Confluence: Improving Iterative Distributed Operations by Key-Dependency-Aware Partitioning

Feng Liang, *Member, IEEE*, Francis C. M. Lau, *Senior Member, IEEE*, Heming Cui, *Member, IEEE*, and Cho-Li Wang, *Member, IEEE*

Abstract—A typical shuffle operation randomly partitions data on many computers and generates significant network traffic. This traffic is pronounced in iterative distributed operations where each iteration invokes a shuffle operation. Our key observation is that data of different iterations are related according to the transformation logic of distributed iterations. In an iteration, if input data have been partitioned by previous iterations to the same node where they are supposed to be processed in this iteration, unnecessary network traffic is avoided in the shuffle operation.

We model general iterative distributed operations as the transform-and-shuffle primitive and define a powerful notion named Confluence key dependency to precisely capture the data relation in the primitive. We further find that by binding key partitions between different iterations based on the Confluence key dependency, the shuffle network traffic can always be reduced with a predicable value. We implement the Confluence system, which automatically generates an efficient key partitioning scheme for iterative distributed operations. Confluence is widely applicable to real-life applications in diverse fields, and both analytical and experimental results show that Confluence can greatly reduce or even totally eliminate the shuffle network traffic.

Index Terms—Spark; Shuffle; Key Dependency; Iterative Distributed Operation; Partitioning

1 Introduction

Distributed applications consisting of iterative distributed operations are pervasive in fields of graph computing [1], [3], database query processing [4], [5], [6] and machine learning [7], [8]. To process big data, distributed frameworks like Hadoop [12]and Dryad [10] are often used. Several distributed paradigms have been developed on top of these frameworks to match various styles and scenarios of computing on large-scale data [4], [5]. MapReduce [12], one of the most popular distributed paradigms, saves output data in file systems, such as HDFS [13]. This is inefficient for tasks of iterative distributed operations, where one iteration reuses the data from the previous iterations. Sharing of data between iterations is usually done through a shuffle operation.

Moreover, the problem of heavy shuffle network traffic greatly impacts the performance of distributed operations. Most distributed paradigms include the shuffle operation, which transfers intermediate output data between computer nodes for the following processing. The intermediate data are stored in disks in some traditional distributed paradigms. The shuffle operation may invoke a large amount of network traffic and disk I/O, and sometimes even dominate the job completion time. For instance, shuffle-heavy jobs generate a large volume of network traffic. A study based on the Yahoo! work trace has revealed as much as 70% of jobs are shuffle-heavy [14]. Though some work claimed that the network for shuffle may be less a bottleneck [2], other studies showed that the shuffle completion time can account for as much as 33% of overall completion time [15], [16]. The heavy shuffle workload problem is pronounced in iterative distributed operations, where shuffle operations transfer large volumes of data between every two iterations.

To improve shuffle performance, several iterative distributed paradigms (e.g., Spark [18]) reuses intermediate data across iterations by storing them in memory. However, the volume of intermediate data and their network traffic is still enormous.

Our key observation is that we can exploit the powerful notion of key dependency and partition data to locations where it will be processed in latter iterations to greatly reduce the shuffle network traffic. Keys are relevant between different iterations and how data are partitioned between iterations can greatly affect the network traffic in shuffle operations. If we have a key partitioning scheme such that data needed for the following computing iterations have been partitioned to the same node before shuffling, we can reduce or even eliminate the cross-node traffic of shuffle operations (we call that the "shuffle size" in the rest of this paper). In pervasive hashed-by-key partitioning schemes, the shuffle size of each map-and-reduce iteration would be almost of the same size as the map output data. The total shuffle size of multiple iterations can be very large.

For instance, the MapReduce-style matrix multiplication algorithm is a two-iteration (stage) operation whose shuffle size can be minimized with a "better" key partitioning scheme. Each entry of the matrix product $C = AB$, where $A \in R^{m \times k}$ and $B \in R^{k \times n}$, is denoted as $C_{ij} = \sum_{p=1}^{k} A_{ip}B_{pj}$. Stage 1 calculates $A_{ip}B_{pj}$, $p = 1, 2, ..., k$.

• *F. Liang, F.C.M. Lau, H. Cui and C.-L. Wang are with Department of Computer Science, The University of Hong Kong.*
E-mail: F. Liang- loengf@connect.hku.hk, F.C.M. Lau- fcmlau@cs.hku.hk, H. Cui- heming@cs.hku.hk, C.-L. Wang- clwang@cs.hku.hk

Stage 2 obtains C_{ij} by summing $A_{ip}B_{pj}$, $p = 1, 2, ..., k$. Fig. 1 shows an example of 2×2 matrix multiplication in a four-node cluster. The data entries are in the key-value format, where the keys are (i, j, p) or (i, j). By using the default hashed key partitioning scheme in both shuffle iterations (Fig. 1(b)), the data entries representing the addends for a particular sum need to be transferred to the same node in the second shuffle iteration. For example, to get $(1, 1) \rightarrow 19$, entry $(1, 1) \rightarrow 14$ in Node 2 needs to be transferred to Node 1 to join entry $(1, 1) \rightarrow 5$. However, we know that for any particular i and j and different p's, all entries with keys (i, j, p) will be transformed to entries with key (i, j) in the second iteration (Fig. 1(c)). A better scheme should have partitioned all entries with key (i, j, p) with a different p to a local node in the first iteration, so that no entry needs to be transferred across nodes in the second shuffle iteration. For example, by assigning $(1, 1, 1) \rightarrow 5$ and $(1, 1, 2) \rightarrow 14$ to the same node (Node 1) in the first shuffle iteration, the result entry $(1, 1) \rightarrow 19$ can be obtained locally in Node 1 in the second iteration without cross-node data transfer.

In the above example, obviously, there is a relation between keys (i, j, p) in the first iteration and keys (i, j) in the second iteration. Knowing this relation, partitioning data entries to decrease/minimize the shuffle becomes possible.

