Measuring and Optimizing Distributed Array Programs

Mingxing Zhang Yongwei Wu Kang Chen Teng Ma Weimin Zheng
Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST)
Tsinghua University, Beijing 100084, China. Research Institute of Tsinghua University In Shenzhen, Shenzhen 518057, China. Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, China.
zhangmx12@mails.tsinghua.edu.cn {wuyw, chenkang, zwm-dcs}@tsinghua.edu.cn stmatengss@163.com

ABSTRACT

Nowadays, there is a rising trend of building array-based distributed computing frameworks, which are suitable for implementing many machine learning and data mining algorithms. However, most of these frameworks only execute each primitive in an isolated manner and in the exact order defined by programmers, which implies a huge space for optimization. In this paper, we propose a novel array-based programming model, named KASEN, which distinguishes itself from models in the existing literature by defining a strict computation and communication model. This model makes it easy to analyze programs' behavior and measure their performance, with which we design a corresponding optimizer that can automatically apply high-level optimizations to the original programs written by programmers. According to our evaluation, the optimizer of KASEN can achieve a significant reduction on memory read/write, buffer allocation and network traffic, which leads to a speedup up to 5.82×.

1. INTRODUCTION

1.1 Motivation

Traditional array-based languages, such as MATLAB and R, are able to concisely express broad ideas about data manipulations. Thus they are commonly used in data-analysis and scientific settings. In order to retain this expressive power on large datasets, many tools [26] have been built to provide a similar interface over modern data-parallel computing frameworks. However, due to the restrictive primitives (e.g., the data set manipulation pattern used in Spark [30]) given by the underlying infrastructure, these tools are usually inefficient [28, 24].

As a result, there is a rising trend of building scalable and fault-tolerant computing frameworks that are directly based on distributed array operations [24, 28, 15]. These frameworks represent data in distributed (N-dimensional) arrays and leverage existing advanced techniques in the High-Performance Computing community to execute each operation (e.g., loop tiling). According to their evaluations, these array-based frameworks have achieved significant performance improvement (at least 10× faster) for many important algorithms.

Nevertheless, an important building block is still missing from the current array-based frameworks' architecture, which is a comprehensive optimizer that can speed up a program by equivalently reordering or even rewriting it. Most of the existing works adopt a naive implementation of executing each array operation in an isolated manner and in the exact order defined by programmers, which implies a huge space for optimization. As an illustration of the potentials on optimization, Spartan [15], a distributed implementation of NumPy, is shipped with a simple optimizer. According to their evaluation, Spartan's optimizer can achieve a speedup up to 2× by only fusing successive *Map* operations and *Map-Reduce* pairs.

1.2 Challenges

The idea of optimizing local array programs has been quite popular [5, 31]. But these tools cannot be applied to distributed array programs directly as they are agnostic to data synchronizations, which are required in a distributed environment. In contrast, we intend to design an optimizer that is both effective for reducing local computation time and communication cost. To achieve this goal, we retrospect our experiences of manually optimizing distributed machine learning programs and summarize three most useful optimization heuristics: *1)* make best use of the data in cache to reduce memory read/write; *2)* reuse buffers to lower memory consumption and allocation overhead; and *3)* adjust the execution order to decrease communication cost. However, although the above three heuristics can be used in the implementation of most algorithms, their usages are highly coupled with the particular algorithm. Hence they are currently applied by programmers manually. That is to say, in order to design an automatic optimizer that is effective for various algorithms, we need to first address several challenges.

i) According to our investigation, existing array-based computing frameworks usually provide users with a variety of operators [15] and the ability of using point-to-point communications [28]. Although these frameworks are generally productive for writing concise programs, they are difficult for analyzing programs' behavior and are therefore hard to optimize. As a result, we need to propose a new programming model that is both restricted enough for enabling effective automatic optimizations and is still generalized enough for writing various algorithms.

ii) Idiomatic array programs, especially when programmed in a selected set of primitives, generate a large number of isolated computation steps and expensive temporary arrays [15]. To address this problem, Spartan has explored the usages of substitution and achieved significant improvement on performance. Specifically, by fusing successive *Map* operations and *Map-Reduce* pairs into a single *Map* or *Reduce* operation respectively, the runtime of Spartan

Proceedings of the VLDB Endowment, Vol. 9, No. 12

is able to pipeline the cached partial results of a former operation to its subsequent operation without allocating a temporary array for reserving intermediate results. However, the method used in Spartan is constrained by both primitives' type and the number of their inputs (only unitary primitives)[1]. Hence, a generalized optimization algorithm that can properly substitute a group of arbitrary operations with a fused operation is desirable.

iii) Communication cost can be reduced by reordering the program execution order. But existing frameworks synchronize their data either always eagerly (immediately after the data become inconsistent) or always lazily (until it is forced by users or used for flow control). As we will show in Section 5, neither of these two options is optimal; therefore a combined method is preferred.

1.3 Our Contribution

In this paper, we resolve the above three challenges in the interface level by presenting a novel array-based programming model KASEN (Section 3). Similar to the MPI standard [13], KASEN only specifies a set of routines without providing exact implementations. However, it constrains the data layout by defining three storage states, the communication by explicit storage-state transitions, and the computation by a restrictive set of primitives (Section 4). Through imposing these restrictions, KASEN makes the data, the computation, and the communication **controllable** and **measurable**, hence enables us to design an effective optimization framework that is decoupled from the implementation of each primitive (Section 5). Here, by using the phrase "controllable and measurable", we mean that static program analyses can be used to reason both how many and what kinds of optimizations can be used, and the effect they will deliver. By using the word "decoupled", we mean that our optimizer works on the interface level. Hence users can use arbitrary methods to implement the primitives defined in KASEN, and they can all benefit from our optimization framework.

As we will show in Section 3.4, KASEN is sufficient for implementing many important machine learning and graph algorithms. More importantly, according to our quantitative analyses on various algorithms, the optimizer of KASEN can achieve significant reduction on their memory read/write, buffer allocation and network traffic (Section 8.1). In order to validate our analysis results, we also built a simple prototype of KASEN. Evaluation results based on this implementation (Section 8.2) show that our optimizer can achieve about $1.22\times - 5.82\times$ speedup on real-world and synthetic datasets.

2. OVERVIEW

KASEN provides a restricted set of primitives, which makes programs written in KASEN are easy to analyze and measure and hence enables us to apply effective optimizations. In this section, we'll describe major design choices of KASEN and the rationale behind.

i) Only two kinds of high-level data structures are allowed in KASEN. Specifically, if KASEN's users expect their data to be distributively stored on a cluster (i.e., each worker of the cluster contains only a part of the data), they should organize their data elements as a 1D dense vector or a 2D dense/sparse matrix. Otherwise, a copy of that data is maintained by each worker of the cluster. Moreover, only a restricted set of computing primitives can be used to operate these two high-level data structures: *1)* three vector-only primitives: **Map**, **Reduce** and **ZipWith**; and *2)* a (sparse-)matrix-vector primitive **MxV**. All these four primitives operate data modelled in vector or matrix as a whole object (i.e., operating only some elements of a vector/matrix is prohibited) and hence enables us to precisely analyze the dependency relations among data variables, which is a prerequisite of the following optimizations. More details about these primitives are given in Section 3.

ii) KASEN explicitly separates communications from local computations. Each variable defined in KASEN can be stored in one of the three different storage states and **only** transitions between these storage states require the network. In other words, all the computing primitives are executed locally. The advantage of this refinement is two-fold: *1)* it makes each step measurable so that we can estimate each step's computation overhead or communication cost; *2)* it naturally fits programs into the bulk synchronous parallel (BSP) model [27]. Thus the optimizer can radically work on each local-computing period without considering the other workers.

iii) We carefully studied the constraints on what kinds of computing primitives can be applied to what kinds of storage states, and the possible forms of equivalent reordering and substitutions. With the guidance of these rules, our optimizer can equivalently transmute the original program for better performance (e.g., reducing the total network traffic by heuristically switching between eager and lazy synchronizations) while guarantee the correctness.

iv) Finally, after the above three steps, we can now concentrate on optimizing each local-computing period, which means that theoretically the baton can be passed to an existing local array optimizer such as BTO [5]. However, the integration of our novel programming model and existing optimizers is nontrivial hence we decide to design a domain-specific optimizer ourselves. The optimizer we developed is simple yet still effective. In particular, our optimizer is based on the same principle of conventional loop fusion techniques. We find that two or more operations can be fused into a generalized X-in/Y-out operation if they share the same memory access pattern and their input arrays are synergistically partitioned[2]. For example, if we want to calculate the dot product of two vectors, we will need to execute a *ZipWith* operation to get the products of the corresponding elements and then a *Reduce* operation to obtain the sum. Fusing these two operations into one enables us to omit the write (by *ZipWith*) and the read (by *Reduce*) of the intermediate vector by pipelining the computations. According to our evaluation, the program's performance can be improved significantly by just applying the above fusion technique greedily.

