Match, and Word Count used the same dataset of type RDD[String] that contains random strings of size 4 so that there were 1000 different strings. The largest dataset used had 2×10^8 elements and size 1.49 GB. Equal checks whether all the strings in the dataset are equal. String Match checks whether the dataset contains "key1", "key2", or "key3". For each different string in the dataset, WordCount counts how many times this string occurs. Histogram scans a dataset P of RGB pixels of type RDD[(Int,Int,Int)], and for each one of the RGB components, it creates a histogram. For instance, the Spark code for the red component is:

```
P.map(_.1). countByValue().
```

The largest dataset used had 2×10^8 elements and size 10.99 GB. Linear Regression takes a dataset of 2-D points of type RDD[(Double,Double)] and calculates the intercept and the slope coefficient that models the dataset. The data used were points $(x + dx, x - dx)$, where x is a random double between 0 and 1000 and dx is a random double between 0 and 10. The largest dataset used had 2×10^8 elements and size 27.99 GB. Group By groups a dataset of type RDD[(Long,Double)] by it first component and sums up the second component. The keys were random long integers with 10 duplicates on the average. The largest dataset used had 2×10^8 elements and size 8.75 GB. We can see that programs generated by DIABLO have performance comparable to the hand-written programs and are faster than those by Casper.

Matrix addition and multiplication: The matrices used in our experiments have type RDD[((Long,Long),Double)]. Although sparse, all matrix elements were provided, were placed in random order, and were filled with random values between 0.0 and 10.0. The DIABLO matrix multiplication program is given in the Introduction, while the hand-written Spark program is as follows:

```
M.map{ case ((i,j),m) => (j,(i,m)) }
 .join( N.map{ case ((i,j),n) => (i,(j,n)) } )
 .map{ case (k,((i,m),(j,n))) => ((i,j),m*n) }
 .reduceByKey(_+_)
```

The matrices used for addition and multiplication were pairs of square matrices of the same size. The largest matrices used in addition had 8000×8000 elements and size 13.83 GB each, while those in multiplication had 4000×4000 elements and size 3.46 GB each. The results are shown in Figures 3.H and I. We can see that here too programs generated by DIABLO have performance comparable to the hand-written programs.

PageRank: The PageRank program computes one iteration of the page-rank algorithm that assigns a rank to each vertex of a graph, which measures its importance relative to the other vertices in the graph. The graphs used in our experiments were synthetic data generated by the RMAT (Recursive MATrix) Graph Generator [11] using the Kronecker graph generator parameters a=0.30, b=0.25, c=0.20, and d=0.25. The number of edges generated were 10 times the number of graph vertices. The largest graph used had 2×10^7 vertices, 2×10^8 edges, and had size 36.32 GB. The results are shown in Figure 3.J. The pagerank step in the hand-written program was simply a join between the graph and the current pagerank, followed by a reduceByKey. The generated DIABLO program though used a triple join among the graph, the current pagerank, and the node fan-out vector, followed by a reduceByKey.

K-Means clustering: The KMeans program computes one iteration step of the K-Means clustering algorithm, which finds the K centroids of a set of 2-D points on a plane. The datasets used in our experiments are random points on a plane inside a 10×10 grid of squares, where each square has a top-left corner at $(i*2+1, j*2+1)$ and bottom-right corner at $(i*2+2, j*2+2)$, for $i \in [0,9]$ and $j \in [0,9]$. That is, there should be 100 centroids, which are the square centers $(i*2+1.5, j*2+1.5)$. The initial centroids were set to be the points $(i*2+1.2, j*2+1.2)$. The largest dataset used had 10^7 data points and size 1.36 GB. The results are shown in Figure 3.K. The hand-written program broadcasts the initial centroids to all workers so that each worker keeps a copy in its memory, and then uses a map followed by a reduceByKey, in which the shuffled data were very small and of constant

size. On the other hand, DIABLO stores the centroids into an RDD and uses Spark joins to correlate points with centroids, making the entire process expensive.

Matrix factorization: The last program to evaluate is one iteration of matrix factorization using gradient descent [30]. The loop-based program was given in Section 3.2. For our experiments, we used the learning rate $a = 0.002$ and the normalization factor $b = 0.02$. The matrix to be factorized, R, was a square sparse matrix $n * n$ with random integer values between 1 and 5, in which only the 10% of the elements were provided (the rest were implicitly zero). The derived matrices P and Q had dimensions $n * 2$ and $2 * n$, respectively, and were initialized with random values between 0.0 and 1.0. The largest matrix R used had 8000×8000 elements and size 13.65 GB. The results are shown in Figure 3.L.

From these experiments, we can see that, except K-Means and Matrix Factorization, the programs generated by DIABLO have performance comparable to the hand-written programs. K-Means and Matrix Factorization are far more complex than the other programs, causing DIABLO to generate some unnecessary joins. These joins could have been eliminated by a more sophisticated query optimizer. The focus of our current work is on generating correct DISC programs from array loops. We are planning to explore more effective query optimization techniques in a future work.