We define the notion of Confluence key dependency to describe this key relation. The Confluence key dependency depicts the logic of distributed application, by specifying how an input dataset is transformed to an output dataset. The Confluence key dependency is simple in form and easy to define in iterative distributed operations. We further find that by using the technique of key partition binding based on the Confluence key dependency, the shuffle size always decreases with a predictable value, and sometimes can be even eliminated.

We present the Confluence Key Partitioning (CKP) scheme for iterative distributed operations. Based on a set of key dependencies, CKP binds key partitions in different iterations and reduces the shuffle size to the maximum extent. Most distributed computing paradigms, such as Spark [18] and Twister [19], provide an interface for adding a user-defined partitioner for shuffle operations. We implement CKP in Spark and programmers can apply CKP to various kinds of distributed applications by add a single line of code in the program. We illustrate with analysis and experiment to real-life applications to show that by applying the CKP scheme, the shuffle size of multiple computing iterations reduces significantly with a predictable value.

To the best of our knowledge, we are the first to represent the logic of iterative distributed operations by key dependency, and to reduce the shuffle network traffic with a predictable value by considering the key dependency across multiple iterations in a key partitioning scheme. Our major contributions are listed as follows.

- **Precise**: We invented the simple yet powerful notion of Confluence key dependency to precisely capture the behavior of data transformation in iterative distributed operations. We defined the transform-and-shuffle primitive that can generally describe all iterative distributed operations.
- **Efficient**: We presented the Confluence Key Partitioning (CKP) scheme to greatly reduce the shuffle size in iterative distributed operations. CKP is efficient and does not introduce extra workload skewness.
- **Universal**: We showed that CKP is easily applicable to diverse real-life distributed applications. We provided a simple interface in distributed frameworks that allows programmers to apply CKP with one single line of code.

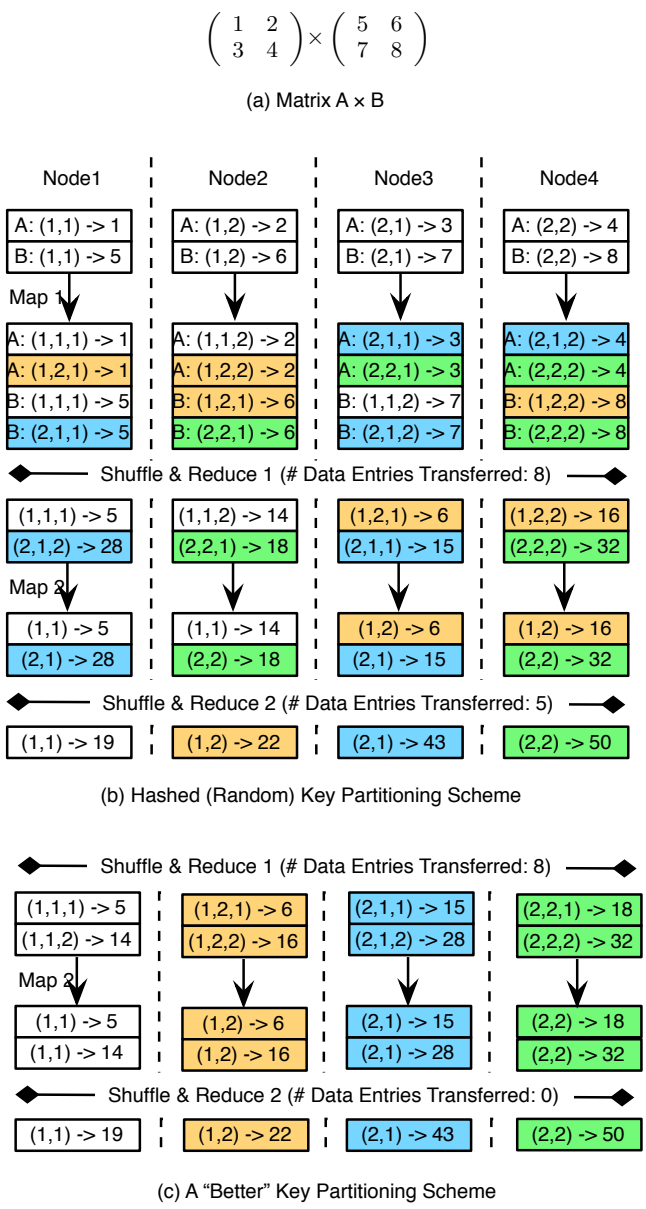


Fig. 1. An Example of the MapReduce-style Matrix Multiplication Algorithm with Different Key Partitioning Schemes

The rest of the paper is organized as follows. In Section 2, we introduces the model of iterative distributed operations and the iterative shuffle size, gives the definition of Confluence key dependency, and shows the calculation of the shuffle size of the random key partitioning scheme based on Confluence key dependency. We present the Confluence key partitioning scheme in Section 3. Section 4 introduces the application of CKP and its limitations, and Section 5 dis-

cusses implementation details of CKP in distributed frameworks. Evaluation results of the performance are presented in Section 6. We discuss the related work in Section 7 and conclude the paper and suggest the future work in Section 8.

2 Model and Definition

2.1 Iterative Distributed Operations

Datasets of large volumes cannot be stored in a single computer node and are often separated into several partitions, where each partition is distributed to a node in a computer cluster. We refer to a dataset which is separately stored in several nodes as a distributed dataset. A function whose inputs include a distributed dataset is called a distributed operation. Each entry of the distributed dataset can be represented as a key-value pair.

In many distributed paradigms, there are often several iterations involving different distributed operations being applied to an input dataset before final results are obtained. Each iteration usually takes outputs of the previous iteration as the input dataset.

