Spartan: A Distributed Array Framework with Smart Tiling

Chien-Chin Huang[†], Qi Chen[*], Zhaoguo Wang[†], Russell Power[†], Jorge Ortiz[‡]
Jinyang Li[†], Zhen Xiao[*]

[†]*New York University,* [*]*Peking University,* [‡]*IBM T.J. Watson Research Center*

Abstract

Application programmers in domains like machine learning, scientific computing, and computational biology are accustomed to using powerful, high productivity array languages such as MatLab, R and NumPy. Distributed array frameworks aim to scale array programs across machines. However, maximizing the locality of access to distributed arrays is an unsolved problem; such locality is critical for high performance. This paper presents Spartan, a distributed array framework that automatically determines how to best partition (aka "tile") n-dimensional arrays and to co-locate data with computation to maximize locality. Spartan combines a lazy-evaluation based, optimizing frontend with a distributed tiled array backend. Central to Spartan's design is a small number of carefully chosen parallel *high-level operators*, which form the expression graph captured by Spartan's frontend during runtime. These operators simplify the programming of distributed applications. More importantly, their well-defined semantics allow Spartan's runtime to calculate the costs of different tiling strategies and pick the best one for evaluating the entire expression graph.

Using Spartan, we have implemented 12 applications from a variety of domains including machine learning and scientific computing. Our evaluations show that Spartan's automatic tiling mechanism leads to good and scalable performance while eliminating the need for manual tiling.

1 Introduction

High productivity array-languages, such as MATLAB [42], NumPy [51] and R [63], are the dominant toolkit for application programmers in areas like machine learning, scientific computing and computational finance. To help array programs scale across machines, there have been many proposals from both the HPC and the systems communities to develop a distributed array framework (discussed in §6). However, despite these efforts, an easy-to-use, high-performance distributed array framework has remained elusive. When distributing array programs, the open challenge is how to maximize the locality of access to array data spread out across the memory of many machines. To improve locality, one needs to both partition arrays smartly and co-locate computation with data. We refer to this as the "tiling" problem. Tiling is crucial for performance; programs that optimize for locality can be an order of magnitude faster than those that don't.

Existing distributed array frameworks do not adequately address the tiling problem. Most systems rely on users to manually specify array partitioning; examples include Pydron [46], Presto [64], MadLINQ [56], Global Arrays [20] and Petsc [9]. Although SciDB [62] can automatically choose a good chunk size to optimize loading arrays from disk, it still relies on a user-defined tiling strategy. Manual tiling can achieve good locality, but makes the resulting system much more tedious and complex to use than their single-machine counterpart. Ideally, a distributed array framework should support automatic tiling with minimal user input to achieve both ease-of-use and high performance.

This paper presents Spartan distributed array framework with smart tiling. Spartan provides the popular Numpy [51] array abstractions while achieving scalable high performance across machines. The key innovation of Spartan is its automatic tiling mechanism: when distributing an n-dimensional array across machines, the runtime of Spartan can automatically decide which axis(es) to cut each array along and to co-locate computation with data.

A major design of Spartan is the five high-level parallel operators, including map, fold, filter, scan and join_update. These high-level operators capture the parallel patterns of most array programs and we use them to distribute a myriad of built-in array functions as well as user programs. The semantics of these high-level operators lead to well-defined cost profiles. The cost profile of an operator gives an estimate of the communication cost for each potential tiling strategy (row-wised, column-wised, etc.) for its inputs. Therefore, it provides crucial information to enable the runtime to perform automatic tiling. As an example, the map operator applies a user-defined function element-wise to several input arrays with the same shape. Thus, this operator achieves the best locality (and zero communication cost) if all its input arrays are partitioned in the same way. Otherwise, the cost equals to the size of those input arrays with different tiling.

At runtime, Spartan splits program execution into a series of frontend and backend steps. On the client machine, the frontend first turns a user program into an expression graph of high-level operators via lazy evaluation. It then runs a greedy search algorithm to find a good tiling for each node in the expression graph to reduce the overall communication cost. Finally, the frontend gives the tiled expression graph to the backend for execution. The backend creates distributed arrays according to the assigned tiling and evaluates each operator by scheduling parallel tasks among a collection of workers.

Spartan's automatic tiling is not without limitations. First, Spartan only aims to minimize network communication and does not consider other performance limiting factors such as how tiling impacts each machine's cache locality. Second, the default cost profile for join_update is not precise in some circumstances and require additional hints from users. While this imposes additional work from users, we have found the efforts to be reasonably low in practice. Third, the greedy search algorithm does not guarantee optimal tiling because the underlying optimization problem is NP-complete.

We have built Spartan to provide similar user interfaces as NumPy. It currently implements 50+ common Numpy functions. We have developed 12 applications on top of Spartan. All of them are simple to write using builtins or Spartan's high-level operators. Evaluations on a local cluster and the Amazon EC2 show that Spartan tiling algorithm can automatically find good tiling for arrays and achieve good scalability. Compared to an existing in-memory distributed array framework, Presto, Spartan applications achieve a speedup of $1.7\times$.

2 Automatic Tiling Overview

The Setup. The Spartan system is comprised of many worker machines in a high speed cluster. Spartan partitions each global array into several tiles (sub-arrays) and distributes each one to a potentially different worker. We refer to the partitioning strategy as *tiling*. There are several ways to "tile" an array. For example, Fig. 1 shows the three tiling choices for a 2D array (aka matrix).

In Spartan, an array is created by loading data from an external storage or as a result of some computation. Spartan decides the tiling choice for the array at its creation time. What is a good tiling choice? We consider the best tiling as one that incurs the minimum communication cost when the array is used in a computation – workers fetch and write as few remote tiles as possible. In this section, we examine what affects good tiling and give an overview of Spartan's approach to automatic tiling.

2.1 What Affects Good Tiling?

Several factors affect the tiling choice for an array. These include how the computation accesses the array, the runtime information of the array and how the array is used

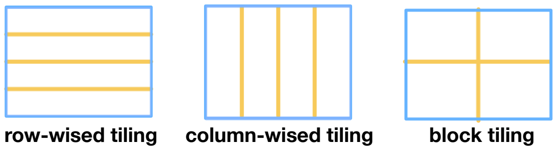

Figure 1: Three tiling methods for 2-dimensional arrays.

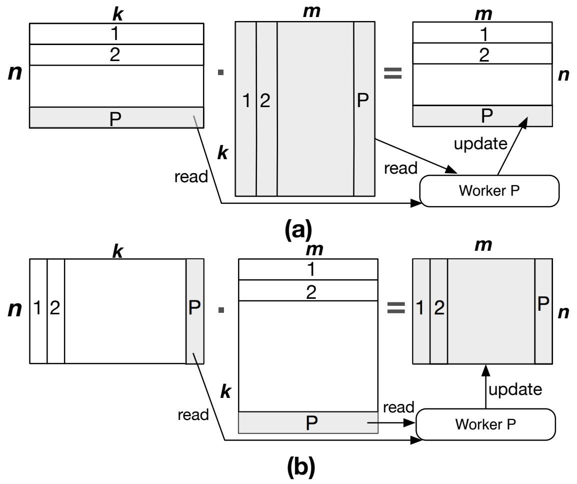

Figure 2: Two ways to implement matrix multiplication X·Y=Z, aka dot operation. Gray areas denote data read or updated by a single worker. In (a), each worker reads the entirety of Y across the network and performs local writes. Its per-worker communication cost is $k * m$. In (b), each worker performs local fetches and sends updates of size $n * m$ over the network. The per-worker communication cost is $n * m$.

across the program. Below, we illustrate how each of the factors affects tiling using concrete examples.

1) The access pattern of an array. Array computation tends to read or update an array along some particular axis. This access information is crucial for determining a good tiling. Fig. 2(a) shows the access pattern of a common implementation of matrix multiplication (aka dot). When computing $X \cdot Y = Z$, this implementation launches p parallel tasks each of which reads X row-wise and reads the entirety of Y. The task then performs a local dot and sends the result row-size to create Z. Consequently, it is best to tile both X and Z row-wise (it does not matter how Y is tiled). Other ways of tiling incur extra communication cost for fetching X and updating Z.

2) The shape and size of an array. The access pattern of an array often depends on the array's shape and size. Therefore, such runtime information affects the array's tiling choice. In addition to Fig. 2(a), there exists an alternative implementation of dot, shown as Fig. 2(b). In this alternative implementation, each of the p parallel tasks reads X column-wise and Y row-wise to perform a local matrix multiplication and update the entirety of Z. The final Z is created by aggregating updates from all p tasks. Consequently, it is best to tile X column-wise and Y row-wise.