7 CONCLUSION

We have addressed the problem of automated parallelization of array-based loops by translating them to comprehensions, which can then be translated and optimized to distributed data parallel operations. The efficiency of our translations would mostly depend on the effectiveness of code optimization after translation, which we are planning to address more thoroughly in a future work. We are also planning to look at cost-based optimizations, such as determining whether an array is small enough to fit in a worker's memory in order to broadcast it to all workers, thus speeding up joins over this array. For example, the centroid vector in the K-means clustering example was small enough to broadcast. One source of inefficiency in our translations is the large number of generated joins. When two arrays are used together in a program, such as in $A[i] * B[i]$, this term is translated to a join between A and B. This join can be avoided if we co-partition these two vectors using the same partitioner. Then, $A[i] * B[i]$ can be implemented using the zipPartitions operation in Spark, which does not cause any shuffling. As a future work, we are also planning to experiment with more platforms as the target of DIABLO, such as Spark SQL, which supports cost-based optimizations. A more effective way to represent arrays is to encode them as distributed bags of tiles, where each tile is a fixed-size array chunk. It is well known in ML and data management communities that such tiled representations largely outperform basic sparse tuple representations. As a future work, we are planning to extend our framework to generate code that processes tiled arrays. Finally, as a future work, we want to investigate how to generate optimal parallel algorithms from loops, such as the SUMMA algorithm [24] for distributed matrix multiplication. For such optimizations, a template-based approach, where a generic template of algebraic operations is mapped to an algorithm, may be a more suitable solution.

Acknowledgments: Our evaluations were performed at the XSEDE Comet cloud computing infrastructure at the San Diego Supercomputer Center (SDSC), www.xsede.org, supported by NSF.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, *et al.* TensorFlow: a system for large-scale machine learning. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] M. B. S. Ahmad and A. Cheung. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *ACM SIGMOD International Conference on Management of Data*, pages 1205–1220, 2018.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). *Chapter 11: Optimizing for Parallelism and Locality*, Addison Wesley, 2007.

[4] Apache Flink. Available: http://flink.apache.org/, 2020.

[5] Apache Hadoop. Available: http://hadoop.apache.org/, 2020.

[6] Apache Spark. Available: http://spark.apache.org/, 2020.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[8] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. In *Journal of Parallel and Distributed Computing (JPDC)*, 8:119–134, 1990.

[9] J. Buck, N. Watkins, J. Lefevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. A. Brandt. SciHadoop: Array-based Query Processing in Hadoop. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[10] Casper. Available: http://casper.uwplse.org/, accessed in January 2020.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining (SDM)*, pages 442–446, 2004.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[13] DIQL: A Data Intensive Query Language. Available: https://github.com/fegaras/DIQL, 2020.

[14] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *ACM SIGMOD International Conference on Management of Data*, pages 1781–1796, 2016.

[15] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing Sequential Graph Computations. In *ACM SIGMOD International Conference on Management of Data*, pages 495–510, 2017.

$\mathcal{E}'[\![e]\!]_\sigma$: Translate the expression e to a a comprehension term

$$\mathcal{E}'[\![V]\!]_\sigma = \begin{cases} \sigma(V) & \text{if } V \in \sigma \\ \{\, V \,\} & \text{otherwise} \end{cases} \tag{18a}$$

$$\mathcal{E}'[\![e]\!]_\sigma = \mathcal{E}[\![e]\!] \quad \textit{(as defined in Figure 2)} \tag{18b}$$

$\mathcal{U}'[\![d]\!]_\sigma(x)$: Transform the state σ by replacing the destination d with the value x

$$\mathcal{U}'[\![V]\!]_\sigma(x) = \sigma[V = x] \tag{19a}$$

$$\mathcal{U}'[\![d.A_i]\!]_\sigma(x) = \mathcal{U}'[\![d]\!]_\sigma(\{\, \langle A_1 = w.A_1, \ldots, A_i = v, \ldots, A_n = w.A_n \rangle \mid w \leftarrow \mathcal{E}'[\![d]\!]_\sigma,\ v \leftarrow x \,\}) \tag{19b}$$

$$\mathcal{U}'[\![V[e]]\!]_\sigma(x) = \sigma[V = \{\, V \lhd \{(i, v)\} \mid i \leftarrow \mathcal{E}'[\![e]\!]_\sigma,\ v \leftarrow x \,\}] \tag{19c}$$

$$\mathcal{U}'[\![V[e_1, e_2]]\!]_\sigma(x) = \sigma[V = \{\, V \lhd \{((i, j), v)\} \mid i \leftarrow \mathcal{E}'[\![e_1]\!]_\sigma,\ j \leftarrow \mathcal{E}'[\![e_2]\!]_\sigma,\ v \leftarrow x \,\}] \tag{19d}$$