For iterative distributed operations, we refer to the general iterative transform-and-shuffle operations. In each iteration of the distributed operation, the "transform" primitive performs one or more transformation operations on input datasets in each local node. The "shuffle" primitive would re-partition the datasets by transferring data entries with the same keys to designated locations and grouping entries of the same key to a key-value pair, as the input for the next iteration. This *value* is actually a list of the values that associate with the key, and the output of each data entry of the shuffle primitive is also called as the key-value-list pair in this paper. Each iteration of distributed operation is in the form of a series of transformations plus at most one shuffle operation.

Note that the division of distributed operations by transform-and-shuffle is similar to the paradigm of MapReduce [12], which divides the operation into two primitives: map and reduce. They are slightly different but can be equivalent in the context of iterative distributed operations. The shuffle primitive is equivalent to the shuffle operation of the reduce primitive in MapReduce. It does not involve any transformation on data. While the transform primitive is equivalent to the reduce operation (excluding the shuffle operation) in the reduce primitive plus the map primitive of the next iteration of a MapReduce operation. The shuffle primitive is also called the shuffle operation, and the transform primitive is called the transform operation or, simply, transformation.

Most iterative distributed paradigms such as the in-memory paradigm Spark [18] and the distributed database query engine Hive [4] can be equated to iterative transform-and-shuffle operations. The matrix multiplication example in Section 1 follows the traditional MapReduce interpretation. From now on in this paper, we use the transform-and-shuffle interpretation when we describe iterative distributed operations.

2.2 Iterative Shuffle Size

We model the overall shuffle size of iterative distributed operations and discuss the time complexity of obtaining the optimal key partitioning scheme that minimizes the overall shuffle size. Table 1 lists some symbols we use in this paper for reference.

TABLE 1
Symbol Reference

Symbol	Description
A_i	Input dataset for iteration i
A'_i	Transformation Output in iteration i
D_k	Set of input entries of the transform operation (in the specified iteration) that have key k
D'_k	Outputs of the transform operation with D_k as the input

For iterative distributed operations, the result of the i_{th} iteration can be obtained recursively by:

$$A'_i = T_i(A_i)$$

$$A_{i+1} = S_i(A'_i)$$

, where A_i is the output of iteration $i-1$ as well as the input of iteration i, T_i is the transform operation of iteration i that operates on A_i locally without changing the partition locality, and S_i is the shuffle operation of iteration i that takes the output A'_i of the transform operation as the input. A_i and A'_i define not only key-value pairs of datasets, but also their partition locality. The keys of A'_i and A_{i+1} are the same, but their partition localities are different. In the first iteration, when i equals 1, A_1 represents the raw input datasets before any processing.

Suppose in the shuffle operation of iteration i, in order to shuffle transformation outputs of an entry $a \in A_i$ to its partition location, the shuffle size is a function of a and A_{i+1}: $g_i(a, A_{i+1})$. The returned value of function g_i can be different in different key partitioning schemes for A_{i+1}. The shuffle size of iteration i is:

$$G_i = \sum_{a \in A_i} g_i(a, A_{i+1}).$$

If m iterations are required to compute the final result, the overall accumulated shuffle size for obtaining the final result is

$$G = \sum_{i=1}^{m} G_i = \sum_{i=1}^{m} \sum_{a \in A_i} g_i(a, A_{i+1}). \quad (1)$$

If contents (including key-value pairs and the partition locality) of $A_i (i = 1, 2, ..., m)$ are already known, to find out the optimal key partitioning scheme that minimizes the overall shuffle size G, we need to exhaustively explore the possible key partition schemes of all the key-value pairs across all the iterations. By calculating the overall shuffle size of each scheme, the optimal solution is the scheme with the minimal size. The time complexity of the exhaustive method is $O(n^{\sum_{i=1}^{m+1} |A_i|})$, where n is the number of nodes in the cluster and $|A_i|$ is the number of data entries of A_i, which indicates that it is a complex problem.

In fact, we usually do not know the contents of A_i and cannot explore the partition scheme for the next iteration until the program has actually finished the i_{th} iteration. Due to this limitation, the idea of finding out the optimal partition schemes for all the iterations to minimize the overall data transfer size G is infeasible.

2.3 Confluence Key Dependency

We define the Confluence key dependency, or for convenience, simply called key dependency in the rest of this paper, to represent transformation logic in every iteration of a distributed operation. Given a key-value pair as the input to an iteration of a distributed operation, the transform operation will generate a (or a set of) new key-value pair(s) based on its logic.

Formally, for a set of input key-value pairs and output key-value pairs in an iteration of a distributed operation, we define their key dependency as

$$k \Rightarrow_{pr} f(k) \tag{2}$$

, where k is a key, f is a mapping function of k, and $f(k)$ is another key. For now, we do not care about what f is like, and consider $f(k)$ is an arbitrary key. The symbol "$\Rightarrow_{pr}$" indicates the fact that for any data entry whose key is k, after the transform primitive, the probability of that the key of an output data entry is $f(k)$ is pr. By definition, we have:

$$k_0 \Rightarrow_{pr} f(k_0) \equiv \forall \langle k, v\rangle \in \{\langle k, v\rangle | k = k_0\} : p(k_{t(\langle k,v\rangle)} = f(k)) = pr$$

, where $t(\langle k, v\rangle)$ is the transform operation on $\langle k, v\rangle$ and outputs a set of key-value pairs, k_d is one of the keys in dataset d, and p is the probability function.

In the key dependency, we call k the input key, $f(k)$ the mapped key, and pr the dependency probability. This key dependency can be read and grasped as "input key k has a pr chance to map to mapped key f(k)" after transformation in the corresponding iteration.

A programmer is supposed to be well acquainted with the logic of the transform operation t. By saying that we know the key dependency for a specific iterative distributed operation, it means that we are able to calculate pr by the probability function p for specific k and $f(k)$. In the matrix multiplication example, we can easily deduce the key dependency of the second iteration: $\forall i, j, p \in domain : (i, j, p) \Rightarrow_{1.0} (i, j)$. Specially, the key mapping function is $f((i, j, p)) = (i, j)$.