Whether to use Fig. 2(a) or Fig. 2(b) to compute $X \cdot Y = Z$ is a runtime choice that depends on the array shapes. Suppose X is an $n \times k$ matrix and Y is a $k \times m$ matrix. Fig. 2(a) has a per task communication cost of $k * m$. This is because each task needs to fetch the entire Y across the network and can be scheduled to co-locate with the tile of X that it intends to read. By contrast, Fig. 2(b) has a per task communication cost of $n * m$. This is because each task needs to send its update of Z over the network and can be scheduled to co-locate with the tiles of X and Y that it intends to read. Therefore, the best tiling choice depends on the shape of X. If $n > k$, the cost of Fig. 2(a) is lower and the system computes `dot` using (a) whose preferred tiling for X is column-wise. If $n < k$, the cost of Fig. 2(b) is lower and the system computes `dot` using (b) whose preferred tiling for X is row-wise.

```
1   func ALS(A):
2     '''
3     Alternating Least Squares
4     Input: A is a n*k user-movie rating matrix.
5     Output: U and M are factor matrices.
6     '''
7     for i from 1 to max_iter
8         U = CalculateUsersFactor(A, M)
9         M = CalculateMoviesFactor(A, U)
10    endfor
11    return U, M
```

Figure 3: Pseudocode of Alternating Least Squares.

3) How an array is used throughout the program. An array can be read by multiple expressions. If these expressions access the array differently, we can reduce communication cost by creating multiple tilings for the array. In order to learn of an array's usage, the system cannot simply handle one expression at a time, but must "look ahead" in execution when determining an array's tiling. Consider the Alternating Least Squares (ALS) computation shown in Fig. 3. ALS solves the collaborative filtering problem by decomposing the given user-item rating matrix. Consider a movie recommendation system under ALS that makes use of two parameters: users and movies. In each iteration, ALS calculates the factor for each user, based on the rating matrix, A, and a movie factor matrix (line 5 in Fig. 3). Then, it calculates the factor for each movie based on the rating matrix, A, and users factor matrix (line 6 in Fig. 3). Thus, ALS needs to access A along both row (users) and column (movies) in one single iteration. If the system decides on A's tiling by line 8 only, it would tile A row-wise. Later, at line 9, the system incurs communication cost when reading A column-wise. This is far from optimal. If we unroll the for loop and look at all the expressions together, we can see that A is accessed by two expressions several times (max_iterations). Thus, the best tiling is to duplicate A and tile one along row and another along column.

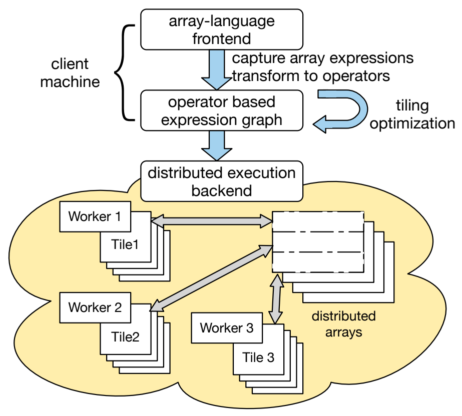

Figure 4: The layered design of Spartan. The frontend builds an expression graph and optimizes it. The backend executes the optimized graph on a cluster of machines. Each worker (3 workers in this figure) owns a portion of the global array.

2.2 Our Approach and Spartan Overview

Like NumPy and other popular array languages, users write applications in Spartan using a large number of built-in functions and array primitives (e.g. +,*,`dot`, `mean`, etc.). Spartan implements its built-in functions using a small number of *high-level parallel operators*. The high-level operators encapsulate common parallel patterns and can efficiently express most types of computation. Users may also directly program using these high-level operators if their computation cannot be expressed by existing builtins.

Spartan uses a layered approach which splits the execution into frontend and backend steps, shown in Fig. 4. The frontend, running on a client machine, captures user code and turns it into an expression graph whose nodes correspond to the high-level operators. Next, the frontend runs a tiling optimizer to determine good tiling for each node in the expression graph. Finally, the frontend sends the tiled expression graph to the backend. The backend provides high performance distributed implementations of high-level operators. For each operator, it schedules a collection of tasks running on many compute machines. The tasks create, fetch and update distributed in-memory arrays based on the tiling hint determined by the optimizer.

Spartan's high-level operators and its layered design help collect the necessary information for automatic tiling. First, by expressing various types of computation in a small set of high-level operators, the data access pattern is made explicit for analysis (§2.1 (1)). Second, the frontend dynamically captures the expression graph with runtime information about the shape of input and intermediate arrays (§2.1 (2)). Third, the expression graph represents a large execution context, thereby allowing the frontend to understand how an array is used by multiple expressions. This is crucial for good tiling (§2.1 (3)).

3 Smart Tiling with High-level Operators

This section describes the design of Spartan, focusing on those parts crucial for automatic tiling. Specifically, we discuss high-level operators (§3.1), how Spartan's frontend turns an array program into a series of expression graphs (§3.2), the basic tiling algorithm (§3.3) and additional optimizations (§3.4).

3.1 High-level Operators

A high-level operator in Spartan is a parallel computation that can be parameterized by some user-defined function [1]. The operators are "functional" in nature: they take arrays or views of arrays as input and generate a new one without modifying existing arrays in place. Spartan supports views of arrays like NumPy. A view is an interface that allows users to manipulate arrays (e.g., swapping axes, slicing) without copying data. When reading a tile of a view, Spartan translates the shape and location from the view to those of the underlying array to fetch data.

High-level operators are crucial to Spartan's smart tiling, but what operators should we use? There are two considerations in choosing them. First, each operator should capture a general parallel pattern that can be used to implement many builtins. Second, each operator should have restricted semantics that correspond to a well-defined cost profile for different ways of tiling its input and output. This enables the captured expression graph to be analyzed to identify good tiling choices.

Spartan's current collection of five high-level operators is the result of many design iterations based on our experience of building various applications and builtins. Below, we describe each operator in turn and also discuss its (communication) cost w.r.t. different tiling choices.

- D=map$(f_{map}, S_1, S_2, \ldots)$ applies function f_{map} in parallel tile-wise over input arrays, $S_1, S_2, \ldots$, and generates output array D with the same shape. The total cost is zero if all inputs have the same tiling. Otherwise, the cost is the total size of all input arrays whose tiling differs from S_1.
  As an example usage of map, Fig. 5(line 4–7) shows the implementation of Spartan's built-in array addition function which simply uses map with f_{map} as Numpy's addition function.
- D=filter(f_{pred}, S) creates a view of S that excludes elements that do not satisfy the given predicate f_{pred}. Alternatively, filter can take a boolean array in place of f_{pred}. Since filter creates a view without copying actual data, the cost is zero.

[1]The user-defined function must be free of side-effects and deterministic.

- D=fold$(f_{accum}, S, axis)$ aggregates input array S using the commutative and associate function f_{accum} along the $axis$ dimension. For example, if S is a $m \times n$ matrix, then folding it along axis=0 creates a vector of n elements. Spartan performs the underlying folding in parallel using up to m tasks. The cost of fold is zero if S is tiled along the $axis$ dimension, otherwise, the cost is $S.size$.
- D=scan$(f_{accum}, S, axis)$ computes cumulative aggregates using f_{accum} over the $axis$ dimension of S. Unlike fold, its output D has the same shape as the input. The cost profile of scan is the same as fold.
- D=join_update$(f_{join}, f_{accum}, S_1, S_2, \ldots, axis_1, axis_2, \ldots, output_shape)$ is more complex than previous operators. This operator treats each input array S_i as a group of tiles along the $axis_i$, The shapes of the input arrays must satisfy the requirement that they have the same number of tiles along their respective $axis_i$. Spartan joins each tile among different groups and applies f_{join} in parallel. Function f_{join} generates some update to be written to output D at a specified location. Multiple workers running f_{join} may concurrently update to the same location of D; such conflicts are automatically resolved by applying f_{accum}.
  As an example of join_update, consider the matrix multiplication implementation in Fig. 2(b), where S_1 is a $n \times k$ matrix and S_2 is a $k \times m$ matrix. Fig. 5 (lines 20–22) uses join_update which divides S_1 into k column vectors and S_2 into k row vectors. The f_{join} (aka `dot_udf`) is called in parallel for each column vector of S_1 joined with the corresponding row vector of S_2. It performs a local dot product of the joined column and row to generate an $n \times m$ output tile. All updates are aggregated together using the addition accumulator to create the final output.
  A special case of join_update is when some input array S_i has $axis_i = -1$. In this case, the entire array S_i will be joined with each tile of other input arrays. Fig. 5 (lines 23-25) uses this special case of join_update to realize the alternative matrix implementation of Fig. 2(a).
  The cost of join_update consists of two parts, 1) the cost to read the input arrays. 2) the cost of updating the output array. If an input array S_i is partitioned along $axis_i$, the input cost for S_i is zero, otherwise, the cost is $S_i.size$. Since the size and shape of output array created by f_{join} is unknown to Spartan, it assumes a default update cost, $D.size$.