$\mathcal{T}[\![s]\!]_\sigma$: Translate the parallelizable program s to a term that transforms the state σ

$$\mathcal{T}[\![d \oplus= e]\!]_\sigma = \mathcal{T}[\![d := d \oplus e]\!]_\sigma \tag{20a}$$

$$\mathcal{T}[\![d := e]\!]_\sigma = \mathcal{U}'[\![d]\!]_\sigma(\mathcal{E}'[\![e]\!]_\sigma) \tag{20b}$$

$$\mathcal{T}[\![\textbf{for } v = e_1, e_2 \textbf{ do } s]\!]_\sigma = (\{\, \lambda x.\, \mathcal{T}[\![s]\!]_x \mid v_1 \leftarrow \mathcal{E}'[\![e_1]\!]_\sigma,\ v_2 \leftarrow \mathcal{E}'[\![e_2]\!]_\sigma,\ v \leftarrow \text{range}(v_1, v_2) \,\}_\circ)\, \sigma \tag{20c}$$

$$\mathcal{T}[\![\textbf{for } v \textbf{ in } e \textbf{ do } s]\!]_\sigma = (\{\, \lambda x.\, \mathcal{T}[\![s]\!]_x \mid A \leftarrow \mathcal{E}'[\![e]\!]_\sigma,\ (i, v) \leftarrow A \,\}_\circ)\, \sigma \tag{20d}$$

$$\mathcal{T}[\![\textbf{if } (e)\ s_1 \textbf{ else } s_2]\!]_\sigma = \{\, x \mid p \leftarrow \mathcal{E}'[\![e]\!]_\sigma,\ x \leftarrow \textbf{if } p \textbf{ then } \mathcal{T}[\![s_1]\!]_\sigma \textbf{ else } \mathcal{T}[\![s_2]\!]_\sigma \,\} \tag{20e}$$

Figure 4: Semantics of a parallelizable program

[16] A. Farzan and V. Nicolet. Synthesis of Divide and Conquer Parallelism for Loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 540âĂŞ-555. 2017.

[17] A. Farzan and V. Nicolet. Modular Synthesis of Divide-and-Conquer Parallelism for Nested Loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[18] G. Fedyukovich, M. B. .S. Ahmad, and R. Bodik. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs, In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[19] L. Fegaras. A Query Processing Framework for Array-Based Computations. In *27th International Conference on Database and Expert Systems Applications (DEXA)*, 2016.

[20] L. Fegaras. An Algebra for Distributed Big Data Analytics. *Journal of Functional Programming*, special issue on Programming Languages for Big Data, Volume 27, 2017.

[21] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.

[22] L. Fegaras and M. H. Noor. Compile-Time Code Generation for Embedded Data-Intensive Query Languages. In *IEEE BigData Congress*, 2018.

[23] A. L. Fisher and A. M. Ghuloum. Parallelizing Complex Scans and Reductions. *ACM SIGPLAN Notices*, 29(6):135-âĂŞ146, 1994.

[24] R. A. Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. In *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[25] Y. Geng, X. Huang, M. Zhu, H. Ruan, and G. Yang. SciHive: Array-based query processing with HiveQL. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (Trustcom)*, 2013.

[26] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.

[27] R. Guravannavar and S. Sudarshan. Rewriting Procedures for Batched Bindings. *PVLDB*, 1(1):1107–1123, 2008.

[28] A. R. Hurson, J. T. Lim, K. M. Kavi, and B. Lee. Parallelization of DOALL and DOACROSS Loops – a Survey. *Advances in Computers*, vol 45, pages 53–103, 1997.

[29] P. Jiang, L. Chen, and G. Agrawal. Revealing Parallel Scans and Reductions in Recurrences through Function Reconstruction. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1âĂŞ-13, 2018.

[30] Y. Koren, R. Bell, and C. Volinsky. Matrix Factorization Techniques for Recommender Systems *IEEE Computer*, 42(8):30âĂŞ-37, August 2009.

[31] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. Franklin, and M.I. Jordan. MLbase: A Distributed Machine Learning System. In *Conference on Innovative Data Systems Research*, 2013.

[32] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB*, 12(11):1553-1567, 2020.

[33] X. Meng, J. Bradley, B. Yavuz, *et al.* MLlib: Machine Learning in Apache Spark. In *Journal of Machine Learning Research*, 17:1-7, 2016.

[34] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 146–155, 2007.

[35] D. W. Palmer, J. F. Prins, and S. Westfold. Work-Efficient Nested Data-Parallelism. In *Symposium on the Frontiers of Massively Parallel Processing*, 1995.