2.4 Random Key Partitioning

We will show that the random key partitioning scheme(RKP) generates larger shuffle size than almost any other key partitioning schemes. In RKP, a key is assigned a random partition location in the shuffle operation. RKP is widely used in popular distributed paradigms. For example, Spark [18] uses the hash-based key partitioning scheme, which partitions keys based on their hash values.

As mentioned in in Section 2.2, the shuffle size $g_i(a, A_{i+1})$ to shuffle transformation outputs of entry a is different in different key partitioning schemes. Given a key dependency implicated by a distributed operation, we discuss how to calculate $g_i(a, A_{i+1})$ in a specific key partitioning scheme. We set off from RKP.

In a cluster of n nodes, using RKP, the probability of a transformed key-value pair is assigned and partitioned to another node (which indicates a network transfer) is $(n - 1)/n$.

In iteration i, let D_k denote all transform operation inputs that have key k, D'_k denote transform operation outputs after transforming D_k, and $|D|$ denote the volume of dataset D. Given a set of key dependencies $k \Rightarrow_{pr_j} k_j, \sum_j pr_j = 1$, the expected shuffle size to transfer D'_k in RKP is:

$$\sum_{a \in D_k} g_i(a, A_{i+1}) = \sum_j \frac{n-1}{n} pr_j |D'_k| = \frac{n-1}{n}|D'_k| \approx |D'_k|. \tag{3}$$

In cases that n is large in a large cluster, the approximately equal sign in Formula 3 can be thought of as an equal sign. Note that the key dependency does not affect the shuffle size of an iteration in RKP at all. For any key partitioning scheme, the shuffle size cannot be larger than the volume of the shuffle operation. That is, $\sum_{a \in D_k} g_i(a, A_{i+1}) <= |D'_k|$ for any key partitioning scheme. By summing up the shuffle sizes for all keys in all iterations by Formula 1, we can easily conclude that the overall shuffle size of RKP is almost larger than any other key partitioning scheme.

3 Confluence

Although it is hard to find the optimal partition solution to minimize the overall shuffle size, we discover that if the transformation logic of a specific distributed operation is already known, the shuffle size can be reduced to the maximum extent by exploring the key dependency of different iterations.

The key dependency is to represent the logic of transform operations, while the key partitioning scheme is to indicate the logic of shuffle operations. By the observation that the RKP shuffle size can be expressed by the terms in the key dependency, we have the intuition that we can use the key dependency to discover other key partitioning schemes that produce a smaller shuffle size.

In this section, we present the Confluence Key Partitioning scheme that makes use of the key dependency to reduce the shuffle size by binding key partition locations. We also analyze that the Confluence Key Partitioning scheme does not increase the workload skewness.

3.1 Key Partition Binding

Binding partitions of input keys to mapped keys based on the key dependency reduces the shuffle size. Assume that we have a set of key dependencies $k \Rightarrow_{pr_j} k_j, \sum_j pr_j = 1$ in an arbitrary iteration i. Now suppose that a key partitioning scheme X always partitions shuffle output data that have key k in the previous iteration $i - 1$ to the same computer node where shuffle output data that have key k_x in this iteration i will be partitioned. We say that key partitioning scheme X binds the partition of input key k to mapped key k_x in the previous iteration.

By the definition of the key dependency and recalling that the shuffle size only refers to cross-node data transfer, we easily deduct the following theorem.

Binding Theorem. *Suppose a key dependency $k \Rightarrow_{pr} f(k)$ in an iteration of a distributed operation, if the partition of key $f(k)$ was bound to k in the previous iteration, the shuffle size for transferring transformation output data that have key $f(k)$ after transforming D_k is zero in this iteration.*

Algorithm 1: Enhanced Key Partitioning Scheme X

```
In a specific iteration i, foreach key k do
    /* K is a key dependency set                    */
    K = {k ⇒_{pr_j} f_j(k) | j = 1, 2, ..., Σ_j pr_j ≤ 1} ;
    Find J such that ∀j, pr_j ≤ pr_J;
    Bind input key k to mapped key f_J(k);
end
```

The insight of the binding theorem is that if we know the key dependency for a dataset in an specific iteration, we can prepare the partition of the dataset in the previous iteration by the notion of binding keys so that the dataset will do local shuffling in this iteration and it will incur no cross-node traffic.

In such a scheme X, reusing the symbols in Section 2.4, by the binding theorem, the expected shuffle size to transfer transformation outputs of the D_k is: $\sum_{a \in D_k} g_i(a, A_{i+1}) = \frac{n-1}{n}(1 - pr_x)|D'_k|$. The shuffle size difference between scheme X and scheme RKP is $\Delta \approx pr_x|D'_k|$. The value of $|D'_k|$ is a constant for a given transform operation and input D_k. The larger pr_x is, the greater scheme X reduces the shuffle size than RKP does.

There are two constrains when binding key partitions. First, key partition binding often be used in all but the first iteration of a distributed operation. The reason is that most distributed frameworks do not support specifying the partition locations of input data loaded from a file system, which means binding key partitions in the first iteration is not feasible. But it could be used in the following iterations.

The second constrain is that any input key can be bound to at most one mapped key, because a key should have a unique partition. This constrain indicates for all key dependencies with the same input key, only one can be selected for key partition binding. These two constrains are consider as common sense when we design key partitioning schemes.

3.2 Confluence Key Partitioning

Surprisingly, by the binding theorem, we find that binding key dependency always reduces shuffle size as compared to RKP. Note that by binding key partitions, the shuffle size which is reduced is linearly proportional to the dependency probability. We can further enhance scheme X by applying key binding based on the key dependency with the maximum dependency probability for each key in an specific iteration (Algorithm 1).

A problem exists in the enhanced scheme X. The key dependency is usually provided by programmers. In just one iteration, programmers need find a distinguished key dependency set for every input key. The key dependency sets of different input keys can be different. The time complexity is the size of a key dependency set (usually small) times the number of keys (usually large), or roughly, O(# keys), which is usually large.