In addition to the five high-level operators, Spartan also provides several primitives to create distributed arrays or views of arrays.

- D=newarray($shape, init\_method$) creates a distributed array with a given shape. The array can be initialized in several ways, 1) by loading data from an external storage, 2) by some computation, e.g. random, zeros.
- D=slice($S, region$) creates a view over a specified region in array S. The $region$ descriptor specifies the start and end of the sliced region along each dimension.
- D=swapaxis($S, axis_1, axis_2$) creates a view of array S by swapping the axes $axis_1$ and $axis_2$. The commonly used built-in transpose function is implemented using this operator. The output view D has a different tiling from S. For example, if S is a column-tiled matrix, then $D = swapaxis(S, 0, 1)$ is effectively a row-tiled matrix.

There is no cost for newarray, newarray and swapaxis (the cost of newarray reading from an external storage is unrelated to tiling).

```
import numpy
import spartan

# Spartan's parallel implementation of
# element-wise array addition
def add(a, b):
    return spartan.map(a, b, f_map=numpy.add)

# User-defined f_join function
def dot_udf(input_tiles):
  output_loc = spartan.location(0,0)
  output_data = numpy.dot(input_tiles[0],
                          input_tiles[1])
  return output_loc, output_data

# Spartan's parallel implementation of
# matrix multiplication
def dot(a, b):
  if a.shape[0] <= a.shape[1]:
    return spartan.join_update(S=(a, b),
                  axes=(1, 0), f_join=dot_udf,
                  shape=..., f_accum=numpy.add)
  else:
    return spartan.join_update(S=(a, b),
                               axes=(0, -1),..)
```

Figure 5: Implementations of add and dot in Spartan.

Based on the high-level operators, Spartan supports 50+ Numpy builtins. Fig. 5 shows two implementations of Spartan's builtins, add and dot.

Although Spartan's map and fold resemble the "map" and "reduce" primitives in the MapReduce world [21, 1, 67, 29], they are more restrictive. Spartan only allows f_{map} to write a tile in the same location of the output array as its input tile location and not some arbitrary location. Similarly, fold can only reduce along some $axis$ as opposed to over arbitrary keys in a key value collection. Such restriction is necessary for them to have a well-defined cost profile.

3.2 Expression Graph Capture

During a user program's execution, Spartan's frontend captures array expressions via lazy evaluation and turns

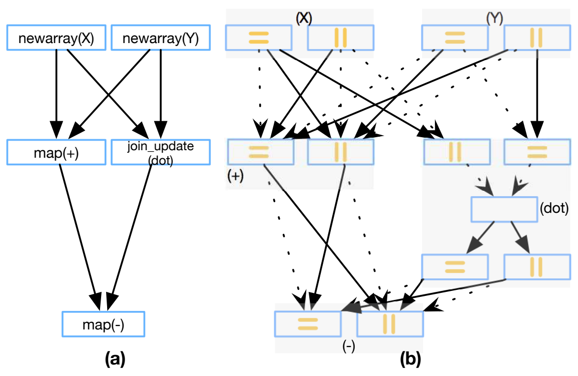

Figure 7: The expression graph and its corresponding tiling graph for $Z = X + Y - X \cdot Y$.

them into a series of expression graphs [15, 4]. In an expression graph, each node corresponds to a high-level operator and an edge from one node to another shows the data dependency between them. Fig. 7(a) shows an example expression graph. Expression graphs are acyclic because Spartan's high-level operators create immutable arrays.

The frontend stops growing an expression graph only when forced: this occurs in a few situations: (1) when a variable is used to determine the control flow, (2) when a variable is used for program output, (3) when a user explicitly requests evaluation. The use of lazy evaluation leads to an implicit form of loop unrolling: as long as there is no data dependent control flow, expression graph will continue growing until pre-configured limits.

3.3 Graph-based Tiling Optimizer

Spartan supports "rectangular" tiles: an n-dimensional array can be partitioned along any one dimension (e.g. row-wise, column-wise), or partitioned along two or more dimensions (e.g. block-wise tiling). Some existing work [28] explored other possible shapes that are more efficient for its applications.

Given an expression graph of high-level operators, the goal of the tiling optimizer is to choose a tiling for each operator node to minimize the overall cost. This optimization problem is NP-Complete (See appendix §A). It is also not practical to find the best tiling via brute force since the expression graph can be very large. Therefore, we propose a graph-based approximation algorithm to identify a good tiling quickly.

The algorithm works in two stages. First, it constructs a tiling graph based on the expression graph and the cost profile of each operator. Next, it uses a greedy strategy to search for a low cost tiling combination.

1) Constructing the tiling graph. The goal of the tiling graph is to *expose the tiling choices and cost in the expression graph*. For each operator in the expression graph, the optimizer transforms it into a *node group*, i.e.

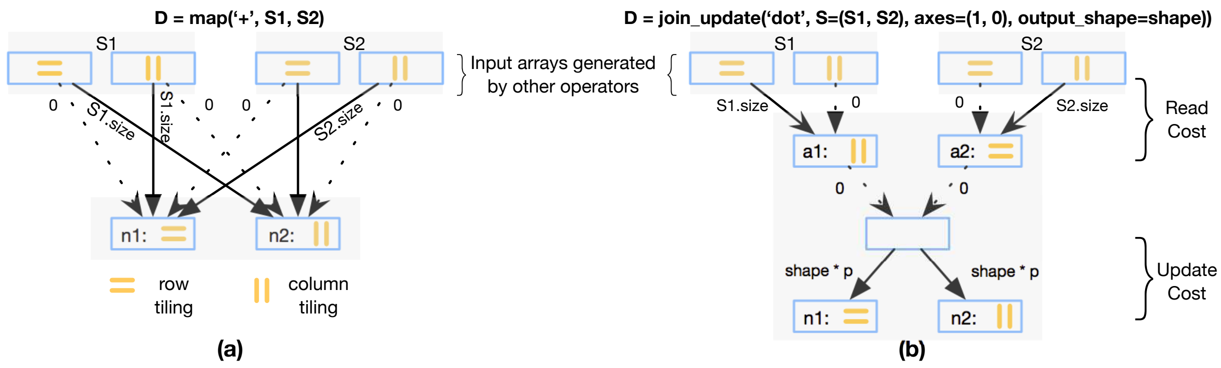

Figure 6: Two examples of building the tiling graph. (a) A plus expression, (S1 + S2), implemented by map operator (b) A dot expression, dot(S1, S2), implemented by join_update operator.

a cluster of several tiling nodes, each representing a specific choice to tile the operator's output or intermediate steps. The weight of each edge that connects two tiling nodes represents the underlying cost if the two operators are tiled according to the tiling nodes.

Fig. 6 shows how a map operator, corresponding to $D = S_1 + S_2$, is transformed. To keep the figure simple, we assume that all arrays are two dimensional with two tiling choices: row-based or column-based. And all dotted lines represent zero edge weights. As Fig. 6 shows, the map operator becomes two nodes in the tiling graph, each representing a different way to tile its output D. Similarly, each of the map operator's input arrays S_1 and S_2 (which are likely outputs from the previous operators) also correspond to two nodes. For map, there is a well-defined way to label the weights among nodes, as illustrated in Fig. 6. For example, if S_2 is tiled column-wise and D is tiled row-wise, the weight between the corresponding two nodes is S_2.size because workers have to read S_2 across the network to perform the map. fold and scan are treated similarly as map, but with edge weights labeled according to their own tiling cost profiles.

Next, we discuss the transformation of join_update. For this operator, we use some intermediate tiling nodes ($a_1, a_2 \ldots$ in Fig. 6(b)) to represent the reading cost during the join. A placeholder node is used to represent the join stage. We use another set of tiling nodes ($n1, n2$ in Fig. 6(b)) to capture the update cost to the output array. Unfortunately, Spartan can not know the precise update cost of join_update without executing the user-defined f_{join} function. Thus, we provide a default update cost according to the common update cost pattern observed in the applications implemented by join_update. If join_update is performed within a loop, the optimizer can adjust the edge cost of the tiling graph according to the actual cost observed during the previous execution of the join_update.