As a summary, due to the strict programming model of KASEN, a program written in it can be easily transformed into a dependency directed acyclic graph (*DDAG*) that formalizes the data dependencies of the program. Each node of a DDAG represents a data variable and the edges are data dependencies. Since KASEN explicitly separates communications from computations, there are also two kinds of edges in a DDAG: *1)* local-computation edges that represent computing primitives; and *2)* global-communication edges that represent storage-state transitions. The communication edges separate a whole DDAG into Sub-DDAGs, each of which is a local-computing period that a local optimizer can work on. As a result, a naive implementation of the global optimizer may explore all equivalent DDAGs by using the reordering/substitution rules we provide, estimate the cost after optimizing each Sub-DDAG, and then choose the best one. However, the overhead of the above naive algorithm grows exponentially with the number of nodes in the DDAG, which may be unacceptable in many real-world cases.

[1] As an illustration of insufficiencies, two binary primitives that share one input array may be fused into a 3-in/2-out computing unit, which can save redundant reads of the common input.

[2] The input arrays have the same size and the elements in these arrays are assigned to the same worker if they have the same index.

Thus we design a greedy-based algorithm to replace it (more details can be found in Section 5.4).

3. PROGRAMMING WITH KASEN

Programming with KASEN is easy, which consists of only two steps: *1)* implementing an algorithm just like you are using a single machine; and *2)* properly partitioning the input data to exclusive worker-number parts. The framework will take care of all the other things such as storage-state transitions and optimizations. In this section, we first describe how to define data in KASEN and then how to operate them. We also provide an example of implementing the PageRank algorithm with KASEN.

3.1 Data

Data Definition As we have described in Section 2, KASEN only provides two kinds of high-level data structures namely **Vector** and **Matrix**. Variables defined in these two types will be automatically partitioned and distributed among workers. In other cases, data can also be modelled as **Shared** variables, which will be replicated and each worker will maintain an entire copy of them. For convenience, readers may analogize the types that KASEN provided to containers defined in C++ STL. They both contain one or a collection of C-style variables and constrain the allowed operations on them. In the rest of this paper, a variable is defined by using:

Shared: Shared<C-style type>

Vector: Vector<C-style type>[Length]

Matrix: Matrix<C-style type>[Height, Width]

Users may also define a fixed-size array of these types by using the C-style grammar, e.g., *"Vector<int> tmp[10]"*. Moreover, users can set the contained type of *Matrix* variables to *null*, which indicates that only the indexes are needed (i.e., no extra properties; usually used for 0/1 matrices).

Data Partition Each *Vector/Matrix* variable defined in KASEN should be partitioned into exclusive worker-number parts and distributively placed on the cluster. And, in order to enable local-only computation of computing primitives, we constrain that if the sizes of two *Vector/Matrix* variables are the same, they should be partitioned synergistically. Specifically, if *Vector* $V1$ and $V2$ have the same length, $V1[i]$ and $V2[i]$ are assigned to the same worker. Similarly, if *Matrix* $M1$ and $M2$ have the same height and the same width, $M1[i][j]$ and $M2[i][j]$ are also assigned to the same worker.

However, as we will discuss in Section 5, the effectiveness of our optimizer (when measured in proportion) is decoupled from exact data partitions. Thus users can choose any of existing partitioning algorithms for the partitioning. In our implementation, we use the block-based partitioning algorithm [9] for dense vectors/matrices and hybrid-cut [10] for sparse matrices since they work well on real-world datasets.

3.2 Computation

Shared Operations Since a copy of the variable is maintained by each worker of the cluster, in KASEN, programmers can write arbitrary functions to operate *Shared* variables, i.e.,

$$Shared\ \ ResS = F(Shared\ \ Arg1,\ Shared\ \ Arg2,\ \ldots)$$

Vector-only Operations Different from the *Shared* operations, KASEN defines three vector-only operations for operating the vectors. Specifically, the **Map** operation transforms an input *Vector* V to an output *Vector ResV* with the same length by applying the user-defined function F to each element of V (with the same supplemental argument).

$$Vector\ \ ResV = Map(Function\ \ F,\ Vector\ \ V,\ Shared\ \ Arg)$$
$$in\ \ which,\ \ ResV[i] = F(V[i],\ Arg)$$

In contrast, the **Reduce** operation transforms an input *Vector* V to an output *Shared* variable *ResS* by applying the user-defined binary function F repeatedly through iterating all the elements of V. In addition, KASEN constrains that the binary function F **must** satisfy the associative and the commutative law. Thus the iteration is not required to be in order.

$$Shared\ \ ResS = Reduce(Function\ \ F,\ Vector\ \ V)$$
$$in\ \ which,\ \ ResS = F(V[0], F(V[1], F(V[2], \ldots)))$$

Finally, the **ZipWith** operation zips two input *Vector* $V1$ and $V2$ into an output *Vector ResV* by applying the user-defined function F to elements with the same index.

$$Vector\ \ ResV = ZipWith(Function\ \ F,\ Vector\ \ V1,$$
$$Vector\ \ V2,\ Shared\ \ Arg)$$
$$in\ \ which,\ \ ResV[i] = F(V1[i],\ V2[i],\ Arg)$$

Note that all the above three vector-only operations share the same memory access pattern, i.e., they will iterate over all the elements of the input vector(s) without caring about the iterating order. Thus they can be executed simultaneously (i.e., fused by pipelining the computation) if the input vectors are synergistically partitioned.

Matrix-vector Operation KASEN also defines a (sparse-)matrix-vector operation **MxV**, whose definition is

$$Vector\ \ ResV = MxV(Function\ \ F_Z,\ Function\ \ F_R,$$
$$Vector\ \ V,\ Matrix\ \ M)$$
$$in\ \ which,\ \ ResV[i] = Reduce(F_R, ZipWith(F_Z, V, M[:,i]^T))$$

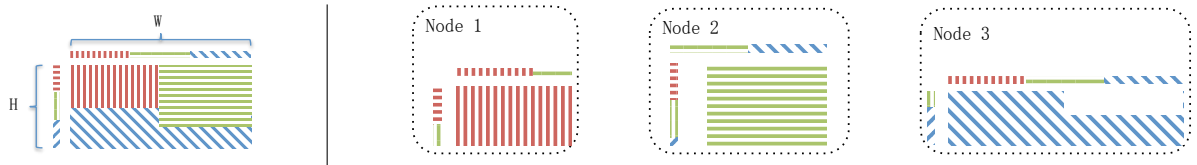
This operation takes a *Vector* with length H, a *Matrix* with size $H \times W$ and outputs a *Vector* with length W. The computation order is just like the normal multiplication between vector and matrix, i.e., the result of the i^{th} element in *ResV* is computed by *1) ZipWith* the vector V and the i^{th} column of matrix W with function F_Z; and *2)* reduces the resulting vector to a scalar with function F_R. Similar to the *Reduce* operation, in order to enable local-only computation of *MxV*, KASEN also constrains that F_R must satisfy the associative and the commutative law. As we will discuss in more detail at Section 5, successive *MxV* operations that share the same input matrix can be fused into a matrix-matrix multiplication and hence save redundant reads of the original input matrix.

Moreover, we also define the operation **MxVT**, which shares the same type of arguments and property with *MxV*, just using the transpose of the input matrix, i.e., M^T, instead of the matrix itself.

One may notice that we do not distinguish sparse matrices from dense ones. This is because that users can always use the best method they believed to implement the **MxV(T)** primitive, and they can all benefit from KASEN's optimization framework.

3.3 Example: PageRank

In this section, we use the PageRank algorithm as an example of programming with KASEN. As a formal definition, PageRank



(a) Assignment & Partitioned State. (b) Consistent State or Pre-reduced State.

Figure 1: An illustration of how to store a matrix and the corresponding vectors in a cluster of 3 workers. Data assigned to the same worker is labeled with the same color.

Algorithm 1 Program for PageRank.