We presents the efficient Confluence Key Partitioning (CKP) scheme reduces or even eliminates the shuffle size in each iteration with as little as $O(1)$ time complexity. The procedure of applying CKP is shown in Algorithm 2. In an iteration, instead of finding a distinguished key dependency set for every input key, CKP finds one general key dependency set that is applicable to any input key.

Algorithm 2: Confluence Key Partitioning

```
foreach iteration i do
    /* Programmers decide that                      */
    b = Should iteration i apply key partition binding?
    if b = True then
        /* K is a key dependency set                */
        Find ∀k, K =
        {k ⇒_{pr_j} f_j(k) | j = 1, 2, ..., Σ_j pr_j ≤ 1} ;
        Find J such that ∀j, pr_j ≤ pr_J;
        ∀k, bind input key k to mapped key f_J(k);
    end
end
```

For example, in the second iteration of matrix multiplication, instead of finding key dependency $(1,0,0) \Rightarrow_1 (1,0)$ for input key $(1,0,0)$ and $(1,0,1) \Rightarrow_1 (1,0)$ for input key $(1,0,1)$, respectively, we can define $(1,0,p) \Rightarrow_1 (1,0)$ for any p in general. Or more generally, we can define $(i,j,p) \Rightarrow_1 (i,j)$ generally for any i, j, p.

A feature of CKP is that it allows a programmer to decide whether a specific iteration should apply key partition binding or not. In some iterations, it may happen that programmers fail to find the key dependency, or maximum dependency probability of the key dependency set is very small, and applying CKP is trivial. In these cases, CKP let the programmer to decide not to bind key partitions. Further discussion about these cases will be presented in Section 4.5. When a key dependency with a large dependency probability exists in an iteration, key partition binding can be applied to reduce shuffle size.

A question about the CKP algorithm is how to figure out the key dependency set in an iteration to apply key partition binding. The key dependency is be derived from the transform operation logic. In some cases, for input key-value entries with a specific key, the transform operation always generate output entries with predictable keys, regardless what the values are. The matrix multiplication algorithm falls in this case. Finding key dependency such cases is usually simple, and a programmer may start from the domain of the input keys and derive the general key dependency for each range of the domain. In some other cases, keys of the output entries are also affected by the values of the input key-value entries. In these cases, a programmer need also consider the key-value entries as a whole to deduct what mapped keys will be and what probability it is to generate an output entry with each mapped key. We will discuss more about finding the key dependency with real-life examples in Section 4.

There is an implicit in the CKP algorithm. For iterations that do not apply key partition binding, the input keys are randomly partitioned, just as in RKP.

CKP reduces the shuffle size in all iterations that apply key partition binding. The shuffle size reduction in each applied iteration compared to RKP is $\Delta \approx \sum_k pr_J|D'_k|$ (recalling the shuffle size reduction by key partition binding in Section 3.1). For other iterations that do not applied key partition binding, the CKP shuffle size is the same as that

of RKP. In all, CKP reduces the shuffle size by a percentage of pr_J in each iteration that applies key partition binding. When pr_J equals 1 in an iteration, the shuffle size is 0 in that iteration.

3.3 Workload Skewness Analysis

CKP does not increase the workload skewness in most applications. The workload of a node is the number of input entries for the transform operation in that node and the workload skewness of an iteration is the standard deviation of the workloads of different nodes in that iteration.

We compare workload skewness of CKP and RKP in iterations that apply key partition binding. In unbound iterations, the key partitioning policy of CKP is the same as that of RKP, and the workload skewness in these iterations is the same in both schemes. Suppose in one of such iterations, keys of input entries follow a distribution $Dist$. In RKP, these input keys have been uniformly partitioned, and the workload skewness is the standard deviation of $Dist$ times the total number of input entries times the average number of distinguished input keys in each node.

In CKP, in the binding iteration, suppose two conditions: 1) input keys are uniformly bound to mapped keys; 2) the mapped keys, which is also the input keys of the next iteration, will be uniformly partitioned in the shuffle operation. The first condition means that the expected number of distinguished input keys bound to every mapped key is the same. If both conditions are satisfied, it indicates that the input keys have also been uniformly partitioned in the previous iteration, and therefore, the workload skewness is the same as that of RKP. The second condition will be eventually satisfied, as the last iteration always randomly partitions keys. If the first condition is always true in all iterations, the second condition can be proved to be always true by induction from the last iteration back to previous iterations.

In all, if input keys are uniformly bound to mapped keys in all iterations, the workload skewness of CKP is the same as that of RKP. Luckily, this condition is true in most applications we observe. For example, in the second iteration of matrix multiplication, the number of input keys (i, j, p) that are bound to mapped key (i, j) is the dimension size of the matrix. In the MovieLensALS algorithm introduced later in Section 4, the expected number of input keys $(userId, itemId)$ that are bound to mapped key $userId$ is the average number of items all users have bought. In a few cases (e.g., KMeans to be introduced in Section 4) when this condition does not hold, we find that the workload skewness difference of CKP and RKP is so small that it could be ignored, regarding the shuffle size CKP reduces.

3.4 Affect of Inaccurate Key Dependency

Note that the programmers do not need to change the original logic of applications in order to apply CKP. Applying any key partition scheme, including CKP, will not affect the logic correctness of the applications, but only affect the traffic performance of shuffle (shuffle size and workload skewness).

An inaccurate key dependency means a key dependency with an inaccurate dependency probability. While a dependency probability of a key dependency implies how much it reduces the shuffle size as compared to RKP. When an inaccurate key dependency is used for binding key partitions in CKP, the only shuffle size effect is that the shuffle size reduction is related to the actual dependency probability, not to the inaccurate one. As to the workload skewness, the inaccurate key dependency may lead to a different key partition binding and unpredictable workload skewness may exist. But as CKP randomly partitions mapped keys in every iteration whose next iteration does not apply key partition binding, the workload skewness readjusts to the same as RKP then.