[36] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB array data storage manager. *PVLDB*, 10(4):349âĂŞ360, 2016.
[37] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan. Translating Imperative Code to MapReduce. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 909âĂŞ-927, 2014.
[38] The SciDB Development Team. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *ACM SIGMOD International Conference on Management of Data*, pages 963-âĂŞ968, 2010.
[39] C. Smith and A. Albarghouthi. MapReduce Program Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 326–340, 2016.
[40] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *ACM SIGMOD International Conference on Management of Data*, pages 253âĂŞ-264, 2011.
[41] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *27th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2015.
[42] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Antony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a Warehousing Solution over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.
[43] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall. Automating Wavefront Parallelization for Sparse Matrix Computations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Article No. 41, pages 1âĂŞ-12, 2016.
[44] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A novel MapReduce-like framework for multiple scientific data formats. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2012.
[45] 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 Systems Design and Implementation (OSDI)*, 2008.
[46] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

A CORRECTNESS PROOFS

Theorem 3.1. *An affine for-loop satisfies:*

$$\textbf{for } i = \ldots \textbf{ do } \{ s_1; s_2 \} = \{ \textbf{for } i = \ldots \textbf{ do } s_1; \textbf{ for } i = \ldots \textbf{ do } s_2 \} \quad (1)$$

Proof. Based on the exception (a) in Definition 3.1, if there is $d \in \mathcal{W}[\![s_1]\!]$ (ie, there is an update $d := e$ in s_1) and $d \in \mathcal{R}[\![s_2]\!]$ (ie, d is read in s_2), then according to restriction (1) in Definition 3.1, d must be affine. That is, the location of d is different for different values of the loop index i, which means that there is no interference across iteration steps. Hence, we can do all the updates $d := e$ first in one loop and then read all d values in a separate loop. Based on the exception (b) in Definition 3.1, if there is $d \oplus\!= e$ in s_1 and $d \in \mathcal{R}[\![s_2]\!]$, then affine$(d, s_2)$ and context$(s_1) \cap$ context(s_2) = indexes(d). That is, $i \in$ indexes(d) since both statements are inside the for-loop of i. Furthermore, d must be affine in the context of s_2, which contains i. Hence, like the previous case for an update to d, we can calculate all increments $d \oplus\!= e$ first in one loop and then read all d values in a separate loop. For all other cases, based on restriction (2) in Definition 3.1, there are no interferences between s_1 and s_2 and, therefore, the loop can be split into two loops. □

Theorem A.1 (Soundness). *The transformation rules in Figure 2 are meaning preserving for all programs that satisfy the recurrence restrictions in Definition 3.1.*

Proof. If we use Equation (1) in Theorem 3.1 as a rewrite rule from left to right, then any loop-based program can be put to a normal form that consists of a single sequential program ss that contains parallelizable for-loops ps:

Sequential Program:

ss	::=	**var** $v : t = e$	declaration
	|	**while** (e) ss	loop
	|	$\{ ss_1; \ldots; ss_n \}$	statement block
	|	ps	

Parallelizable Program:

ps	::=	$d \oplus\!= e$	incremental update
	|	$d := e$	assignment
	|	**for** $v = e_1, e_2$ **do** ps	iteration
	|	**for** v **in** e **do** ps	traversal
	|	**if** (e) ps_1 **else** ps_2	conditional

The sequential part generated by the rules in Figure 2 is equivalent to the derived sequential program ps. Therefore, to prove Theorem A.1, we need to prove that the derived parallelizable programs are equivalent to those generated by the rules in Figure 2. To prove this equivalence, we have to give formal semantics to the parallelizable programs in the form of monoid comprehensions and then prove the equivalence of the derived monoid comprehensions. The formal semantics of a parallelizable program is given by the rules in Figure 4. It is based on denotational semantics, which is often used to ascribe a formal meaning to imperative programming languages. In denotational semantics, to capture the meaning of an imperative program, we use a state σ to encapsulate all updatable variables. Then, a statement is translated to a state transformer, which is a function from the current state to a new state. If the statement does not cause any side effects, the state is propagated as is; otherwise, the state is replaced with a new state that reflects the updates. In our semantics, the state σ is a map from a variable name to a bag of values, where the bag can have 0 or 1 elements. Then, $\sigma(V)$ returns this bag and $\sigma[V = v]$ replaces the value of the variable V with the bag v.

The Rules (20a) and (20b) in Figure 4 translate assignments and incremental updates to state transformers using Rules (19a)-(19d), which replace the σ component associated with the destination d with a new value x. To capture the

sequential semantics of a for-loop, we use a comprehension $\{\!|\, f \mid \overline{q} \,|\!\}_\circ$ over the monoid $\circ$. The function composition $\circ$ satisfies $(f_2 \circ f_1)(x) = f_2(f_1(x))$, that is, it combines two state transformers. Hence, the comprehension $\{\!|\, f \mid \overline{q} \,|\!\}_\circ$ returns a function, which when applied to a state σ, it transforms σ to $f(f(\ldots f(\sigma)))$. This means that, if f is the state transformer associated with the body of a for-loop, then this comprehension captures the for-loop iteration.