Fig. 6(b) shows the tiling graph used for the matrix multiplication function implemented in join_update. This implementation corresponds to the data access pattern shown in Fig. 2(b). As shown in Fig. 5, the join axes for the first and second arrays are column and row respectively. The edge weight for S_i is 0 if it matches the join axis and is $S_i.size$ otherwise. The cost is $S_i.size$ is because each worker needs to update the entirety of the result matrix. The edge weights for n_1 and n_2 are both $p * output\_shape$.

Fig. 7 gives an example showing a specific array execution ($Z = X + Y - X \cdot Y$)) and its corresponding expression graph and tiling graph. We omitted the details of other edge weights to keep the graph readable.

2) Searching for a good tiling. Deciding a tiling choice for an operator corresponds to picking one node among the corresponding node group in the underlying tiling graph and different combinations of tiling nodes pose different costs. As a result, the next step for the tiling optimizer is to analyze the tiling graph and find a combination of tiling choices that minimizes the overall cost. The tiling optimizer adopts a greedy search algorithm. The heuristic is to decide the tiling for the node group with the maximum connectivity first. Here, connectivity of a node group is the number of its adjacent node groups. When deciding a tiling for a node group X, the algorithm chooses the one resulting in the minimum cost for X. Why does this heuristic work? The cost of a tiling for an operator depends on the tiling choices of its adjacent operators. Thus, an operator with more adjacent operators has a higher impact on overall cost. Consequently, the algorithm should first minimize the cost of node groups with higher connectivity[2].

Fig. 8 shows the pseudo code for the tiling algorithm. Given a tiling graph G, the algorithm processes node groups in the order of edge connectivity (Line 19–20). For each node group (x in Line 20), the algorithm calculates the cost of each tiling node and chooses the tiling

[2]Another natural heuristic is to search the node group with largest array size first. Unfortunately, this algorithm does not perform well according to our experiments.

node with the minimum cost (Line 23–29). After deciding the good tiling ($x.chosenTiling$ in Line 30) for node group x, the algorithm removes all edges connected to all other tiling nodes (Line 32). This implies that the algorithm can't freely choose tiling for adjacent node groups of x any more – it must consider the chosen tiling of x.

$FindCost$ obtains the cost of a tiling node (T in Line 1) by calculating the sum of the minimum edge weight between each adjacent node group and T (Line 4–14). If the adjacent node group is a view operator such as swapaxis, its tiling node will be decided by T. To get accurate cost affected by T, the algorithm should also consider the adjacent node groups for its view operators. As a result, $FindCost$ recursively finds the cost of the view node group (Line 5–6). The result corresponds to the best possible cost for tiling node T.

The complexity of the tiling algorithm is $O(E * N)$ where E is the number of edges in the tiling graph and N is the number of node groups. It is not guaranteed to find the optimal tiling. However, we find that the greedy strategy works well in practice (§5).

```
1  func FindCost(NodeGroup G, TileNode T)
2    # Find the cost for tiling node T of G
3    cost = 0
4    foreach NodeGroup g in G.connectedGroups():
5      if IsView(g, G):
6        cost += FindCost(g, g.viewTileNode(T))
7      else:
8        edgeCost = INFINITY
9        foreach Edge e in g <-> T
10         edgeCost = min(edgeCost, e.cost)
11       endfor
12       cost += edgeCost
13     endif
14   endfor
15   return cost
16
17 func FindTiling(TilingGraph G)
18   # Find good tiling for every operator in G.
19   GroupList = SortGroupByConnectivity(G)
20   foreach NodeGroup x in GroupList
21     minCost = INFINITY
22     goodTiling = NONE
23     foreach TileNode y in x
24       cost = FindCost(x, y)
25       if cost < minCost:
26          minCost = cost
27          goodTiling = y
28       endif
29     endfor
30     x.chosenTiling = goodTiling
31     # Other Group can only connect to goodTiling.
32     x.removeAllConnectedEdgesExcept(goodTiling)
33   endfor
34   return G
```

Figure 8: The maximum connectivity group first algorithm to find good tiling based on the tiling graph.

3.4 Additional Tiling Optimizations

Duplication of arrays. As the ALS example in Fig 3 shows, some arrays may be accessed along different axes several times. To reduce communication, Spartan supports duplication of arrays and tiles each replica along different dimensions. To support duplication in the tiling optimizer, we add a "duplication tile" node to each node group in the underlying tiling graph. As duplication of arrays increases memory consumption. Spartan allows users to specify the memory budget for duplicating arrays to limit memory usage. Whenever the optimizer chooses to "duplicate tile" which causes an operator's output to be duplicated, it deducts from the memory budget. The optimizer will not choose duplication tiling without enough memory budget.

Sparse arrays. Dense arrays and sparse arrays are different in several aspects. First, the size of a sparse array can't be known based on the shape. Smart tiling estimates the size by sampling before constructing the tiling graph. Second, the non-zero elements distribution of intermediate arrays may be different from those of the input arrays. Smart tiling addresses this problem by adjusting edge weights after executing operators. This technique is the same as how Spartan improves its initial imprecise cost estimate of join_update with successive execution. Finally, the distribution of a sparse array can be skewed. Smart tiling can use fine-grained tiles to help backend to perform work stealing [55].

4 Implementation

Since NumPy is wildly popular in machine learning and scientific computing, our implementation goal is to replicate the "feel" of NumPy as much as possible. Our prototype currently supports 50+ most commonly used Numpy builtins.

The Spartan frontend, written in Python, captures expression graph and performs tiling optimization (§3). The Spartan backend, consists of one designated master and many worker processes on a cluster of machines. Below, we provide more details on the major backend components:

Execution engine. The backend provides efficient implementations of all high-level operators. Given an expression graph, the master is responsible for coordinating the execution of one node (a high-level operator) at a time. To execute a node, the master first creates an output array with the given tiling hint and then schedules a set of tasks to run user-defined parameter functions in parallel according to the data locality. Locality here means the task is executed on the worker that stores its input source tile. If the node corresponds to a join_update, scan or fold, the backend also associates a user-defined accumulator function with the output array to aggregate updates from multiple workers.

User-defined parameter functions are written in Python NumPy and process one tile instead of one element at a time. Like MatLab, NumPy relies on high performance C-based linear algebra libraries like BLAS [35] or LAPACK [6]. As a result, the local execution of parameter functions in each worker is efficient.

Distributed, tiled arrays. Each distributed array is partitioned into a set of tiles according to its tiling hint and stored in workers' memory. To create an array, the master assigns each of its tile to a worker (e.g. in a round-robin fashion) and distributes the tile-to-worker mapping to all workers so everybody can access remote tiles without consulting the master. If two arrays of the same shape have identical hints, the master ensures that tiles corresponding to the same region in both arrays are co-located in the memory of the same worker.

Fault tolerance. To recover from worker failure in the middle of a long computation, the backend checkpoints in-memory arrays to durable storage. Our implementation currently adopts the simplest design: after finishing an entire operator, the master periodically instructs all workers to save their tiles and also saves its own state.

5 Evaluation

In this section, we measured the performance of our smart tiling algorithm. We also evaluated the scalability of applications and compared against other open-source distributed array frameworks.

5.1 Experimental setup

We evaluated the performance of Spartan on both our local cluster as well as Amazon EC2. The local cluster is a heterogeneous setup consisting of eleven machines: 6 machines have 8-core AMD Opterons with 16GB of RAM, and 5 machines have 4-core Intel Xeons with 8GB of RAM. The machines are connected by gigabit Ethernet. For the EC2 experiments, we use 128 spot instances of the older generation m2.xlarge. Each of these instances has 17.1GB memory and 2 virtual CPUs. The network performance is rated as "moderate", which is approximately 300Mbps according to our measurements.

Unless otherwise mentioned, we ran multiple worker processes on each machine, one associated with each CPU core. We use 12 applications as our benchmarks. They include algorithms from machine learning, data mining and computational finance.