TABLE 2
CKP Shuffle Size Reduction in Different Distributed Operations

Distributed Operations	CKP Shuffle Size Reduction
MatrixMultiplication	$n^3 for R^{n \times n}$
MovieLensALS	# of $(user, item)$ entries
GoogleAlbum	# of album entries and user entries
PageRank	volume of link lists of all URL's in every join iteration
MultiAdjacentList	1/2 the RKP shuffle size in every iteration
KMeans	$\lvert A\rvert \cdot \sum_{i=j+1}^{m} p_{ij}$ [1] upper bound: $(m-j)\lvert A\rvert$ lower bound: 0

[1] $|A|$ is the volume of points, m is the number of iterations.

4 Application

In this section, we show how CKP is applied in various iterative distributed applications, by digging out the key dependency with a large dependency probability in some iterations. We also discuss limitations of CKP when in some circumstances it is not applicable, and solutions to handle these limitations.

Luckily, we find that many applications have key dependency with a large dependency probability in some iteration(s). In the second iteration of the matrix multiplication algorithm, the transform operation does a many-to-one processing and the key dependency is obvious, $\forall$ key (i, j, p), $(i, j, p) \Rightarrow_{1.0} (i, j)$, which has 100% dependency probability for all the key dependencies. In some applications, the dependency probability is not 100% but is still large enough for applying CKP. We illustrate the techniques for finding the key dependency of iterative distributed operations with the real-life application examples from different fields, namely MovieLensALS, GoogleAlbum, MultiAdjacentList and KMeans. A summary of the CKP shuffle size improvements to the distributed applications discussed below is shown in Table 2.

4.1 MovieLensALS

MovieLensALS [22] is a recommendation application that uses the Alternating Least Squares (ALS) method in collaborative filtering in order to recommend product items to the users based on the user-item rating information. During the execution, it organizes the rating data into user-item blocks grouped by key "$(userId, itemId)$". In the successive iterations, it needs to reorganize the user-item block data by generating user blocks and item blocks grouped by key "$userId$" and key "$itemId$", respectively. This operation is done in two iterations as shown in Fig. 2(a).

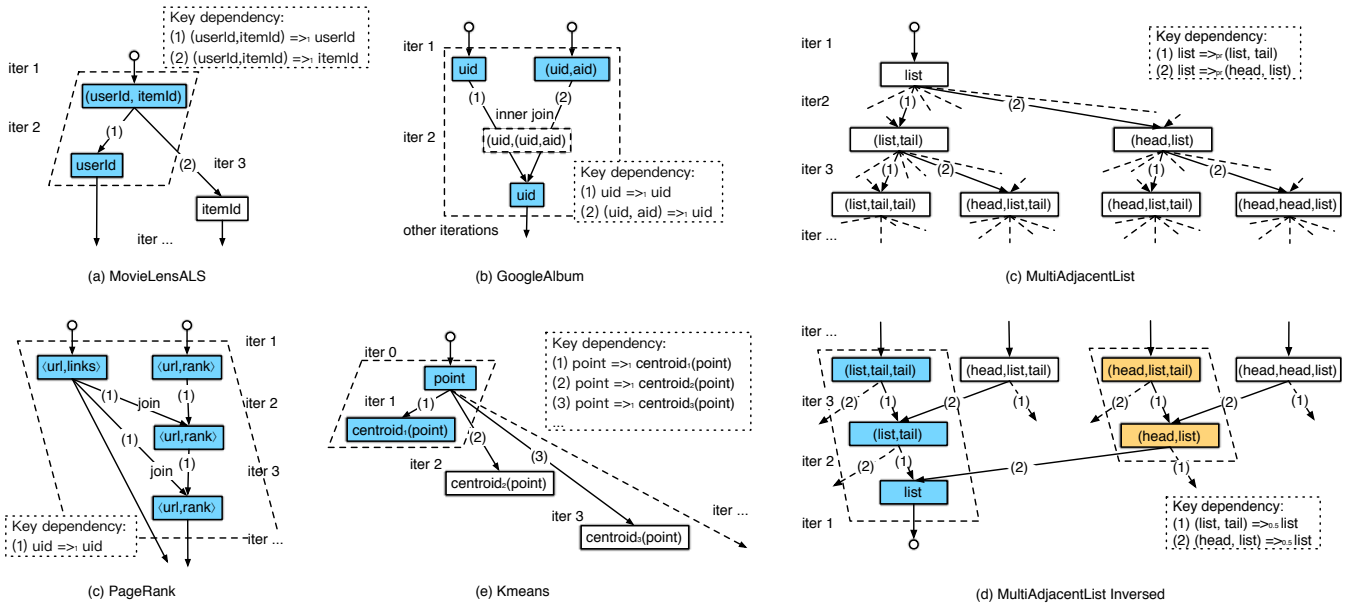


Fig. 2. The Transformation Lineage of Keys between Iterations and the Key Dependency of Different Applications. Key Partition Binding is Applied to Keys in Iterations Wrapped in Dashed Rhombus Boxes

A key dependency from input key $(userId, ItemId)$ exists in iteration 2 and iteration 3, respectively. As these two iterations use the same copy of input dataset, by the constrain of key partition binding, we can apply key partition binding in either iteration, but not in both. If we select key dependency $(userId, itemId) \Rightarrow userId$ in iteration 2, the shuffle size generating the user block in iteration 2 is 0, and the shuffle size generating the item block in iteration 3 is the same of that of RKP.