We first prove the following theorem using structural induction on the statement s:

$$\Sigma_\sigma(\mathcal{S}[\![s]\!](\overline{q})) = (\{\!|\, \lambda x.\, \mathcal{T}[\![s]\!]_x \mid \overline{q} \,|\!\}_\circ)\,\sigma \tag{21}$$

where $\Sigma_\sigma([\, V_1 := v_1, \ldots, V_n := v_n \,]) = \sigma[V_1 = v_1, \ldots, V_n = v_n]$, that is, Σ_σ converts a list of updates to a state transformation. For the induction base case, we will only prove Equation (21) for the statement $V[e_1] \oplus= e_2$. (The other cases are easier to prove.) The left side is:

$$\begin{array}{l}
\Sigma_\sigma(\mathcal{S}[\![V[e_1] \oplus= e_2]\!](\overline{q})) \\
\quad \textit{(from Equation (15a))} \\
= \Sigma_\sigma(\mathcal{U}[\![V[e_1]]\!](\{\!|\, (k, w \oplus (\oplus/v)) \mid \overline{q},\ v \leftarrow \mathcal{E}[\![e_2]\!], \\
\quad k \leftarrow \mathcal{K}[\![V[e_1]]\!],\ \textbf{group by}\ k,\ w \leftarrow \mathcal{D}[\![V[e_1]]\!](k) \,|\!\})) \\
\quad \textit{(from Equation (14c))} \\
= \sigma[V = \{\!|\, \sigma.V \lhd \{\!|\, (k, w \oplus (\oplus/v)) \mid \overline{q},\ v \leftarrow \mathcal{E}[\![e_2]\!], \\
\quad k \leftarrow \mathcal{K}[\![V[e_1]]\!],\ \textbf{group by}\ k,\ w \leftarrow \mathcal{D}[\![V[e_1]]\!](k) \,|\!\} \,|\!\}] \\
\quad \textit{(from Equations (12c) and (13c))} \\
= \sigma[V = \{\!|\, \sigma.V \lhd \{\!|\, (k, w \oplus (\oplus/v)) \mid \overline{q},\ v \leftarrow \mathcal{E}[\![e_2]\!], \\
\quad k \leftarrow \mathcal{E}[\![e_1]\!],\ \textbf{group by}\ k,\ (i, w) \leftarrow \sigma.V,\ i = k \,|\!\} \,|\!\}] \\
= \sigma[V = \{\!|\, \sigma.V \lhd \{\!|\, (k, w \oplus (\oplus/s)) \mid (k, s) \leftarrow G, \\
\qquad\qquad (i, w) \leftarrow \sigma.V,\ i = k \,|\!\} \,|\!\}] \\
\quad \textit{(defined as:)} \\
= M(\sigma, G)
\end{array}$$

where the key-value map G is:

$$G = \{\!|\, (k, v) \mid \overline{q},\ v \leftarrow \mathcal{E}[\![e_2]\!],\ k \leftarrow \mathcal{E}[\![e_1]\!],\ \textbf{group by}\ k \,|\!\}$$

The right side of Equation (21) is:

$$\begin{array}{l}
(\{\!|\, \lambda x.\, \mathcal{T}[\![V[e_1] \oplus= e_2]\!]_x \mid \overline{q} \,|\!\}_\circ)\,\sigma \\
\quad \textit{(from Equations (20a) and (20b)} \\
\quad\ \textit{and after normalization)} \\
= (\{\!|\, \lambda x.\, \mathcal{U}'[\![V[e_1]]\!]_x(\mathcal{E}'[\![V[e_1] \oplus e_2]\!]_x) \mid \overline{q} \,|\!\}_\circ)\,\sigma \\
\quad \textit{(from Equations (19c) and (11c))} \\
= (\{\!|\, \lambda x.\, x[V = x.V \lhd \{\!|\, (j, w \oplus v) \mid j \leftarrow \mathcal{E}'[\![e_1]\!]_x, \\
\qquad k \leftarrow \mathcal{E}'[\![e_1]\!]_x,\ (i, w) \leftarrow x.V,\ i = k, \\
\qquad v \leftarrow \mathcal{E}'[\![e_2]\!]_x \,|\!\}] \mid \overline{q} \,|\!\}_\circ)\,\sigma \\
\quad \textit{(after removing the repeated generator} \\
\quad\ \textit{and moving the last generator)} \\
= (\{\!|\, \lambda x.\, x[V = x.V \lhd \{\!|\, (k, w \oplus v) \mid v \leftarrow \mathcal{E}'[\![e_2]\!]_x, \\
\quad k \leftarrow \mathcal{E}'[\![e_1]\!]_x,\ (i, w) \leftarrow x.V,\ i = k \,|\!\}] \mid \overline{q} \,|\!\}_\circ)\,\sigma \\
\quad (\mathcal{E}'[\![e_1]\!]_x = \mathcal{E}'[\![e_1]\!]_\sigma \textit{ and } \mathcal{E}'[\![e_2]\!]_x = \mathcal{E}'[\![e_2]\!]_\sigma \\
\quad\ \textit{because } e_1 \textit{ and } e_2 \textit{ do not interfere with } V) \\
= (\{\!|\, \lambda x.\, x[V = x.V \lhd \{\!|\, (k, w \oplus v) \mid v \leftarrow \mathcal{E}'[\![e_2]\!]_\sigma, \\
\quad k \leftarrow \mathcal{E}'[\![e_1]\!]_\sigma,\ (i, w) \leftarrow x.V,\ i = k \,|\!\}] \mid \overline{q} \,|\!\}_\circ)\,\sigma \\
= (\{\!|\, \lambda x.\, x[V = x.V \lhd \{\!|\, (k, w \oplus v) \mid (i, w) \leftarrow x.V,\ i = k \,|\!\}] \mid \\
\quad \overline{q},\ v \leftarrow \mathcal{E}'[\![e_2]\!]_\sigma,\ k \leftarrow \mathcal{E}'[\![e_1]\!]_\sigma \,|\!\}_\circ)\,\sigma \\
\quad \textit{(by unnesting the group-by } G) \\
= (\{\!|\, \lambda x.\, x[V = x.V \lhd \{\!|\, (k, w \oplus v) \mid (i, w) \leftarrow x.V,\ i = k \,|\!\}] \\
\quad \mid (k, s) \leftarrow G,\ v \leftarrow s \,|\!\}_\circ)\,\sigma \\
\quad \textit{(defined as:)} \\
= N(\sigma, G)
\end{array}$$