Data:
Shared<int> N = load();
Matrix<null>[N, N] G = load();
Vector<double>[N] pr = {1}, new_pr, tmp1, tmp2, degree;
Shared<double> diff, r = load(), theta = load();
Shared<bool> not-converged = True;
Definition:
$F_1(x,y) = x/y$
$F_2(x,r) = (1-r)+r*x$
$F_3(x,y) = abs(x-y)$
$F_4(x) = x$
Computation:
1: degree = **MxVT**(*, +, G, pr);
2: **while** not-converged **do**
3: tmp1 = **ZipWith**(F_1, pr, degree);
4: tmp2 = **MxV**(*, +, G, tmp1);
5: new_pr = **Map**(F_2, tmp2, r);
6: diff = **Reduce**(+, **ZipWith**(F_3, pr, new_pr));
7: pr = **Map**(F_4, new_pr);
8: not-converged = (diff > theta);
9: **end while**

computes a value named "page rank" for every vertex in a directed graph, which is usually implemented by initializing each vertex with the same initial value 1 and then updating them iteratively. At every iteration, the updating formula is as follow: $pr^{t+1}[v] := (1-r)+r*\sum_{(u,v)\in E}(pr^t[u]/degree(u))$, where r is the probability of a random jump, E is the set of directed edges in graph G, $pr^t[u]$ denotes the page rank of vertex u at iteration t, and $degree(u)$ is the out degree of vertex u.

The program of PageRank is given in Algorithm 1. As one can see from the program, an **MxVT** operation is first used for calculating the out degree of each vertex. Then, the pagerank of each vertex is calculated by repeatedly performing the above updating formula until convergence.

3.4 Discussion

Although KASEN currently only supports a restrictive set of primitives, it is already sufficient for implementing many important machine learning and graph algorithms. Specifically, we have checked that all the level 1 and level 2 operations of BLAS that do not output a matrix can be implemented by KASEN. All of the other operations that are not supported run in at least quadratic time and hence are infrequently used in big data applications. As a real example, the L-BFGS algorithm [19] can be implemented by KASEN, which is a generalized technique for finding zeroes or local maxima/minima of functions. Thus the L-BFGS algorithm can be used to solve many other problems that are essentially equivalent to finding an extremum, such as linear regression, SVM. Moreover, the programming model of PowerGraph [14] is equivalent to the combination of using *MxV* and *MxVT*, thus all the graph algorithms supported by PowerGraph is also supported by KASEN.

4. SPECIFICATIONS

Section 3 only discusses how a program is written by users, without mentioning how the program is really interpreted/executed by the runtime. Thus, in this section, we describe in details the specifications of *1)* how data are actually stored in a cluster (i.e., the three storage states); *2)* how data are synchronized (i.e., the storage-state transition); and *3)* the constraints on what kinds of primitives can be applied to what kinds of data, and what to do if these constraints are not satisfied.

4.1 Three Storage States

As discussed in Section 3.1, programmers of KASEN should provide an assignment for elements of each *Vector/Matrix* variable, so that *1)* they are exclusively partitioned into worker-number parts; and *2)* variables that have the same size should be partitioned synergistically, i.e., elements of these variables must be assigned to the same worker if they have the same index. In the rest of this paper, if an element of a vector or matrix is stored in its assigned worker, we name it as the **master**. For a vector variable V or a matrix variable M, if each of the workers only stores elements assigned to it, the variable is stored in the **Partitioned** state, which is represented by adding a subscript P to the variable's label, i.e., V_P and M_P.

However, the partitioning of matrices and vectors are decoupled. Thus, when computing a $MxV(T)$ operation, the sub-matrix assigned to one worker may contain non-zero elements that the corresponding vector elements are not assigned to this worker. As a result, we also allow vectors to be stored in a **Consistent** state, so that each worker stores both the elements that are assigned to it and several **slaves** that have the same value as the corresponding masters. Formally, if a non-zero element in the i^{th} row and j^{th} column of an $H\times W$ *Matrix* M is assigned to worker k, the i^{th} element of all the vectors with length H, and the j^{th} element of all the vectors with length W, are either assigned to worker k or mirrored by slaves stored in worker k. In this paper, we represent a vector V in consistent state by adding a subscript C, i.e., V_C.

Moreover, we also allow *Vector* variables to be stored in a **Pre-reduced** state for representing intermediate results of some primitives. For each *Vector* in the pre-reduced state, an associative and commutative binary function R must be assigned to it as the reducing function. The exact value of each element is a combination of the master and all the slaves, and it is calculated by applying the reducing function R repeatedly. For instance, if an element of a pre-reduced *Vector* is stored in one master a and two slaves b and c, the actual value of this element is $R(a,R(b,c))$. As another example, if we want to get the result of multiplying a consistent vector V_C with a partitioned matrix M_P, we only need to multiply the consistent sub-vector to the partitioned sub-matrix stored in each worker **locally**, and the result will be a pre-reduced vector with reducing function $+$, which means that the actual value is the sum of master and all the slaves. The existence of pre-reduced state enables us to postpone certain network communications, and hence enables much more potential for reducing communication costs (see Sec-

Table 1: All the possible forms of storage-state transitions and local computations. The type of each variable is the same as the definitions given in Section 3.2. There is an "iff" clause for Equations (10), (11), (14), and (15), which means that if and only if the following constraint is satisfied the corresponding primitive can accept one of its input *Vector* to be stored in pre-reduced state. Moreover, the Equation (11) and (15) are labeled with a $\star$. When these two kinds of conditions are met, the user-given function is only applied to the masters.

Explicit storage-state transition	
(1) $V_{PR(R)} \xrightarrow{\text{gather}} V_P \xrightarrow{\text{scatter}} V_C$	(2) $S_{PR(R)} \xrightarrow{\text{gather}} S_P \xrightarrow{\text{scatter}} S_C$
Operating shared variables	
(3) $ResS_C = F(Arg1_C, Arg2_C, \ldots)$	
Reduce	
(4) $ResS_{PR(F)} = Reduce(F, V_P)$	(5) $ResS_{PR(F)} = Reduce(F, V_{PR(F)})$
MxV(T)	
(6) $ResV_{PR(F_R)} = MxV(F_Z, F_R, V_C, M_P)$	(7) $ResV_{PR(F_R)} = MxVT(F_Z, F_R, V_C, M_P)$
Map	
(8) $ResV_P = Map(F, V_P, Arg_C)$	(9) $ResV_C = Map(F, V_C, Arg_C)$
(10) $ResV_{PR(R)} = Map(F, V_{PR(R)}, Arg_C)$ *iff* $F(R(a, b), Arg) = R(F(a, Arg), F(b, Arg))$	
(11) $ResV_{PR(R)} = Map(F, V_{PR(R)}, Arg_C)$ *iff* $F(R(a, b), Arg) = R(F(a, Arg), b) = R(a, F(b, Arg))^\star$	
ZipWith	
(12) $ResV_P = ZipWith(F, V1_P, V2_P, Arg_C)$	(13) $ResV_C = ZipWith(F, V1_C, V2_C, Arg_C)$
(14) $ResV_{PR(R)} = ZipWith(F, V1_{PR(R)}, V2_C, Arg_C)$ *iff* $F(R(a, b), c, Arg) = R(F(a, c, Arg), F(b, c, Arg))$	
(15) $ResV_{PR(R)} = ZipWith(F, V1_{PR(R)}, V2_P, Arg_C)$ *iff* $F(R(a, b), c, Arg) = R(F(a, c, Arg), b) = R(a, F(b, c, Arg))^\star$	

tion 5 for more details). Again, in the rest of this paper, we represent a vector V stored in pre-reduced state with reducing function R by adding a subscript $PR(R)$ to the variable's label, i.e., $V_{PR(R)}$.

As a summarization, Figure 1a shows an assignment from user, and Figure 1b gives the exact data stored in each worker. In Figure 1a, elements assigned to different workers are labeled with different colors. If data is stored in partitioned state, each worker only stores the elements assigned to it. In contrast, as shown by Figure 1b, certain number of slaves are maintained for data stored in consistent or pre-reduced state. Specifically, in a consistent/pre-reduced state, each worker stores both the data assigned to it (the masters) and slaves of other workers according to the assigned sub-matrix. The only difference between these two states is that: *1)* in consistent state, the values of slaves are equal to the corresponding masters; and *2)* in pre-reduced state, the actual value of each element is a combination of master and all the slaves.

Similar to vectors, a *Shared* variable can also be stored in partitioned, consistent or pre-reduced state, which are represented by S_P, S_C and $S_{PR(R)}$, respectively. The value of S_P is only stored in its master; while the actual value of $S_{PR(R)}$ is calculated by reducing all the workers' value.