4.2 GoogleAlbum and PageRank

CKP can usually be applied to *join* operations in distributed databases. For example, in a Google album application [26], an entry in table Users is identified by primary key "uid" and an entry in table Albums by "(uid, aid)" (Fig. 2(b)). When an *inner join* operation interleaves these two tables and gets albums for each user, the transform operation processes interleaved data with key "$(uid, (uid, aid))$" to album data with key uid. The key dependency here is $(uid, (uid, aid)) \Rightarrow_1 uid$. In CKP, when binding the interleaved key $(uid, (uid, aid))$ to key uid, it meaning both binding key uid in table Users to key uid (which is a trivial binding) and binding key (uid, aid) in table Albums to key uid.

We can also interpret this binding as putting album entries of the same user together in a local node before such an "inner join" operation. Spanner [26] uses the similar notion of hierarchical schema to maintain data locality in the distributed database. The hierarchical schema specifies data locations once when persisting data in storage, while CKP specifies the data locations iteratively during the computing procedure of a distributed operation.

Similarly in the famous PageRank algorithm, the link list of a URL is represented as $\langle url, links\rangle$, and the rank of a URL is $\langle url, rank\rangle$. These two datasets are joined to entries like $\langle url, (links, rank)\rangle$ in multiple iterations. The key dependency is $url \Rightarrow_1 url$ for both datasets. Note that the link list dataset is loaded from a distributed file system initially and it has not been partitioned by its key url. Without CKP, entries with key url_0 in the link list dataset need to be transfered to the partition of key url_0 in every iteration. In CKP, by binding partition of input key url to mapped key url, the link list dataset is partitioned by its key url first. Therefore, the link list and the rank with the same URL are on the same node in every iteration, and no shuffle network traffic incurs in these $join$ operations. This partition optimization is also suggested in the usage of RDD [18], we formalize it here by the concept of key dependency.

4.3 MultiAdjacentList

MultiAdjacentList [23] is a graph computing application which generates lists of different lengths in a graph by concatenating adjacent nodes to the head and the tail of a list, recursively. In every iteration, for a data entry $\langle L_i, value\rangle$, key L_i represents a list of length i and the value is a set of adjacent nodes to the head or the tail of the list. The transform operation generates a group of new lists of length $i+1$ by appending each head node and each tail node to it. Each new list acts as the key and its new head nodes and tail nodes act as the value. The iterative transformation of the data in algorithm MultiAdjacentList is shown in Fig. 2(c).

For any input key L_i in iteration i, key dependencies are $L_i \Rightarrow_{pr_j} (head_j, L_i)$ and $L_i \Rightarrow_{pr_l} (L_i, tail_l)$ for many different j's and l's, because a list L_i may have many adjacent head or tail nodes. Therefore, values of pr_{j1} and pr_{j2} can be small and benefits of applying CKP based on these key dependencies is little. But if we inverse the direction of key flow as in Fig. 2(d), we have new key dependencies with dependency probability of 0.5. The key dependencies of the inversed key flow reflects the fact that an entry with key $(L_i, tail)$ is 50% generated from entries with key L_i (and

another 50% from entries with key $(L_{i-1}, tail)$, where L_{i-1} is a sublist of L_i without the head node).

4.4 KMeans

KMeans [24] is a famous clustering algorithm in data mining. It groups vector points into k clusters where each vector point belongs to the cluster with the minimum mean value. This procedure is done in a dedicated number of iterations. With the initial k chosen points regarded as centroids of the clusters, in each iteration, every point is compared with the k centroids and is grouped into a cluster where the Euclidean distance between the point and the centroid is the smallest. After all points are grouped, each new cluster centroid is chosen in the next iteration by the mean of the points in that cluster. To calculate the mean of each cluster whose points are located in different nodes, the shuffle operation takes place to partition points of the same cluster to one node. A point is always transfromed to an entry with its cluster centroid as the key in every iteration.

Key dependency exists in each cluster iteration of the KMeans algorithm (Fig. 2(e)). In CKP, we choose a specific iteration j, and bind key partitions of the points based on the key dependency $point \Rightarrow_1 centroid_j(point)$ in iteration j. As all iterations use the same copy of the point dataset, the binding can be interpreted as: if a point is partitioned to a cluster in an iteration j, we bind its partition to this cluster afterwards, regardless of which cluster it will be actually grouped into in the following transform operations.

We can predicate the shuffle size after apply CKP. Suppose in iteration $i, i > j$, pr_{ij} is the probability of that any point is grouped into the same cluster as it was grouped into in a previous iteration j. CKP reduces the shuffle size by a percentage of pr_{ij} in each iteration i, $i > j$. The value of pr_{ij} is affected by the input dataset, initial centroids of the clusters, and the value of j, i.e., from which iteration on point partitions are bound to clusters. For example, the cluster a point is grouped into tends to be stabler in latter iteration. With a higher value of j, the value of $pr_{ij}(i > j)$ tends to be higher. But accordingly, the number of iterations that apply key partition binding is smaller. Programmers may need to tune the value and make a trade-off then. In our implementation in Section 5, we choose j to 1.

4.5 Limitations Applying Confluence

CKP can be widely applied to reduce the shuffle size of the iterative distributed paradigms that compute and store intermediate transformation outputs in local storages (including disk and memory), like Spark and Twister. However, we find it not applicable or directly applicable in the following circumstances and we suggest solutions for these problems.

First, CKP is not usually applicable for single-iteration distributed operations. CKP requires that each key of a transformation input dataset have a partition location. But in single-iteration distributed operations, input datasets are usually attained from a distributed file system (e.g., HDFS), where data locations are not organized by their key partition locations. An alternative solution is to modify the mechanism of distributed file systems on choosing the partition locations of data blocks when storing data to a file. If a data entry is able to declare the preferred location as its key partition location, and if the distributed file system write the entry to its preferred location, the key partition locations of the dataset sustain and CKP can be applicable.