5.2 Tiling

Smart Tiling Evaluation for Applications: We compared the running time of applications with the tiling generated by smart tiling against the best tiling – the tiling that incurs the minimum communication cost. The best tiling can be pre-calculated by using a brute-force algorithm to traverse the expression graph and search the minimum communication cost among all possible tiling choices. The experiment runs on 128 EC2 instances. Fig. 9 only shows 10 applications because the computational finance ones operate on one-dimensional arrays which can only be tiled along one axis. For applications which are not perfectly scalable such as ALS and Cholesky, we set the sample sizes up to 10 million. For others, the sample sizes are up to 1 billion due to the memory limitation.

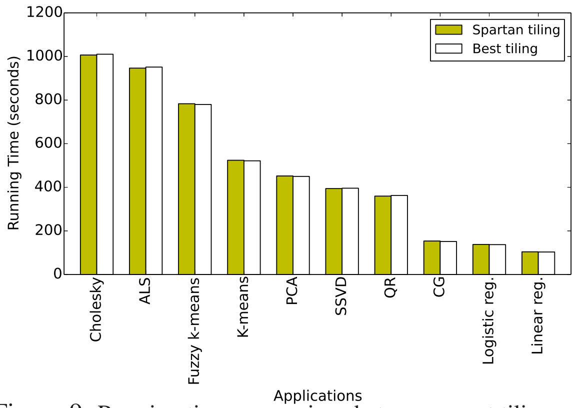

Figure 9: Running time comparison between smart tiling and the best tiling for 10 applications.

These applications show various kinds of tiling patterns. First, many applications contain expressions or operators that require runtime shape and axis information to best tile matrices, e.g. dot and join_update. Smart tiling analyzes the runtime information and gives the best tiling for the applications such as row-wise tiling for Regression and block tiling for Cholesky decomposition. Second, some program flows pass the intermediate matrices to expressions that change the view of tiling, e.g. swapaxis. Smart tiling identifies the best tiling through the global view of computation. Example applications include SSVD and PCA. Finally, some applications, like ALS, access matrices along different axes several times. As described in §2.1, the best tiling for these applications is duplication tiling.

Fig. 9 shows that Spartan's smart tiling is able to give the best tiling and improve the performance for all applications. Note that the application running time of the best tiling and Spartan's smart tiling are not the same; sometimes Spartan's smart tiling even outperforms the best tiling. The difference is caused by the instability of Amazon EC2. Spartan's optimizer makes the same choices as the best tiling for all applications.

A bad tiling can result in huge network transmission. For instance, if the tiling of the input arrays for logistic regression is partitioning along the smaller dimension, workers need to remotely fetch the matrix which is more than 512GB in the evaluation (4GB network transmission per instance in one iteration which result in approximately an extra 110 seconds in our environment). Another interesting example is ALS. Simply row-wise or column-wise tiling can result in 40% performance degradation compared to duplication tiling. Moreover, the running speed of smart tiling is fast. For example, the brute-force algorithm needs more than 500 seconds to analyze a 14-operators ALS while Spartan's smart tiling derives the same result in 0.06 seconds.

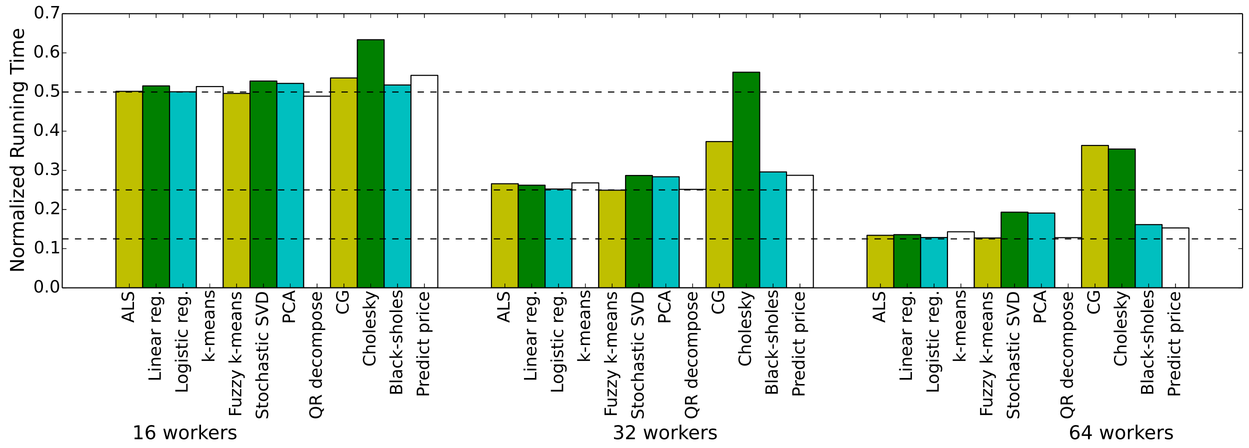

Figure 12: Fixed input size, varying number of workers. Normalized running time is calculated by dividing 8 worker running time on local cluster.

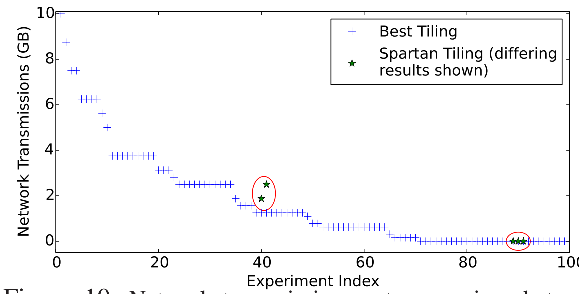

Figure 10: Network transmission cost comparison between smart tiling and the best tiling for 100 randomly generated programs. Sorted by network transmission for readability only (array sizes are randomly chosen from a set and there is no relation between experiment index and network transmission).

```
def sub_optimal_case_pattern(SIZE):
    A = expr.rand((SIZE, SIZE))
    B = expr.rand((SIZE, SIZE))
    C = A + B
    D = expr.transpose(A) + expr.transpose(B)
    E = C + D
```

Figure 11: An example that smart tiling gives sub-optimal tiling.

Smart Tiling Evaluation for Randomly Generated Programs: Although smart tiling gives the best tiling for applications we implemented, there is no guarantee that smart tiling performs well for various kinds of applications. Therefore, we examined the performance of smart tiling for randomly generated programs. Each array dimension is randomly chosen from 128K to 512K. These programs contain various numbers and types of operators Spartan has supported. The number of operators per program ranges from 2 to 15.

Fig. 10 shows the network transmission cost of 100 randomly generated programs with the tiling given by smart tiling and the best tiling. The result shows that Spartan's smart tiling can give the best tiling for most programs. It is also fast compared to the brute-force algorithm. For all programs, smart tiling needs less than 0.1 seconds while the brute-force algorithm spends 1900 seconds when the program contains 15 operators.

Fig. 11 shows the pattern residing in those programs that smart tiling gives sub-optimal tiling. The best tiling for Fig. 11 is to tile D column-wise and other operators row-wise. However, smart tiling inspects the tiling cost for C first and then for D because of the maximum connectivity. It finds that row-wise tiling costs zero for both operators. Therefore, smart tiling partitions both C and D row-wise and thus gives sub-optimal tiling due to the conflict views (caused by transpose) of C and D.

Although smart tiling cannot give the best tiling for these programs, this sub-optimal case rarely happens. Smart tiling produces a conflict view only when a program exhibits two patterns simultaneously: 1) Two operators have different views of tiling from the same input arrays. 2) Both operators have more connectivity than their input arrays. As Fig. 10 shows, only 5 out of 100 random generated programs satisfy both requirements. For three of them, the best tiling needs zero network transmission while the smart tiling needs around 0.01 GB network transmission. The number is not large because these expressions include fold which reduces the size of matrices. For the other two instances, the best tiling requires 1.3 GB but the smart tiling consumes 1.9GB and 2.6GB respectively.

5.3 Scaling

We evaluated the scalability of all applications in two ways. First, the applications use fixed-size inputs and run across a varying number of workers. Second, the applications use inputs whose sizes are scaled linearly with the number of workers. All results are normalized by the 8 workers baseline cluster size to show the relative savings (comparing with 1 worker is not fair because there is no communication for only 1 worker). All inputs are synthetic data.

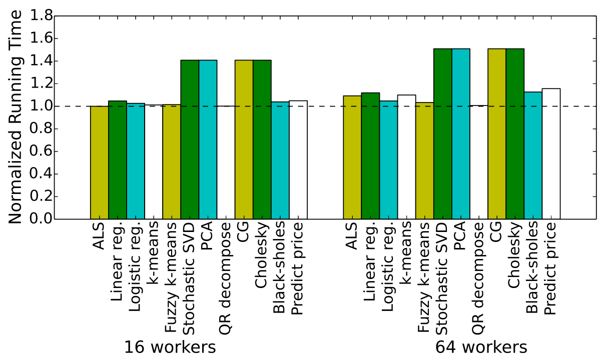

Figure 12: Scaling input size on local cluster.