4.2 Explicit Storage State Transition

Explicit transitions between the above three storage states are unidirectional. A pre-reduced vector $V_{PR(R)}$ can be transformed into a partitioned vector V_P via a **gather** operation, implying that the master of each element gathers the information of all its slaves and calculate the actual value of that element by applying the reducing function R repeatedly. After the *gather* operation, only masters contain valid data. Then, a partitioned vector V_P can be transformed into a consistent vector V_C via a **scatter** operation, which indicates that the master of each element scatters its value to all its slaves. The state transition of a *Shared* variable is just the same as vectors. However, the matrix should always be stored in partitioned state. As discussed in Section 2, **only** these explicit storage-state transitions require the network.

One may notice that the amount of network traffic for each *gather/scatter* operation depends on the data partition. However, as we will describe in Section 5, what the optimizer of KASEN actually does is reducing the number of these storage-state transitions. Thus, our optimizer's effectiveness is decoupled with the exact data partitioning.

4.3 Constraints on Executing the Primitives

Since the execution of a computing primitive only computes a partial result on the data locally stored in each worker, we should carefully consider the storage state of input data and the property that the user-defined function has before executing the four primitives defined in Section 3.2. Specifically, all the possible forms of executions are given in Table 1 and we summarize them into the following rules:

i) All the *Shared* variables should be stored in consistent state before executing **any** primitive. This is obvious since *Shared* variables can be operated by an arbitrary function.

ii) The input vector of a *Reduce* operation should be stored in partitioned state. It can also be stored in pre-reduced state if the user-defined function is the same as the vector's reducing function. In contrast, the resulted *Shared* variable is always stored in pre-reduced state since each worker only calculates the result locally.

iii) Before executing *MxV(T)*, the input vector V must be stored in consistent state, and the resulted vector is stored in pre-reduce state with reducing function F_R, which is the same as the second argument of *MxV(T)*.

iv) The input vector V of a *Map* operation is allowed to be stored in partitioned, consistent and pre-reduce states, and the resulted vector *ResV* has the same storage state as V. However, if the input vector is stored in pre-reduce state with reducing function R, the execution of this *Map* operation is only allowed if the user-defined F and R satisfy the constraint given by Equation (10) or (11).

In the rest of this paper, we will name the constraint of Equation (10) as the **exchange law**. In order to prove that the exchange law is a sufficient constraint, we should show that after each worker locally applying the function F to all the masters and slaves of a pre-reduced vector $V_{PR(R)}$, the result will be a pre-reduced vector $ResV_{PR(R)}$. It is equivalent to show that the exchange law can be extended to a generalized form, i.e., $F(R(a,R(b,R(c,\ldots)))) = R(F(a),R(F(b),R(F(C),\ldots)))$, which can be proved by using the exchange law repeatedly.

As a comparison, we name the constraint of Equation (11) as the **pushdown law**, since the out-layer operation can be "pushed

down" to either of the inner-layer operation's input. In such a case, the user-given function F is only applied to masters of the pre-reduced vector, so that F is only applied once for every element. For example, if "$F(x) = x + 1$" and the reducing function of the input pre-reduced vector is also $+$, they satisfy the pushdown law. As a result, each worker can execute this operation by independently adding 1 to each master of the vector. The whole operation can be executed locally by each worker, and the result is still a pre-reduced vector with reducing function $+$.

v) The input vector $V1$ and $V2$ of a *ZipWith* operation are allowed to be stored *1)* in both partitioned state; *2)* in both consistent state; *3)* one in pre-reduce state and one in consistent state if the user-defined function F and the reducing function R satisfy the constraint given by Equation (14), which is the **distributive law**; and *4)* one in pre-reduce state and one in partitioned state if F and R satisfy the constraint given by Equation (15), which is another kind of the pushdown law. Taking the most popular reducing function $+$ as an illustration, if $F = *$, they satisfy the distributive law and if $F = -$, they satisfy the pushdown law. The proof of Equation (14) and (15) is similar to the proof of Equation (10) and (11) respectively, thus we omit it because of the space limit.

Finally, if the runtime detects that the current storage state of the input variable does not satisfy any of the forms listed in Table 1, it should insert explicit storage-state transitions. As we will discuss in the next section, the optimizer will instruct the runtime to insert these transitions in a way that incurs the lowest network traffic.

5. FUSION-BASED OPTIMIZATION

As discussed in Section 1.2, in order to enable effective and automatic optimizations, a restrictive computation model is preferred. In light of this principle, we selectively choose only four kinds of computation primitives for KASEN (i.e., *Map*, *Reduce*, *ZipWith*, and *MxM*), which, according to our investigation, are already enough for implementing various important algorithms. Moreover, all these primitives are simple transformations that are prevalently used and extensively studied in the literature: *1)* all the three vector-only operators can be implemented with a single for loop, which, if used isolatedly, can be tuned to perform at or near the possible peak by any sophisticated compiler; and *2) MxV(T)* is also a popular and important computation kernel that has been provided by many existing libraries [9, 20].

Nevertheless, tuning these operations in isolation will miss the opportunity of optimizations that result from composing multiple subprograms. To resolve this issue, we design an optimization framework for KASEN, which is based on the same principle of conventional loop fusion techniques. As we will show later in Section 8, according to our theoretical analyses and experimental results, KASEN's optimizer can achieve significant reductions on both the local computation time and communication cost.

5.1 Loop Fusion

Data locality plays a vital role in the implementation of efficient array programs, and it is especially important for programs that process "Big Data". As implied by the name, the input size of big data programs is tremendous. Thus, in order to be practical, most of big data algorithms run in quasilinear time (i.e, $O(n * log^k n)$), which indicates that their performances are usually limited by memory bandwidth [5]. In such cases, tuning the performance of each single operation becomes not enough, and hence an optimizer that can cross operation bounds is highly desired.

To this end, a sort of optimization techniques named loop fusion has been proposed and studied in the HPC community. Loop fusion is a transformation that fuses one or more loops into a single loop. When the fused loops reference the same matrices or vectors, the temporal locality of those references can be improved. Taking the PageRank program given in Algorithm 1 as an example, if implemented in an isolated manner, the five vector-only operations in line 5-7 and line 3 of the next iteration should take five for loops (as shown in Figure 2 (a))[3], whose data locality can be largely improved by fusing them into a single for loop. As only nine different variables are read/write for each *i*, if the cache is large enough for storing them, the fused implementation given in Figure 2 (b) will only read three vectors (i.e., *tmp2*, *pr*, *degree*) and write two vectors (i.e., *pr*, *tmp1*), which leads to a 55% reduction on memory read/write. According to recent investigations [5], loop fusion can achieve significant runtime speedups for memory-bound linear algebra computations.

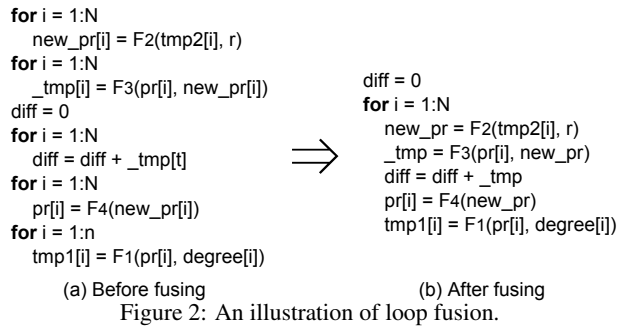


Figure 2: An illustration of loop fusion.

In KASEN, a program can also be optimized by using the same principle of loop fusion. As we have mentioned in Section 3.2, the four kinds of computing primitives can be grouped into two categories according to their memory access pattern, and a sequence of operations that belong to the same category can be fused for improving the performance. However, applying loop fusion to KASEN is not as straightforward as fusing all the operations. As listed in Table 1, the output variable of each operation is stored in various storage states, and only several combinations of operation types and storage states are valid. Thus fusing can be interrupted because of a state mismatch, and proper storage state transitions are needed to be automatically inserted. Moreover, in order to decrease the amount of network traffic, we also need to design a hybrid scheme (that combines eager and lazy synchronization) for this insertion.

In the rest of this section, we will first describe the specifications of two internal operators used in KASEN. Each of them represents a unique memory access pattern, and all the external operators (as listed in Section 3.2) that follow this pattern can be replaced by the corresponding internal operator. Then, we will introduce the Dependency DAG, which is a data structure that facilitates the analyzing procedure of KASEN's optimization. Finally, we will discuss about our optimization framework, including some implementation details and real-world examples.

5.2 Internal Operators

The most straightforward method of enabling fusion on KASEN is providing some more combined versions of operators. For example, we can provide a 3-in/1-out operator *ZipWith3(Func, a, b, c)* for replacing the original combination of two *ZipWith* operations (i.e., *ZipWith(Func$_1$, ZipWith(Func$_2$, a, b), c)*). However, it is infeasible to add every possible combination to the standard. Even if

[3]One may notice that the temporary variables *_tmp* and *new_pr* are vectors in Figure 2 (a) and are only scalars in Figure 2 (b). This is an illustration of how fusion can help reducing temporary vectors.

it is possible, they are still tedious for programmers to use. As a comparison, in KASEN, we only define one more **internal** operation for every unique memory access pattern (i.e., two in total). And it is the optimizer's responsibility to automatically map external operations to internal ones, which is transparent to programmers.