To prove that $M(\sigma, G) = N(\sigma, G)$, we use induction over G. It is easy to prove this equality for an empty and a unary G. For $G = G_1 \Updownarrow_\uplus G_2$, we assume the equality is true for G_1 and G_2 (induction hypotheses) and prove it for G (induction step):

$$\begin{array}{l}
M(\sigma, G) = M(\sigma, G_1 \Updownarrow_\uplus G_2) \\
= \sigma[V = \{\!|\, \sigma.V \lhd \{\!|\, (k, w \oplus (\oplus/s)) \mid (k, s) \leftarrow (G_1 \Updownarrow_\uplus G_2), \\
\qquad\qquad (i, w) \leftarrow \sigma.V,\ i = k \,|\!\} \,|\!\}] \\
= \sigma[V = \{\!|\, \sigma.V \lhd \{\!|\, (k, w \oplus (\oplus/s)) \mid (k, s) \leftarrow G_1, \\
\qquad\qquad (i, w) \leftarrow \sigma.V,\ i = k \,|\!\} \\
\qquad\qquad \lhd \{\!|\, (k, w \oplus (\oplus/s)) \mid (k, s) \leftarrow G_2, \\
\qquad\qquad (i, w) \leftarrow \sigma.V,\ i = k \,|\!\} \,|\!\}] \\
= (\lambda x.\, M(x, G_2))(M(\sigma, G_1)) \\
\quad \textit{(induction hypotheses)} \\
= (\lambda x.\, N(x, G_2))(N(\sigma, G_1)) \\
= N(\sigma, G_1 \Updownarrow_\uplus G_2) = N(\sigma, G)
\end{array}$$

We will now use the following law:

$$\{\!|\, \{\!|\, f \mid \overline{q_1} \,|\!\}_\circ \mid \overline{q_2} \,|\!\}_\circ = \{\!|\, f \mid \overline{q_2}, \overline{q_1} \,|\!\}_\circ \tag{22}$$

to prove Equation (21) for a statement **for** $v = e_1, e_2$ **do** s (induction step), assuming that it is true for s (induction

hypothesis):

$$
\begin{aligned}
&\Sigma_\sigma(\mathcal{S}[\![\textbf{for } v = e_1, e_2 \textbf{ do } s]\!](\overline{q})) \\
&\quad \textit{(from Equation (15d))} \\
&= \Sigma_\sigma(\mathcal{S}[\![s]\!](\overline{q} +\!\!+ [\, v_1 \leftarrow \mathcal{E}[\![e_1]\!],\ v_2 \leftarrow \mathcal{E}[\![e_2]\!], \\
&\qquad\qquad v \leftarrow \text{range}(v_1, v_2)\,])) \\
&\quad \textit{(induction hypothesis)} \\
&= \{\!|\, \lambda x.\, \mathcal{T}[\![s]\!]_x \mid \overline{q},\ v_1 \leftarrow \mathcal{E}[\![e_1]\!],\ v_2 \leftarrow \mathcal{E}[\![e_2]\!], \\
&\qquad\qquad v \leftarrow \text{range}(v_1, v_2) \,|\!\}_\circ\, \sigma \\
&\quad \textit{(from Equation (22))} \\
&= (\{\!|\, \lambda x.\, (\{\!|\, \lambda z.\, \mathcal{T}[\![s]\!]_z \mid v_1 \leftarrow \mathcal{E}'[\![e_1]\!]_x,\ v_2 \leftarrow \mathcal{E}'[\![e_2]\!]_x, \\
&\qquad\qquad\qquad v \leftarrow \text{range}(v_1, v_2) \,|\!\}_\circ)\, x \mid \overline{q} \,|\!\}_\circ)\, \sigma \\
&\quad \textit{(from Equation (20c))} \\
&= (\{\!|\, \lambda x.\, \mathcal{T}[\![\textbf{for } v = e_1, e_2 \textbf{ do } s]\!]_x \mid \overline{q} \,|\!\}_\circ)\, \sigma
\end{aligned}
$$

There is a similar proof for the other cases.