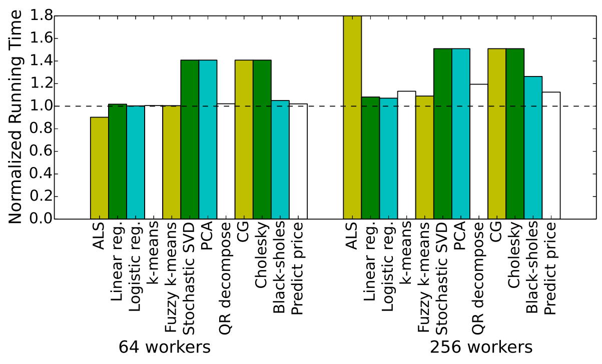

Figure 13: Scaling input size on 128 instances EC2.

Fixed input size. Fig. 12 shows the running time of 12 applications on the local cluster. The number of workers used in the experiments increases from 8 to 64. The dotted lines corresponding to $\frac{1}{2}$, $\frac{1}{4}$ or $\frac{1}{8}$ ratio represent the ideal scaling for 16, 32, and 64 workers.

The evaluation shows that the running time of many applications achieves perfect scaling. Some of them do not scale well due to the inefficiencies of the underlying algorithms. CG has many dependent folds that reduce to one value on one worker. Cholesky also has many dependent steps: the parallelism available in each step grows and shrinks, thus Cholesky cannot always utilize all workers.

Scaling input size. Fig. 12 shows the performance for 16 and 64 workers. Ideal scaling corresponds to a flat line of 1.0.To examine the scalability on a larger-scale system, we ran the experiment on EC2. Fig. 13 illustrates the experiment running up to 256 workers. The result is similar to that of Fig. 12 except for ALS. There are three matrices in ALS, rating matrix, sample matrix and item matrix. While Spartan's smart tiling can reduce the reading cost of rating matrix by duplication, ALS still needs to randomly fetch sample matrix and item matrix in each iteration and results in large communication. Thus, ALS is not scalable for large-scale datasets.

	Running Time (seconds)	Sample Size
Spartan	523.95s	1 billion
Presto	882.47s	1 billion
SciDB	2573.83s	10 million

Figure 14: K-Means performance comparison with Presto and SciDB on 128 instances EC2. The dataset for Spartan and Presto contains 1 billion points, 50 dimensions and 128 centers. The dataset for SciDB contains 10 million points.

5.4 Comparison with other systems

We compared the performance of Spartan's k-means with the implementation of Presto (also called Distributed R) and SciDB. The synthetic dataset contains 1 billion samples with 50 dimensions and 128 centers for Presto and Spartan while only 10 million samples for SciDB.

Fig. 14 shows that the performance of Spartan is 1.7x faster than Presto. Though both Spartan and Presto partition the arrays row-wise which is the best tiling, Presto requires users to explicitly assign the tiling while Spartan needs no user hints. Thus, the performance difference of Spartan and Presto comes from the backend library and implementation. We have verified this by running k-means only on a single worker.

Unlike Spartan and Presto, SciDB is not an in-memory distributed system and thus has much slower performance. The basic partition unit in SciDB is a chunk. It is important for SciDB to select the correct chunk size to reduce disk I/O. However, in Spartan, we focus on how to reduce the network communication.

6 Related Work

There is much prior work in the area of distributed array framework design and optimization.

Compiler-assisted data distribution. Prior work in this space proposes static, compile-time techniques for analysis. The first set of techniques focuses on partitioning [28] and the latter set on data co-location [33, 53, 45]. Prior work also has examined nested loops with affine array subscript patterns, using different structures (vector [28], matrix [58] or reference [30]) to model memory access patterns or polyhedral model [40] to perform localization analysis. Since static analysis deals poorly with ambiguities in source code [7], recent work proposes profile-guided methods [18] and memory-tracing [52] to capture memory access patterns. Simpler approaches focus on examining stencil code [52, 24, 26, 32, 25]. Spartan simplifies analysis significantly since high-level operator access patterns are well-defined.

Access patterns can be used to find a distribution of data that minimizes communication cost [28, 57, 10, 22, 27]. All approaches construct a weighted graph that captures possible layouts. Although searching the optimal solution is NP-Complete [31, 34, 36, 37], heuristics perform well in practice [37, 53]. Spartan adopts the idea of constructing a weighted graph. However, unlike prior work that requires language-specific compile-tile analysis, Spartan's high-level operators with know tiling costs provide enough information to analysis.

Parallel vector languages. ZPL [38], SISAL [43], NESL [13] and MatLab*P [17] share a common goal with Spartan. These languages expose distributed arrays and vector primitives and some provide a few core operators for parallel operations. Unlike Spartan, ZPL does not allow arbitrary indexing of distributed arrays and does not allow parallelization of indexable arrays. NESL relies on a *PRAM* model which assumes that a shared, distributed region of memory can be accessed with low latency. Spartan makes no such assumption. SISAL provides an explicit tiled model for arrays [23], however does not consider tiling strategies.

Distributed programming frameworks. Most distributed frameworks target primitives for key-value collections (e.g. MapReduce [21], Dryad [29], Piccolo [55], Spark [67], Ciel [48], Dandelion [59] and Naiad [47]). Some provide graph-centric primitives (e.g. GraphLab [39] and Pregel [41]). While one can encode arrays as key-value collections or graphs, doing so is much less efficient than Spartan's tile-based backend. It is possible to implement Spartan's backend by augmenting an in-memory framework, such as Spark or Piccolo. However, we built our prototype from scratch to allow for better integration with NumPy.

FlumeJava [15] provides programmers with a set of high-level operators. Its operators are transformed into MapReduce's [21] dataflow functions. FlumeJava is targeted at key-value collections instead of arrays. FlumeJava's operators look similar to Spartan's, but their underlying semantics are specific to key-value collections instead of arrays. Moreover, FlumeJava does not explicitly optimize for data locality because it is not designed for in-memory computation.

Relational queries are a natural layer on top of key-value centric distributed execution frameworks, as seen in systems like DryadLINQ [66], Shark [65], Dandelion [59] and Dremel [44]. Several efforts attempt to build an array interfaces on these. MadLINQ [56] adds support for distributed arrays and array-style computation to the dataflow model of DryadLINQ [66]. SciHadoop [14] is a plug-in for Hadoop to process array-formatted data. Google's R extensions [61], Presto [64] and SparkR [3] extend the R language to support distributed arrays. Julia [2] is a newly developed dynamic language designed for high performance and scientific computing. Julia provides primitives for users to parallel loops and distribute arrays. These extensions and languages rely on users to specify a tiling for each array, which burdens users with making non-trivial optimization that require deep familiarity which each operation and its data.

Distributed array libraries. Optimized, distributed linear algebra libraries, like LAPACK [6], ScaLAPACK [16], Elemental [54] Global Arrays Toolkit [49] and Petsc [8, 9] expose APIs specifically designed for large matrix operations. They focus on providing highly optimized implementations of specific operations. However, their speed depends on correct partitioning of arrays and their programming model is difficult to extend.

Global Address Spaces. Systems such as Unified Parallel C [19] and co-array Fortran [50] provide a global distributed address space for sharing arrays. They can be used to implement the backend for distributed array libraries. They do not directly provide a fully functional distributed array language.

Specialized application frameworks. There are a number of frameworks specifically targeted for distributed machine learning (e.g. MLBase [60], Apache Mahout [5], and Theano [12], for GPUs). Unlike these, Spartan targets a much wider audience and thus must address the complete set of challenges, including support for a number built-ins, minimizing the number of temporary copies and optimizing for locality.

Array Databases and Query Languages SciDB [62] and RasDaMan [11] are distributed databases with n-dimensional data storage and an array query language inspired by SQL. These represent the database community's answer to big numerical computation. The query language is flexible, but as the designers of SciDB have seen, application programmers often prefer expressing problems in more comprehensive array languages. SciDB-R is an attempt to win over R programmers by letting R scripts access data in SciDB and use SciDB to execute some R commands. SciDB's partition strategy is optimized for disk utilization. In contrast, Spartan focuses on in-memory data.