IterOnVec As we have mentioned in section 3.2, all the three vector-only operations (i.e., *Map*, *Reduce*, and *ZipWith*) KASEN provided share the same memory access pattern. They can all be implemented by iterating on the input vector(s), and, more importantly, the order of this iteration does not matter. As a result, instead of executing each of the operations in isolation, a sequence of vector-only operations can also be executed as a single fused for loop (as shown by Figure 2 (b)). Even in a distributed environment, this fusion is possible as long as all the needed input data are already stored in local (i.e., the combination rules given in Table 1 are not violated). As a result, we define an *IterOnVec* operator to formalize this kind of fusion. The *IterOnVec* operation is implemented as a variadic function that takes a fused X-in/Y-out function, read X input variables, and output Y variables simultaneously. If the fused external operations reference the same vectors, the redundant read of them are reduced. Moreover, all the temporary vectors that are original used for reserving intermediate results can be replaced with only a single temporary variable, which is much cheaper.

MxM(T) Similar to *IterOnVec*, *MxM(T)* is a generalized variadic operator that can replace a sequence of *MxV(T)* operations. However, it requires that all these fused *MxV(T)*s should share the same input matrix, and they do not constitute a dependency relation. In such case, redundant read of the input *Matrix* variable can be saved. As for the implementation, *MxM(T)* essentially implements the so-called GEneral Matrix to Matrix Multiplication (GEMM) operation in which the combined matrix is small but dense, while the other one can be sparse. As GEMM is also a prevalently used computation kernel that has been studied extensively, we omit the details of its implementation for saving the space.

5.3 Dependency DAG

For facilitating the automatic procedure of mapping external APIs to internal operators, we proposed a data structure named Dependency DAG (DDAG). Each DDAG is an equivalent representation of a code snippet, which is both easy for human to read and for computer to process. Specifically, each node of a DDAG represents a variable and the edges are data dependencies. Since KASEN explicitly separates communications from computations, there are also two kinds of edges in a DDAG: *1)* local-computation edges that represent computing primitives (i.e., external computation operators); and *2)* global-communication edges that represent storage-state transitions. Moreover, the storage state of each variable, which is critical for determining whether a DDAG violates the combination rules or not, is attached to each node as a property.

As an illustration, Figure 3 presents two equivalent DDAGs that can be transformed from Algorithm 1's line 5-7 and line 3 from the next iteration. In this figure, the storage states are labeled in subscripts, while the local-computation and global-communication edges are represented by black solid and blue dotted arrows respectively. As we can see, the only difference of these two equivalent DDAGs are related to the storage state transformations. DDAG 2 defers one *Scatter* operation to bottom of the DDAG, while, in contrast, DDAG 1 uses an eager-synchronization schema that synchronizes vector *tmp2* immediately after it becomes inconsistent (i.e., stored in pre-reduced state). As we will discuss in the next section, since the cost of operating a pre-reduced/consistent *Vector* variable is bigger than operating a partitioned vector, the hybrid schema used by DDAG 2 is a better choice. Moreover, Section 6

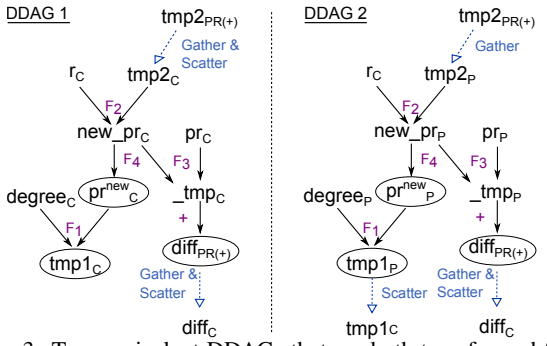


Figure 3: Two equivalent DDAGs that are both transformed from Algorithm 1's line 5-7 and line 3 from the next iteration.

will give exact formulas that can be used to estimate the cost of these two DDAGs.

Based on the above definitions, we can further define fusible Sub-DDAGs. A fusible Sub-DDAG is a connected part of the original DDAG that can be fused and executed by either one *IterOnVec* or one *MxM(T)* operation. That is to say, all the operations compose a fusible Sub-DDAG should be *1)* vector-only operators; or *2)* a sequence of *MxV(T)* operations that share the same input matrix. Take Figure 3 as an example, all the local-computation edges and the related nodes constitute a fusible Sub-DDAG; they can be fused and executed by one *IterOnVec* operation.

5.4 The Optimization Framework

With former tools, we can now start designing an optimizer for KASEN. Since the whole program can be transferred to a DDAG, a naive implementation of the optimizer is enumerating all the possible equivalent DDAGs; estimating the cost after fusing every fusible Sub-DDAG; and then choose the best execution plan from them. However, such an implementation may incur an unacceptable overhead since the number of equivalent DDAGs grows exponentially with the number of nodes in the DDAG. Thus we design a greedy-based algorithm that dynamically outputs one fusible Sub-DDAG at a time for the execution.

Overview As a summary, our optimizer will try to expand the current constructing Sub-DDAG as much as possible as long as the operations that presented by the added edges are *1)* ready for execution; and *2)* can be fused with the other operations that are already in the Sub-DDAG. Each iteration of our optimizer is implemented by the following steps:

Step 1: Start with an empty Sub-DDAG.

Step 2: Expand the current Sub-DDAG by repeatedly adding statements that are currently ready for execution and can be fused with the other operations that are already in the Sub-DDAG.

- A statement is **not** ready for execution if:
  - (a) the data needed are not yet available;
  - (b) the data are ready but the storage states of the data do not match the possible forms listed in Table 1;
  - (c) conditions of the corresponding control statements are not yet available.
- Two operations can be fused if:
  - (a) they are both vector-only operations and the input vectors are synergistically partitioned;
  - (b) they are both *MxV* or both *MxVT* operations that read the same *Matrix* variable and do not constitute a dependency relation.

Step 3: Insert explicit storage-state transitions in a way that incurs lowest network traffic.
Step 4: Identify variables that are still needed after finishing all the operations belonged to this Sub-DDAG; and prepare buffers for them.
Step 5: Fuse the whole Sub-DDAG into one *IterOnVec* or *MxM(T)* operation that executes all the sub-operations simultaneously.

Initialization As for the implementation of the above optimization algorithm, the runtime of KASEN first conducts a flow-sensitive dependency analysis [16] on the program. For each statement, the dependency analysis outputs two kinds of information: *1)* always-dependent statements, which are the statements that the current statement will always read data from in all cases; and *2)* may-dependent statements, which are the statements that the current statement may read data from in some cases but we cannot decide now because some control conditions are not available. For example, at the beginning of executing Algorithm 1, the *MxV* operation that produces *tmp2* is always-dependent on the *ZipWith* operation on line 1. In contrast, this *ZipWith* operation on line 1 is only may-dependent on the *Map* operation at line 7, because the value of *not-converged* is not yet available.
Sub-DDAG Expansion With these dependency relations, a statement that is ready for execution can be identified by *1)* it does not have any may-dependent statement (i.e., all the control dependencies are available); *2)* all the corresponding always-dependent statements are executed or have already been added to the current Sub-DDAG; and *3)* the storage states of input data match the possible forms listed in Table 1. The runtime should expand the current Sub-DDAG as much as possible by repeatedly adding ready statements that can be fused. Note that the programming order of statements does not matter in DDAG, thus the later state can be added to the current Sub-DDAG even if some of the prior statements are not. Moreover, we allow users to label a condition as "likely to be true" so that such a condition is assumed as true during the expansion. For example, through labeling the coverage condition tested in line 2 of Algorithm 1, the may-dependent relation that we mention above will be considered as an always-dependent relation. Hence we are enabled to fuse the operations in line 5-7 with the operation on line 3 of the next iteration.
Inserting Network Primitives After the expansion is finished, the runtime needs to add storage-state transitions for enabling the execution. For example, in Algorithm 1, variables *tmp2* and *diff* are originally stored in pre-reduced state after the execution of line 4 and line 6 (as defined by rule 6 and rule 4 in Table 1 respectively). If a lazy transition strategy is used, the synchronization of these variables can be deferred until line 8 (synchronize *diff* as required by rule 3 of Table 1) and line 4 of the next iteration (synchronize *tmp1* as required by rule 6 of Table 1). Here, by using the word "synchronize", we mean that a pair of *gather* and *scatter* operations is inserted to transfer the variable's storage state from pre-reduced to consistent. However, this deferring is unfavorable. Since the cost of operating a pre-reduced/consistent *Vector* variable is bigger than a partitioned one, the optimal strategy (as shown in Figure 3) is inserting a *gather* operation for *tmp2* immediately after line 4 and a *scatter* operation of *tmp1* before line 4. Although the communication cost is not reduced by this hybrid strategy, the amount of memory read/write is decreased.