The correctness of Theorem A.1 comes directly from Equation (21) when $\overline{q}$ is empty:

$$\Sigma_\sigma(\mathcal{S}[\![s]\!]([\ ])) = (\{\!|\, \lambda x.\, \mathcal{T}[\![s]\!]_x \mid \,|\!\}_\circ)\, \sigma = \mathcal{T}[\![s]\!]_\sigma$$

which correctly captures the meaning of s. □

B BENCHMARK PROGRAMS

Conditional Sum in Spark:

```
V. filter ( _ < 100). reduce(_+_)
```

Conditional Sum in DIABLO:

```
var sum: Double = 0.0;

for v in V do
    if (v < 100)
        sum += v;
```

Equal in Spark:

```
val x = V. first ()
V.map(_ == x). reduce(_&&_)
```

Equal in DIABLO:

```
var eq: Boolean = true;

for v in V do
    eq := eq && v == x;
```

String Match in Spark:

```
val key1 = "key1"
val key2 = "key2"
val key3 = "key3"
words.map{ w => (w == key1)
                || (w == key2)
                || (w == key3) }
     .reduce(_ || _)
```

String Match in DIABLO:

```
var c: Boolean = false;

for w in words do
    c := c || (w == key1 || w == key2
               || w == key3);
```

Word Count in Spark:

```
words.map((_ ,1)). reduceByKey(_+_)
```

Word Count in DIABLO:

```
var C: map[String, Int] = map();

for w in words do
    C[w] += 1;
```

Histogram in Spark:

```
case class Color ( red: Int, green: Int, blue: Int )
val R = P.map(_.red). countByValue()
val G = P.map(_.green). countByValue()
val B = P.map(_.blue). countByValue()
```

Histogram in DIABLO:

```
var R: map[Int, Int] = map();
var G: map[Int, Int] = map();
var B: map[Int, Int] = map();

for p in P do {
    R[p.red] += 1;
    G[p.green] += 1;
    B[p.blue] += 1;
};
```

Linear Regression in Spark:

```
val x_bar = P.map(_._1). reduce(_+_)/n
val y_bar = P.map(_._2). reduce(_+_)/n

val xx_bar = P.map(x => (x._1−x_bar)*(x._1−x_bar))
              .reduce(_+_)
val yy_bar = P.map(y => (y._2−y_bar)*(y._2−y_bar))
              .reduce(_+_)
val xy_bar = P.map(p => (p._1−x_bar)*(p._2−y_bar))
              .reduce(_+_)
val slope = xy_bar/xx_bar
val intercept = y_bar − slope * x_bar
```

Linear Regression in DIABLO:

```
var sum_x: Double = 0.0;
var sum_y: Double = 0.0;
var x_bar: Double = 0.0;
var y_bar: Double = 0.0;
var xx_bar: Double = 0.0;
```

```
var yy_bar: Double = 0.0;
var xy_bar: Double = 0.0;
var slope: Double = 0.0;
var intercept: Double = 0.0;

for p in P do {
    sum_x += p._1;
    sum_y += p._2;
};

x_bar := sum_x/n;
y_bar := sum_y/n;

for p in P do {
    xx_bar += (p._1-x_bar)*(p._1-x_bar);
    yy_bar += (p._2-y_bar)*(p._2-y_bar);
    xy_bar += (p._1-x_bar)*(p._2-y_bar);
};

slope := xy_bar/xx_bar;
intercept := y_bar-slope*x_bar;
```

Group-by in Spark:

```
case class GB ( K: Long, A: Double )
V.map{ case GB(k,v) => (k,v) }. reduceByKey(_+_)
```

Group-by in DIABLO:

```
var C: vector[Double] = vector ();

for v in V do
    C[v.K] += v.A;
```

Matrix Addition in Spark:

```
M.join(N).mapValues{ case (m,n) => n + m }
```

Matrix Addition in DIABLO:

```
var R: matrix[Double] = matrix ();

for i = 0, n-1 do
    for j = 0, mm-1 do
        R[i,j] := M[i,j]+N[i,j];
```

Matrix Multiplication in Spark:

```
M.map{ case ((i,j),m) => (j,(i,m)) }
 .join( N.map{ case ((i,j),n) => (i,(j,n)) } )
 .map{ case (k,((i,m),(j,n))) => ((i,j),m*n) }
 .reduceByKey(_+_)
```