7 Conclusion

Spartan is a distributed array framework that provides a smart tiling algorithm to effectively partition distributed arrays. A set of carefully chosen high-level operators export well-defined communication cost and simplify the tiling process. User array code is captured by the frontend and turned into an expression graph whose nodes correspond to these high-level operators. With the expression graph, our smart tiling can estimate the communication cost across expressions and find good tilings for all the expressions.

Acknowledgments: We thank the anonymous reviewers and our shepherd David Shue. This work is supported in part by NSF (CSR-1065169) and a Google research award.

References

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Julia language. `http://julialang.org`.

[3] Sparkr: R frontend for spark. `http://amplab-extras.github.io/SparkR-pkg`.

[4] Dataflow program graphs. *IEEE Computer 15* (1982).

[5] Mahout: Scalable machine learning and data mining, 2012. `http://mahout.apache.org`.

[6] ANDERSON, E., BAI, Z., DONGARRA, J., GREENBAUM, A., MCKENNEY, A., DU CROZ, J., HAMMERLING, S., DEMMEL, J., BISCHOF, C., AND SORENSEN, D. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (1990), IEEE Computer Society Press, pp. 2–11.

[7] ANDERSON, P. Software engineering technology the use and limitations of static-analysis tools to improve software quality, 2008.

[8] BALAY, S., ABHYANKAR, S., ADAMS, M. F., BROWN, J., BRUNE, P., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W. D., KAUSHIK, D., KNEPLEY, M. G., MCINNES, L. C., RUPP, K., SMITH, B. F., AND ZHANG, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[9] BALAY, S., GROPP, W. D., MCINNES, L. C., AND SMITH, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, pp. 163–202.

[10] BAU, D., KODUKULA, I., KOTLYAR, V., PINGALI, K., AND STODGHILL, P. Solving alignment using elementary linear algebra. In *Languages and Compilers for Parallel Computing*. Springer, 1995, pp. 46–60.

[11] BAUMANN, P., DEHMEL, A., FURTADO, P., RITSCH, R., AND WIDMANN, N. The multidimensional database system RasDaMan. In *ACM SIGMOD Record* (1998), vol. 27, ACM, pp. 575–577.

[12] BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)* (2010).

[13] BLELLOCH, G. E. NESL: A nested data-parallel language.(version 3.1). Tech. rep., DTIC Document, 1995.

[14] BUCK, J. B., WATKINS, N., LEFEVRE, J., IOANNIDOU, K., MALTZAHN, C., POLYZOTIS, N., AND BRANDT, S. Scihadoop: array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011).

[15] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI - ACM SIGPLAN 2010* (2010).

[16] CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the* (1992), IEEE, pp. 120–127.

[17] CHOY, R., EDELMAN, A., AND OF, C. M. Parallel matlab: Doing it right. *Proceedings of the IEEE 93* (2005), 331–341.

[18] CHU, M., RAVINDRAN, R., AND MAHLKE, S. Data access partitioning for fine-grain parallelism on multicore architectures. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on* (2007), IEEE, pp. 369–380.

[19] CONSORTIUM, U. UPC language specifications, v1.2. Tech. rep., Lawrence Berkeley National Lab, 2005.

[20] DAILY, J., AND LEWIS, R. R. Using the global arrays toolkit to reimplement numpy for distributed computation. In *Proceedings of the 10th Python in Science Conference* (2011).

[21] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)* (2004).

[22] D'HOLLANDER, E. Partitioning and labeling of index sets in do loops with constant dependence vectors. In *1989 International Conference on Parallel Processing, University Park, PA* (1989).

[23] GAUDIOT, J.-L., BOHM, W., NAJJAR, W., DEBONI, T., FEO, J., AND MILLER, P. The sisal model of functional programming and its implementation. In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium* (1997), IEEE, pp. 112–123.

[24] HE, J., SNAVELY, A. E., VAN DER WIJNGAART, R. F., AND FRUMKIN, M. A. Automatic recognition of performance idioms in scientific applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (2011), IEEE, pp. 118–127.

[25] HENRETTY, T., STOCK, K., POUCHET, L.-N., FRANCHETTI, F., RAMANUJAM, J., AND SADAYAPPAN, P. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction* (2011), Springer, pp. 225–245.

[26] HERNANDEZ, C. K. O. Open64-based regular stencil shape recognition in hercules.

[27] HUANG, C.-H., AND SADAYAPPAN, P. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing 19*, 2 (1993), 90–102.

[28] HUDAK, D. E., AND ABRAHAM, S. G. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *ACM SIGARCH Computer Architecture News* (1990), vol. 18, ACM, pp. 187–200.

[29] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)* (2007).

[30] JU, Y.-J., AND DIETZ, H. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Languages and Compilers for Parallel Computing*. Springer, 1992, pp. 344–358.

[31] KENNEDY, K., AND KREMER, U. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS) 20*, 4 (1998), 869–916.

[32] KESSLER, C. W. Pattern-driven automatic parallelization. *Scientific Programming 5*, 3 (1996), 251–274.

[33] KNOBE, K., LUKAS, J. D., AND STEELE JR, G. L. Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of Parallel and Distributed Computing 8*, 2 (1990), 102–118.

[34] KREMER, U. Np-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands* (1993).

[35] LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS) 5*, 3 (1979), 308–323.

[36] LI, J., AND CHEN, M. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the* (1990), IEEE, pp. 424–433.

[37] LI, J., AND CHEN, M. The data alignment phase in compiling programs for distributed-memory machines. *Journal of parallel and distributed computing 13*, 2 (1991), 213–221.

[38] LIN, C., AND SNYDER, L. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*. Springer, 1994, pp. 96–114.

[39] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)* (2012).

[40] LU, Q., ALIAS, C., BONDHUGULA, U., HENRETTY, T., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., SADAYAPPAN, P., CHEN, Y., LIN, H., ET AL. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on* (2009), IEEE, pp. 348–357.

[41] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), ACM, pp. 135–146.

[42] MATHWORKS. MATLAB software.

[43] MCGRAW, J., SKEDZIELEWSKI, S., ALLAN, S., OLDEHOEFT, R., GLAUERT, J., KIRKHAM, C., NOYCE, B., AND THOMAS, R. *SISAL: streams and iteration in a single assignment language. Language Reference Manual*. 1985.

[44] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *VLDB* (2010).

[45] MILOSAVLJEVIC, I. Z., AND JABRI, M. A. Automatic array alignment in parallel matlab scripts. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings* (1999), IEEE, pp. 285–289.

[46] MÜLLER, S. C., ALONSO, G., AMARA, A., AND CSILLAGHY, A. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 645–659.

[47] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[48] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: a universal execution engine for distributed data-flow computing. NSDI.

[49] NIEPLOCHA, J., HARRISON, R. J., AND LITTLEFIELD, R. J. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing 10*, 2 (1996), 169–189.

[50] NUMRICH, R. W., AND REID, J. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum 17* (1998).

[51] OLIPHANT, T., ET AL. NumPy, a Python library for numerical computations.

[52] PARK, E., KARTSAKLIS, C., JANJUSIC, T., AND CAVAZOS, J. Trace-driven memory access pattern recognition in computational kernels. In *Proceedings of the Second Workshop on Optimizing Stencil Computations* (2014), ACM, pp. 25–32.

[53] PHILIPPSEN, M. *Automatic alignment of array data and processes to reduce communication time on DMPPs*, vol. 30. ACM, 1995.

[54] POULSON, J., MARKER, B., VAN DE GEIJN, R. A., HAMMOND, J. R., AND ROMERO, N. A. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw. 39*, 2 (feb 2013), 13:1–13:24.

[55] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Symposium on Operating System Design and Implementation (OSDI)* (2010), pp. 293–306.

[56] Qian, Z., Chen, X., Kang, N., Chen, M., Yu, Y., Moscibroda, T., and Zhang, Z. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), EuroSys '12.

[57] Ramanujam, J., and Sadayappan, P. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (1989), ACM, pp. 637–646.

[58] Ramanujam, J., and Sadayappan, P. Compile-time techniques for data distribution in distributed memory machines. *Parallel and Distributed Systems, IEEE Transactions on 2*, 4 (1991), 472–482.

[59] Rossbach, C. J., Yu, Y., Currey, J., Martin, J.-P., and Fetterly, D. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 49–68.

[60] Sparks, E. R., Talwalkar, A., Smith, V., Kottalama, J., Pan, X., Gonzaleza, J., Franklin, M., Jordana, M., and Kraskab, T. MLI: An API for distributed machine learning. In *arXiv:1310.5426* (2013).