According to Table 1, at most one *Vector* variable can be stored in pre-reduced state before the execution of a primitive. Thus we can prove that the best transition strategy must be adding storage-state transitions either at the bottom of a Sub-DDAG (i.e., lazy) or at the top of a Sub-DDAG (i.e., eager); never in between. For instance, if we only want to know the result of *Reduce*(F_1, *MxV*(F_2, F_1, V_C, M_P)), a lazy transition is preferred since it only needs to synchronize the final result and saves the synchronization of the intermediate *Vector* generated by *MxV*. In contrast, in some algorithms (e.g., L-BFGS [19]), there are two or more consistent *Vector* variables that depend on one pre-reduced vector. In this case, an eager transfer can reduce the number of synchronized vectors. As a result, the optimizer should enumerate all the possible strategies[4]; estimate the amount of computation and network traffic; and intelligently choose the less time-consuming one[5].
Buffer Preparation In the fourth step, we need to identify variables that are still needed after finishing all the operations belonged to this Sub-DDAG. It is achieved by identifying data that are depended (always-dependent or may-dependent) by unexecuted statements. The three examples of needed data are circled in Figure 3. Since we now know the exact number and size of the needed variables, we only need to spare enough buffers for them. The buffer reusing can be resolved by sparing a big buffer pool (a set of vectors) and increasing the number of buffered vectors if needed only at the beginning of each Sub-DDAG's execution.
Executing In the fifth step, all the operations in the current Sub-DDAG are fused and executed simultaneously. For a Sub-DDAG that consists of vector-only operations, an *IterOnVec* operation is executed. Otherwise, an *MxM(T)* operation is executed.

5.5 Example

Finally, let us take the whole PageRank program given in Algorithm 1 as an example. At the beginning of its execution, all the variables are stored in consistent state hence the *MxVT* operation at line 1 is ready for execution and is added to the current Sub-DDAG. However, all the other operations cannot be fused with it and therefore Sub-DDAG 1 consists of only one operation. Similarly, the second and third Sub-DDAGs generated by our optimizer also contain only one computing primitive. The only difference between these two Sub-DDAGs and Sub-DDAG 1 is that a storage-state transition is automatically inserted at the top of them, because the corresponding variable's storage state does not satisfy any rule listed in Table 1 (e.g., a *gather* operation is inserted for transferring the *degree* vector from pre-reduced state to partitioned state, because F_1 does not satisfy the distributed or pushdown law).

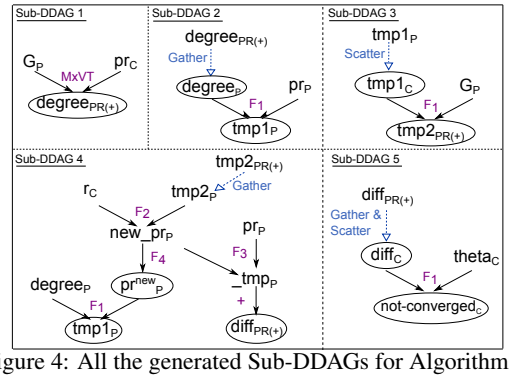


Figure 4: All the generated Sub-DDAGs for Algorithm 1

In contrast with all the above Sub-DDAGs, Sub-DDAG 4 consists of five external operators and is fused into one *IterOnVec* operation. It should be noted that the *gather* operation that transfers *tmp2* from pre-reduced state to partitioned state can actually

[4]The number of possible strategies is less than 3, because at most one input vector is allowed to be stored in pre-reduced state.
[5]The estimation should be a combination of data size, CPU frequency, and memory/network bandwidth.

be postponed, because F_2 can be split into two parts that satisfy exchange law and pushdown law respectively. But, as discussed in Section 5.4, using such a lazy transition strategy here is not good for performance. One may also notice that the *ZipWith* operation at line 3 of Algorithm 1, which belongs to the next iteration, is also fused into Sub-DDAG 4 even though that currently the value of *not-converged* is not available. This ascribes to the capability of labeling a condition as "likely to be true". Finally, the last Sub-DDAG 5 also contains only one computing operation.

6. MEASURING FORMULAS

The strict model of KASEN not only enables us to do effective optimizations but also makes us to be able to statically estimate a program's performance, which is useful for measuring the performance gain of our optimizer. In this section, we will introduce the measuring formulas for every external/internal operation, which can estimate the amount of memory read/write, buffer allocation, and communication cost for it[6]. Since results of these formulas only depend on the partitioning algorithm, they are fixed during the whole execution.

Specifically, for each *Vector* variable that has N masters (i.e., length of the vector) and S slaves (i.e., number of replicas in consistent state), *1)* S elements are transferred for the execution of each *gather/scatter* operation; *2)* N elements are read/wrote if the vector is used as an input/output variable and is stored in partitioned state; *3)* $N+S$ elements are read/wrote if the vector is used as an input/output variable and is stored in consistent/pre-reduced state. As for each *MxV(T)* operation "$ResV = MxV(F_Z,\ F_R,\ V,\ M)$", for each non-zero element (i, j) of M, the system needs to read *sizeof(M[i][j])* + *sizeof(V[i])* + *sizeof(ResV[j])* bytes and write *sizeof(ResV[j])* bytes.

As for the internal operations described in Section 5.2, the cost of each *IterOnVec* or *MxM(T)* operation can be estimated by counting its input and output variables. The execution of an internal operation (either *IterOnVec* or *MxM(T)*) will only read every input variable once and write every output variable once (if the cache is large enough for reserving all the intermediate data used for **one** element). Moreover, since we need to allocate a buffer for each output variable, the amount of buffer allocation is the same as the sum of output variables. In a data dependency figure like Figure 3, input variables are represented by variable nodes that have no in-coming edges. In contrast, the output variables are variables that are still needed later, which include not only variable nodes that have no out-coming edges. As a result, the output variables are marked explicitly by adding a circle around their labels. Procedures for identifying these output variables have already been described in Section 5.4.

As an illustration of the formulas' usages, in Sub-DDAG 4 of Figure 4, a total of two *Map* operations, two *ZipWith* operations and one *Reduce* operation should be executed before the fusion, and hence a total of $7N$ elements are read and $4N$ elements are written. However, after these operations are fused into one *IterOnVec* operation, only three vectors (i.e., *tmp2*, *pr*, and *degree*) are read and two vectors (i.e., pr^{new} and *tmp1*) are written. All the other intermediate results are only cached in the cache and will be pipelined to the next step immediately, thus they do not incur memory write or buffer allocations. As a result, a total of $4N$ elements read and $2N$ elements write are saved. A more detailed analysis of the whole PageRank algorithm is given in Section 8.1.

[6]The computation cost depends on the user-given functions and hence cannot be estimated. However, since we focus on optimizing a program in the interface level, the estimation of computation cost is also unnecessary.

7. IMPLEMENTAION

We have repeatedly emphasized that both KASEN and its optimizer work on the interface level, so that they are decoupled with their implementations. As a result, benefits result from our optimization algorithm can be statically calculated by mathematical formulas (as we will give in Section 8.1), which would not change with the exact evaluating environment. However, it is still meaningful to present some experimental results, especially the comparison between KASEN and the other existing frameworks. To this end, we developed a C++11-based prototype of KASEN on top of MPICH2, which is a state-of-the-art implementation of the MPI standard.

Our implementation of KASEN is entirely symmetric, hence there is no single coordinating instance or scheduler. At the beginning of an execution, each worker of the cluster starts by reading a unique subset of data from an NFS, which is also the destination of final results[7]. After loading, the initial data placement usually incurs suboptimal synchronization overhead and may not satisfy the constraints of synergistic partitioning that we have stated in Section 3.1. As a result, there is then a re-dispatching phase of the loaded data. Specifically, this re-dispatching phase consists of three steps. First, our system calculates an element assignment for each group of *Vector/Matrix* variables that have the same dimension size. For groups that consist of only dense vectors or matrices, the classical block-based partitioning algorithm is used. In contrast, the *hybrid-cut* partitioning algorithm [10] is used for partitioning sparse matrices. After the assignment, a global shuffling is used for each node to send data that is loaded by not assigned to it to other workers, which constitute the masters of every data element. As we have described in Section 4.1, our system automatically will set up the corresponding slaves of every data element for enabling local-only computations. That is to say, after all the above initializing steps, all the data variables are stored in consistent state.