Matrix Multiplication in DIABLO:

```
var R: matrix[Double] = matrix ();

for i = 0, n-1 do
    for j = 0, n-1 do {
        R[i,j] := 0.0;
        for k = 0, mm-1 do
            R[i,j] += M[i,k]*N[k,j];
};
```

PageRank in Spark:

```
val links = E.map(_._1).groupByKey().cache()
var ranks = links.mapValues(v => 1.0/ vertices )

for (i <- 1 to num_steps) {
    val contribs
      = links.join(ranks).values.flatMap {
            case (urls, rank)
              => val size = urls.size
                 urls.map(url => (url, rank / size))
        }
    ranks = contribs.reduceByKey(_ + _)
              .mapValues(0.15/ vertices + 0.85 * _)
}
```

PageRank in DIABLO:

```
var P: vector[Double] = vector ();
var C: vector[Int] = vector ();
var N: Int = vertices ;
var b: Double = 0.85;

for i = 1, N do {
    C[i] := 0;
    P[i] := 1.0/N;
};

for i = 1, N do
    for j = 1, N do
        if (E[i,j])
            C[i] += 1;

var k: Int = 0;

while (k < num_steps) {
  var Q: matrix[Double] = matrix ();
  k += 1;
  for i = 1, N do
    for j = 1, N do
        if (E[i,j])
            Q[i,j] := P[i];
```

```
    for i = 1, N do
        P[i] := (1-b)/N;
    for i = 1, N do
        for j = 1, N do
            P[i] += b*Q[j,i]/C[j];
};
```

KMeans Clustering in Spark:

```
var centroids = initial_centroids

def distance ( x, y )
  = Math.sqrt((x._1-y._1)*(x._1-y._1)
              +(x._2-y._2)*(x._2-y._2))

case class Avg ( sum: (Double,Double), count: Long ) {
  def ^^ ( x: Avg ): Avg
    = Avg((sum._1+x.sum._1,sum._2+x.sum._2),count+x.count)
  def value (): (Double,Double)
    = (sum._1/count,sum._2/count)
}

case class ArgMin ( index: Long, distance: Double ) {
  def ^ ( x: ArgMin ): ArgMin
    = if (distance <= x.distance) this else x
}

for ( i <- 1 to num_steps )
    centroids
      = points.map{ p => (centroids.minBy(distance(p,_)),
                          Avg(p,1)) }
            .reduceByKey(_ ^^ _)
            .map(_._2.value())
            . collect ()
```

KMeans Clustering in DIABLO: C and Avg are defined as Scala Arrays.

```
var closest: vector[ArgMin] = vector();

var steps: Int = 0;
while (steps < num_steps) {
    steps += 1;
    for i = 0, N-1 do {
        closest[i] := ArgMin(0,10000.0);
        for j = 0, K-1 do
            closest[i]
              := closest[i]
                 ^ ArgMin(j,distance(P[i],C[j]));
        avg[closest[i].index]
          := avg[closest[i].index] ^^ Avg(P[i],1);
    };
    for i = 0, K-1 do
        C[i] := avg[i].value();
};
```

Matrix Factorization in Spark:

```
def transpose ( x )
  = x.map{ case ((i,j),v) => ((j,i),v) }

def op ( f: (Double,Double) => Double, x, y )
  = x.join(y).mapValues{ case ((a,b)) => f(a,b) }

def multiply ( x, y )
  = x.map{ case ((i,j),m) => (j,(i,m)) }
      .join( y.map{ case ((i,j),n) => (i,(j,n)) } )
      .map{ case (k,((i,m),(j,n))) => ((i,j),m*n) }
      .reduceByKey(_+_)

for ( i <- 1 to num_steps ) {
  val E = op( _-_, R, multiply(P,Q) ).cache()
  P = op( _+_, P, op( _-_,
      multiply(E,transpose(Q)).mapValues(_*2),
      P.mapValues(_*b) ).mapValues(_*a) ).cache()
  Q = op( _+_, Q, op( _-_,
      transpose(multiply(transpose(E),P)).mapValues(_*2),
      Q.mapValues(_*b) ).mapValues(_*a) ).cache()
}
```

Matrix Factorization in DIABLO:

```
var P: matrix[Double] = matrix();
var Q: matrix[Double] = matrix();
var pq: matrix[Double] = matrix();
var E: matrix[Double] = matrix();

var steps: Int = 0;
while ( steps < num_steps ) {
  steps += 1;
  for i = 0, n-1 do
      for j = 0, m-1 do {
          pq[i,j] := 0.0;
          for k = 0, d-1 do
              pq[i,j] += P[i,k]*Q[k,j];
          E[i,j] := R[i,j]-pq[i,j];
          for k = 0, d-1 do {
              P[i,k] := P[i,k] ^ (2*a*E[i,j]*Q[k,j]);
              Q[k,j] := Q[k,j] ^ (2*a*E[i,j]*P[i,k]);
          };
      };
};
```