[61] Stokely, M., Rohani, F., and Tassone, E. Large-scale parallel statistical forecasting computations in r. In *JSM Proceedings, Section on Physical and Engineering Sciences* (Alexandria, VA, 2011).

[62] Stonebraker, M., Brown, P., Becla, J., and Zhang, D. Scidb: A new dbms for science and other applications with complex analytics.

[63] Team, R. D. R: A language and environment for statistical computing.

[64] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A., and Schreiber, R. S. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems (Eurosys)* (2013).

[65] Xin, R. S., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., and Stoica, I. Shark: Sql and rich analytics at scale. In *SIGMOD* (2013).

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

[67] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.

A NP-Completeness Proof

A.1 Problem Definition

To simplify the proof, we consider only $newarray$, map and $swapaxis$ operators. The general case is discussed in section A.4. This problem contains several operators in a program and each one can be the input of others. The first step is to build an expression graph for this problem as shown in section 3.3. Next is to convert the expression graph to the tiling graph. We define a *tiling graph* as following:

1. A node group represents an operator and contains several partition nodes.
2. If an operator A is an input of an operator B in the expression graph, there are some edges between node group A and group B in the tiling graph. How node group A connects to node group B depends on the type of operator B.
3. The cost of an edge $A.tiling_I \rightarrow B.tiling_K$ is the network transmission cost to do operator B when A is tiled as $tiling_I$ and B is tiled as $tiling_K$.

Figure 15 shows three operators that will be used in the proof. There are two kinds of tilings, row and column, for each operator. There is no input for a $newarray$. As for map, there is at least one input array. The tiling nodes of an input node group are fully connected to the tiling nodes of map. If two tiling nodes represent the same tilings, there is no cost for the edge between them. Otherwise, the cost is the size of the array, N. The last operator is $swapaxis$. There is one input array for $swapaxis$ and each tiling node of the input array connects to the tiling node of $swapaxis$ representing the swapped tiling. The cost for both edges are zero.

The problem is to choose a unique tiling node for each node group without conflict and achieve the minimum overall cost (summation of cost of all edges adjacent to two chosen tiling nodes). Conflict means that if there are edges between node group A and node group B, the chosen nodes must bear the same relationship. For example, if the chosen tiling node for the input of $swapaxis$ means row tiling, the chosen tiling node for $swapaxis$ can only be column tiling to avoid conflict.

Instead of directly proving the problem, we prove the corresponding verify problem which is to find out if there is a choice with the cost less than or equal to K where K is an integer. We denote the verify problem as $TILING(K)$.

A.2 NP Proof

To show that $TILING$ is in NP, we need to prove that a given choice can be verified in polynomial time. Suppose N is the number of node groups. Given a solution, we can verify the solution by adding up the cost for all edges connected to each chosen tiling node. There are at most $N-1$ edges connected to a tiling node and N chosen tiling nodes, we can get the total cost in $O(n^2)$. Therefore, $TILING(K)$ is in NP.

A.3 NP-Completeness Proof

To show $TILING(K)$ is NP-Complete, we prove that $NAE-3SAT(N)$ can be reduced to $TILING(K)$. $NAE-3SAT$ is similar to $3SAT$ except that each clause must have at least one $true$ and one $false$. Therefore, it rules out TTT and FFF while $3SAT$ only excludes FFF.

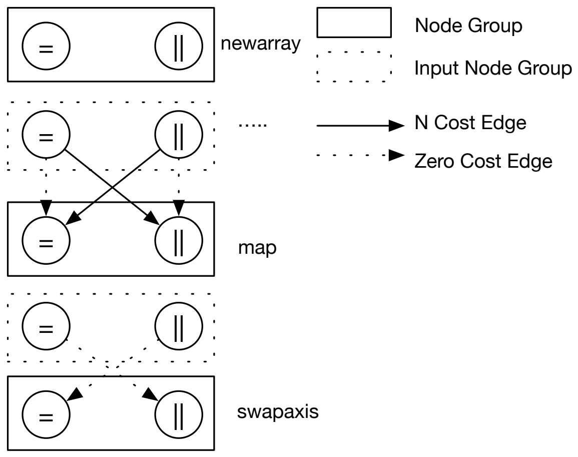

Figure 15: Three node groups and edge relationship with their input(s).

Assume that there are N literals and M clauses in the given question. M is polynomial to N. We prove that $NAE-3SAT(N)$ can be reduced to $TILING(K)$ where $K = M*2$.

1. **Construction Function,** $C(I)$:
   (a) For $C(I)$, $True$ is viewed as row tiling and $false$ is viewed as column tiling.
   (b) Each literal in $NAE-3SAT$ is an $array$ in $TILING(K)$. A negation literal is viewed as a $swapaxis$ of the original $array$.
   (c) For each clause $c_i = (L_1 \vee L_2 \vee L_3)$, $C(I)$ creates six expressions:

   $$E_1 = map(swapaxis(L_1, 0, 1), L_2)$$
   $$E_2 = map(swapaxis(L_1, 0, 1), L_3)$$
   $$E_3 = map(swapaxis(L_2, 0, 1), L_1)$$
   $$E_4 = map(swapaxis(L_2, 0, 1), L_3)$$
   $$E_5 = map(swapaxis(L_3, 0, 1), L_1)$$
   $$E_6 = map(swapaxis(L_3, 0, 1), L_2)$$

   For a negation literal, L, $swapaxis(L)$ represent the original $array$. For example, $C(I)$ creates six expressions for $c_j = (\neg L_1 \vee L_2 \vee L_3)$:

   $$E_1 = map(L_1, L_2)$$
   $$E_2 = map(L_1, L_3)$$
   $$E_3 = map(swapaxis(L_2, 0, 1), swapaxis(L_1, 0, 1))$$
   $$E_4 = map(swapaxis(L_2, 0, 1), L_3)$$
   $$E_5 = map(swapaxis(L_3, 0, 1), swapaxis(L_1, 0, 1))$$
   $$E_6 = map(swapaxis(L_3, 0, 1), L_2)$$

   For explanation purpose, we call the six expressions created by $C(I)$ a clause group.
   (d) After converting all clauses to clause groups, $C(I)$ create a cost graph according to the definition. Without loss of generality, we assume that the array size is 1. Therefore, the cost for an edge is either 0 or 1.

   For a clause group, if three literal have the same symbols, $true$ or $false$, the minimum cost is 6. For example, if three literals are all $true$ or all $false$ for $c_i = (L_1 \vee L_2 \vee L_3)$, the two inputs for each map of the clause group must have different tilings because of $swapaxis$. Thus the cost for map node group can only be 1. Since there are six maps for a clause group, the minimum cost is 6.

   For other cases, the minimum cost of a clause group is 2. For example, if L_1 is the only $true$ for $c_i = (L_1 \vee L_2 \vee L_3)$, only the input tilings of maps for $E4$ and $E6$ are different. Since all maps are not referenced by other operators, we can freely choose their tilings based only on the input tilings. Thus the cost for this case is 2. Other combinations are just symmetries of the above case and have the same cost.

   The time complexity for $C(I)$ is $O(N^2)$.

2. $C(B)$ **belongs to** $TILING(K)$ **if** B **belongs to** $TILING(K)$:
   If S is a solution for B, every clause in S has at least one $true$ and one $false$. This implies that at least one row tiling input and column tiling input for each clause group of $C(S)$. Therefore, the cost for $C(S)$ is $M*2$ which is equal to K.

3. B **belongs to** $NAE-3AT$ **if** $C(B)$ **belongs to** $TILING(K)$:
   If S is a solution for $C(B)$, there are at least one row tiling and one column tiling for each clause group. In other words, if one clause group has all row tiling inputs or all column tiling inputs, the total cost for the tiling graph will be at least $2*(M-1)+6 > K$. As a result, no clause group has all row tiling or column tiling input. Therefore, S is a solution for B.

Step 2 and step 3 prove that $NAE-3SAT$ can be reduced to $TILING(K)$.

A.4 General Graph

The previous proof only considers the tiling graph with $Array$, map and $swapaxis$. However, we argue that even though the tiling graph contains more different operators, it is still an NP-Complete problem to find out the solution. For any $TILING(K)$ which contains only the three operators, we add some other operators and expression which are independent from the original ones. Thus the new tiling graph contains two sub tiling graphs, the original tiling graph and the tiling graph representing the newly added operators. Moreover, two sub tiling graphs are not connected. Thus, to solve new $TILING(K')$ must first solve the $TILING(K)$ which is NP-Complete. Thus, we can also reduce $TILING(K)$ to the general graph.