Finally, the exact computation is carried out by repeatedly generating a Sub-DDAG and then executing it. Users of KASEN should write their program with only external operators, in which the user-given function can be implemented by static functions, functors, or even lambda functions. More specifically, in contrast of operating the whole vector/matrix, the working place of each user-given function is related to only one certain index. For example, the type signature of map operator is "*std::function<void(void *in, void *out)>*". As we can see, both the input and output elements are passed by their pointers, in which *in* points to an element from the input vector and *out* points to the corresponding output element that has the same index. The fusion of these user-given functions is simple: *1)* A Sub-DDAG is constructed for each *IterOnVec* operation, which can be executed by using a two-layer loop. The out layer of this loop iterates over all the elements and the inner layer iterates the related user-given function in a topological order.; *2)* As for *MxM(T)*, it is just a standard GEMM operation that can also be implemented by a two-layer loop (one for data elements and one for user-given functions, with arbitrary order).

As for the communication primitives, both *gather* and *scatter* are implemented by invoking an *MPI_Alltoallv* operation over the *MPI_COMM_WORLD* communicator that contains all the workers. Each worker will find out "where are the slaves of the masters assigned to this worker" at the preparing phase and reserves all

[7]After computation, KASEN will first transfer all the output variables to *consistent* state, and then dump them to the NFS.

these master-slave relations. Thus only one round of *MPI_Alltoallv* is used for each *gather/scatter* operation.

8. EVALUATION

We evaluate our optimizer by comparing it with a prevalent implementation that executes each primitive in isolation and always synchronizes data lazily. Specifically, we first present the results of our quantitative analyses on various algorithms, which computes the exact amount of memory read/write, buffer allocation, and network traffic saved by our optimizer. Then, we implement these algorithms and evaluate the overall performance improvement. Finally, we compare the performance of our system with existing array-based frameworks.

8.1 Quantitative Analyses

PageRank As discussed in Section 5.5, each iteration of the PageRank algorithm can be fused into an *MxM* operation (Sub-DDAG 2 in Figure 4) and a single 4-In/3-Out *IterOnVec* operation (Sub-DDAG 4 in Figure 4). Thus, if we use N and S to designate the number of masters and slaves of each vector respectively, M to designate the number of non-zero elements of the matrix, the standard implementation of PageRank needs to read $60(N+S)+24M$ bytes data and write $40(N+S)+8M$ bytes data per iteration[8]. In contrast, only $20N+24M$ bytes memory read and $16N+8M$ bytes memory write per iteration are needed after the optimization. As for buffer allocation, a total of $16N$ bytes of intermediate vectors are saved.

Take the Web graph from Google programming contest[18], which contains $9*10^5$ vertices and $5*10^6$ edges, as an example. When partitioned into 16 nodes with hybrid-cut, there are about $3*10^6$ slaves for each vector. Thus we can calculate that about *204MB* memory read and *134MB* memory write are saved per iteration, which accounts for 68% and 45% of the total memory read and write, respectively.

Black-Scholes In order to reduce the implementation complexity, most of the computing frameworks only provide unitary and binary primitives. Although all the possible computation can be built up by using these low-level primitives, it is inefficient without optimizations. As an illustration, the Black-Scholes algorithm is often used in financial applications for pricing put and call options. An idiomatic implementation of this algorithm usually contains eight *Map* and six *ZipWith* operations. Even if programmers do the optimization manually, one *Map* and six *ZipWith* operations are still needed (because of the limitation of supported primitives). However, both of these two kinds of programs can be fused into a single 2-In/2-Out *IterOnVec* operation by our optimizer. Thus, a total of 90% and 84% memory read/write can be saved, compared to the idiomatic and partially-fused implementation respectively.

K-means Another merit of our optimizer is enabling programmers to write *performance-equivalent* programs that are more thinkable and hence it puts less burden on programmers. For example, the K-means clustering algorithm is a popular unsupervised machine learning technique. With KASEN, K-means can be straightforwardly implemented by using a for loop that executes one *Map* and $k-1$ *Zipwith* operations for finding the nearest mean of each point, and then use k pairs of *Zipwith* and *Reduce* to calculate new means. Our optimizer will fuse all these $3k$ operations into one $(k+1)$-In/k-Out *IterOnVec* operation and hence has a performance that is comparable with manually optimized code; approximately $(8k-3)/(8k-1)$ of the memory read/write are reduced.

[8]Each element of the *degree* vector is 4 bytes, while the other vector elements are 8 bytes. As for the matrix, since we use the COO format, each non-zero element has 8 bytes.

L-BFGS Finally, we have also conducted a quantitative analysis on the L-BFGS algorithm, which is one of the most frequently used optimization methods in practice [29] and relatively more complex than the above three examples. Specifically, given an optimization problem with H samples and W features, L-BFGS only needs to store m vectors of length W to approximate the Hessian implicitly, and hence requires much less memory than the traditional BFGS algorithm. According to our evaluation, a total of $2H+(10m+7)W$ elements read and $H+(4m+4)W$ elements write can be saved per iteration by the optimization. Take the Ads CTR prediction task described by Chen et al [11] as an example, in an industry-level logistic regression problem, there are about 10^9 samples (H) and features (W); the data matrix is sparse and contains only $23H$ non-zero elements; m is set to 10; and all the elements are stored in 8-byte double. In such a case, our optimizer can save about *872GB* memory read and *360GB* memory write per iteration, which accounts for 48% and 63% of the total memory read/write respectively. Moreover, our optimizer can also save a pair of *gather* and *scatter* operations of a *Vector* with length W per iteration by using the hybrid transition instead of the lazy transition strategy. Since there are a total of 6 *gather/scatter* operations per iteration before the optimization, this is a 33% reduction on network traffic, disregarding the exact partitioning method.

8.2 Overall Performance

In order to further validate the results of our quantitative analyses, we implement the corresponding algorithms with our implementation of KASEN and evaluate them on a cluster of 16 machines. Each of the machines is outfitted with Intel Xeon 4-core CPU (@ 2.50GHz), 28 GB memory space, and a Mellanox ConnectX 3 InfiniBand NIC (MT27500 @40 Gbps).

PageRank For evaluating the performance of PageRank algorithm, we use several real-world graphs collected by the SNAP project [1]. Table 3 presents our evaluation results, which are calculated by running the algorithm for 30 iterations and calculating the average speedup between the original version of the program and the optimized one; about $1.22X-2.20X$ speedup is achieved by our optimizer.

Table 3: The evaluation results of PageRank.

Dataset	# of workers	Speedup
web-Google [18] (sparsity ≈ 6)	1	1.22X
	4	1.46X
	16	1.70X
web-ND [3] (sparsity ≈ 4)	1	1.42X
	4	1.60X
	16	1.89X
wiki-Talk [17] (sparsity ≈ 2)	1	1.56X
	4	1.75X
	16	2.20X

As we can see from the table, the speedup has a positive correlation with the number of nodes. This is because that the number of slaves is increased when the graph is partitioned into more subgraphs. Moreover, the speedup is bigger if the graph is sparser, which is also consistent with our quantitative analysis result.

Black-Scholes The Black-Scholes algorithm consists of only one kind of operation (i.e., iterating over the data) and does not require any cross-worker communications. Thus, in Table 4, we only present evaluation results of running the program on one node. Compared to the idiomatic form of the program, about 1.8X speedup can be achieved by our optimizer. Even if compared with the partially-fused version of the program, there is still about 1.5X

Table 2: Comparison on performance between KASEN, Spartan and PowerGraph. All the experiments are done with 16 workers, i.e., one worker per machine. The number of nodes used for evaluating K-Means is 160M. Times are given in milliseconds per iteration.

App.	PageRank			Black-Scholes		K-Means	
	Web-Google	Web-ND	Wiki-Talk	160M Stocks	480M Stocks	10 Clusters	20 Clusters
KASEN	234	38	31	1904	5201	115	190
SPARTAN	1174	436	217	3234	9023	729	1397
POWERGRAPH	332	237	425	—	—	5674	8747

speedup. These numbers of speedup are relatively smaller than the reduction on memory read/write only. This is because the cumulative distribution function (CDF) calculated in the algorithm is complex and hence the Black-Scholes algorithm is not memory-bounded.

Table 4: The evaluation results of Black-Scholes.

# of stocks	Idiomatic	Partial	Optimized	Speedup
160M	50.12*s*	42.08*s*	29.04*s*	$1.45X - 1.72X$
480M	160.77*s*	134.98*s*	83.24*s*	$1.62X - 1.93X$

However, besides the performance improvement, our optimizer of KASEN can also mitigate the memory pressure due to the significant reduction on temporary vectors. Taking the Black-Scholes algorithm as an example, our optimizer reduces near 71% of the peak memory consumption compared to the idiomatic program (56% reduction for the partially-fused version).

K-means Similar to the Black-Scholes algorithm, since the K-means algorithm uses *Vector* and *Shared* variables only, all the data elements are equally partitioned (with no slaves) to each machine. Thus the normalized speedup does not change a lot when testing with different number of machines and different numbers of data points. In Figure 5, we only illustrate the results of running the program on all the 16 machines and with a total of *160M* input data points. As we can see from the figure, about $4.14X - 5.82X$ speedup is achieved by our optimizer, and it can be even bigger if a larger number of clusters (i.e., k) is tested.

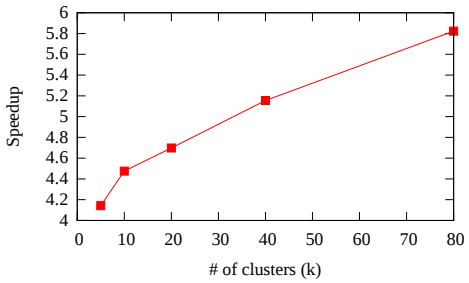


Figure 5: Speedup of the K-means algorithm.

L-BFGS We further evaluate our implementation of the L-BFGS algorithm with a synthetic dataset that is randomly generated. Specifically, the dataset contains $1.3 * 10^8$ samples and $7 * 10^6$ features; with sparsity 23. According to our evaluation, our optimizer can achieve about $1.59X - 3.54X$ speedup.

8.3 Comparison with Other Systems

Our optimizer is independent with the exact implementation of underlying systems. However, in order to justify that the current simple implementation of our prototype is already efficient, we have also compared it with Spartan [15] and PowerGraph [14]. Spartan is a recently proposed array-based framework, which is reported [15] to be faster than similar systems such as Presto [28] and SciDB [8]. However, as we can see from Table 2, our system is up to 11× faster than Spartan, especially when the application incurs considerable network traffic (e.g., PageRank). The main reason of this speedup is that *1)* We use *hybrid-cut* [10] to partition the sparse matrices, while Spartan is based on tiling and its matrices can only be partitioned with block-based partitioners. This will lead to at most ten times larger network traffic; *2)* In our system, data in matrices are ordered in Hilbert order [6], which provides better data locality; Finally, *3)* to the best of our knowledge, Spartan is the only framework that has tried to improve a program's performance by transparently altering user-written programs. But, as we have discussed in Section 1.2, the method used in Spartan is constrained in both primitives' type and the number of their inputs. Hence, it is less effective than our algorithm. For example, since the optimizer of Spartan cannot merge *Map* operation with *ZipWith*, it can only fuse the idiomatic program of Black-Scholes into four *Map* and six *ZipWith* operations (even more operations than the partial-fused version of program we tested). This defect is more apparent when optimizing K-means, the implementation of K-means provided by Spartan consists of many *Map* operations that only part of their inputs are overlapped which cannot be fused by Spartan's optimizer. As for PowerGraph, it has been shown by recent investigations [25] that array-based systems can be much faster than vertex-centric systems, and this result is repeated by our experiments.

9. OTHER RELATED WORK

Distributed array operation Many distributed array libraries [22, 23] have been built for providing highly optimized implementations of specific operations. Algorithms used by these libraries or even themselves are building blocks of modern array-based frameworks. However, although they are very fast for operations that have been built-in, they do not provide optimization that considers multiple operations.

Distributed array frameworks For retaining the expressive power of array operations on large datasets, many systems have been proposed to provide an array interface over modern data-parallel computing frameworks. However, these tools are usually reported to be inefficient [28, 24], because they are not build on a specific underlying infrastructure that originally designed for arrays (e.g., MLib/SystemML are based on Spark [30, 7] and SciDB [8] requires frequent disk I/O). As a result, there is currently a rising trend of building computing frameworks that are directly based on distributed array operations (e.g., Presto [28], MadLINQ [24], CombBLAS/KDT [9, 20], TensorFlow [2]). However, since static analysis deals poorly with ambiguities in source code [4], most of these frameworks adopt a naive implementation of executing each operation in isolation and in the exact order written by programmers, which is suboptimal. In contrast, KASEN defines a restricted programming model and hence enables us to design an optimizer that is both automatic and effective.

Local array optimizer Besides the researches on distributed array programs, there is a long tradition of developing domain-specific compilers that optimize local array programs. For example, MaJIC and FALCOM compilers for MATLAB optimize matrix expressions according to algebraic laws [21, 12], but they do not perform any fusion-based optimization. Among these technologies, BTO [5] is the most closely related work of KASEN, which also proposes a fusion-based technique for optimizing sequences of BLAS operations. However, it is only a local array optimizer and hence is ignorant of data synchronizations that are required in a distributed environment. As a result, if it is directly used on a distributed program, the outputted program may incur suboptimal amount of network traffic or even produce incorrect results.

10. CONCLUSION

In this paper, we present an array-based programming model KASEN, which defines a strict data, communication and computation model and hence enables us to measure and optimize programs automatically. According to our investigation, KASEN is sufficient for implementing many important machine learning and graph algorithms, and more importantly, the optimizer shipped with KASEN can achieve significant reduction on both memory read-/write, buffer allocation and network traffic. Evaluation based on our implementation of KASEN shows that the optimizer of KASEN can achieve $1.22X - 5.83X$ speedup for various algorithms.

Acknowledgements. This Work is supported by Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), the National Basic Research (973) Program of China (2014CB340402), Chinese Special Project of Science and Technology (2013zx01039002002).

11. REFERENCES

[1] `http://snap.stanford.edu/data`.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[3] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the world-wide web. *Nature*, 401:130–131, 1999.

[4] P. Anderson. The use and limitations of static-analysis tools to improve software quality. *CrossTalk: The Journal of Defense Software Engineering*, 21:18–21, 2008.

[5] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. SC '09, pages 59:1–59:12.

[6] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model. SPAA '07, pages 61–70.

[7] M. Boehm, D. Burdick, A. Evfimievski, B. Reinwald, P. Sen, S. Tatikonda, and Y. Tian. Compiling machine learning algorithms with systemml. SOCC '13, pages 57:1–57:1.

[8] P. G. Brown. Overview of scidb: Large scale array storage, processing and analysis. SIGMOD '10, pages 963–968.

[9] A. Buluç and J. R. Gilbert. The combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25:496–509, 2011.

[10] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. EuroSys '15, pages 1:1–1:15.

[11] W. Chen, Z. Wang, and J. Zhou. Large-scale L-BFGS using MapReduce. NIPS '14, pages 1332–1340.

[12] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: A MATLAB interactive restructuring compiler. LCPC '95, pages 269–288.

[13] M. P. Forum. MPI: A Message-Passing Interface Standard. Technical report, 1994.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. OSDI'12, pages 17–30.

[15] C.-C. Huang, Q. Chen, Z. Wang, R. Power, J. Ortiz, J. Li, and Z. Xiao. Spartan: A distributed array framework with smart tiling. USENIX ATC '15, pages 1–15.

[16] S. Lerner, D. Grove, and C. Chambers. Composing Dataflow Analyses and Transformations. POPL '02, pages 270–282.

[17] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. WWW '10, pages 641–650.

[18] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6:29–123, 2009.

[19] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, pages 503–528, 1989.

[20] A. Lugowski, D. Alber, A. Buluç, J. R. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A flexible open-source toolbox for scalable complex graph analysis. SDM '12, pages 930–941.

[21] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. ICS '99, pages 434–443.

[22] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *THE JOURNAL OF SUPERCOMPUTING*, 10:10–197, 1996.

[23] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39:13:1–13:24, 2013.

[24] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud. EuroSys '12, pages 197–210.

[25] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. SIGMOD '14, pages 979–990.

[26] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. CLOUDCOM '10, pages 721–726.

[27] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[28] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. EuroSys '13, pages 197–210.

[29] S. J. Wright and J. Nocedal. *Numerical optimization*, volume 2. Springer New York, 1999.

[30] 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. NSDI'12, pages 2–2.

[31] Y. Zhang and J. Yang. Optimizing i/o for big array analytics. *Proc. VLDB Endow.*, pages 764–775, 2012.