Second, CKP is not applicable when intermediate transformation data change their partition location in iterative distributed operations. CKP assumes that data would sustain the partition location during transformation, which should be a local processing of data in each node without cross-node data transfer. This assumption is violated when performing iterative distributed operations in distributed paradigms that relocate the data during transformation. For example, if we use MapReduce [12] to implement a iterative distributed operation, output data of the reduce phase are stored into HDFS, where the data may be transfered to other nodes for persistence due to its internal mechanism. A solution to this problem to use iterative distributed paradigms, such as Twister [19] and Haloop [17], that retain the intermediate transformation data in local nodes.

The third circumstance is that programmers fail to find the key dependency or the maximum dependency probability of the key dependency set is small and trivial in all iterations. The reason of the former case is either that the programmers do not have enough information on the logic and input data of the distributed application to deduce the key dependency, or that the key dependency is not clear by the nature of the distributed application. the latter case happens when input keys are diversely mapped to lots of mapped keys during transformation. Applying CKP in this case gains trivial shuffle size improvement. A solution to this circumstance relies on programmers to discover key dependencies with large dependency probability, usually in transform operations that reduce the dimensions of key tuples. An example is the mapping from key $(userId, itemId)$ to $(userId)$ in the MovieLensALS algorithm. When all dependency probabilities are small, the shuffle size improvement of CKP can be very limited.

5 Implementation

This section introduces implementation details of binding partitions to executors to facilitate CKP and automating CKP in distributed frameworks.

5.1 Binding Partition and Executor in Spark

In the implementation of Spark, the partition is only a logical location where the task executes the dataset, while the logical locations of tasks are not tied up with the actual(physical) positions of the executors. In other words, the tasks working on the same partition are not guaranteed to be assigned to the same executor (or the same computer node) across different iterations, in which case shuffling of data is still needed.

To amend the mismatch of the logical partition and the actual position of the executor, we implement the new feature in Spark to allow the binding of the task partitions and the executors. The binding can be done in the hash manner. In the default settings of the task scheduler of Spark, when an executor becomes free, the task scheduler selects the task with whichever partition from the head of task queue. The selected task will run in that executor. Now, if the partition-executor-binding feature is turn on, when the task scheduler

TABLE 3
Key Mapping Functions of ConfluencePartitioner in Different Iterative Distributed Operations

Distributed Operations	Key Dependency	Key Mapping Function
MatrixMultiplication	$(i,j,p) \Rightarrow_1 (i,j)$	(a:Any) => a match {case (_ , _, _) => (a._1, a._2)}
MovieLensALS	$(userId, itemId) \Rightarrow_1 userId$	(a:Any) => a match {case (_ , _) => a._1}
GoogleAlbum	$(uid, (uid, aid)) \Rightarrow_1 uid$	(a:Any) => a match {case (_ , _) => a._1}
PageRank	$url \Rightarrow_1 url$	(a:Any) => a match {case _ => a}
MultiAdjacentList	$(list, tail) \Rightarrow_{0.5} list$	(a:Any) => a match {case s:String => {s.split(" ")(0) }}
KMeans	$point \Rightarrow_1 centroid_1(point)$	(a:Any) => a match {case _ => closestPoint(a, centroids$_1$)[1]

[1] $centroid_1(point)$ stands for the index of the centroid point that the $point$ is grouped to in the first iteration. closestPoint(a, centroids$_1$) is the function which returns the cluster of point "a", that is, the index of the point in "centroids$_1$" that is in the shortest Euclidean distance with "a". The "centroids$_1$" are the cluster centroids of the first iteration.

selects a task for the free executor, it selects the task whose task partition hash code is the same as the hash code of the free executor. The hash manner binding ensures that the tasks of the same partition will always be allocated to the same executor in different iterations. Finding the first task that first hashed maps the executor from the task queue, the time complexity of scheduling each task is $O(N)$, where N is the number of the executors in each computing iteration.

The partition-executor-binding implementation will not introduce negative side effects into the system in the following three aspects: 1) The order to run the pending tasks of the same iteration does not affect the completion time of each computing iteration; 2) In the hash manner, each executor is expected to run the same number of tasks in each computing iteration; 3) The task scheduling overhead is trivial and negligible, which is only linear to the number of executors.

5.2 Confluence Partitioner Interface

We provide an interface in Spark to allow programmers apply CKP in a distributed application with one single line of code. To apply CKP, the user-definable partitioner offered by pervasive distributed frameworks allows programmers to implement specific partitioners for their distributed applications.

We make one step further by implementing a class named ConfluencePartitioner, which only requires the programmer to provide the key mapping function of the key dependency with the maximum dependency probability. Now recall that in Section 2.3, the key mapping function f is a function of the input key k and returns the mapped key $f(k)$ in a key dependency.

In ConfluencePartitioner, to calculate the partition of a input key-value entry in the shuffle operation, the key mapping function is applied to the input key, and the mapped key is returned. The mapped key is hashed based on the number of partitions and the hash value decides the partition of this input entry. In this way, all input entries whose returned key by the mapping function are the same are partitioned to the same node.

To apply CKP in a shuffle operation, programmers only need add a ConfluencePartitioner with the corresponding mapping function as the partitioner. The code looks like:

```
data.shuffleOp (new ConfluencePartitioner
   (numPartitions, mappingFunction));
```

, where "data" is often an RDD in Spark and "shuffleOp" is any shuffle function that repartition distributed data, such as $reduceByKey$ or $join$. Fig. 3 shows the code of using ConfluencePartitioner to apply CKP in MovieLensALS and KMeans. Only the single line of code is added in MovieLensALS to apply CKP. In KMeans, the partition of a point are bound to its cluster centroid in a specific iteration j. This line of code binds the partition of key $(userId, itemId)$ to that of key $userId$ when creating the user-item block data.

```
/**** Original MovieLensALS ****/
val userPart = new HashPartitioner(16)
ratings.mapPartitions{...}
       .groupByKey(userPart).mapValues{...}

/**** MovieLensALS applying CKP ****/
val userFunc = (a:Any) => {a match {case
    (_,_) => a._1}}
val userPart = new
    ConfluencePartitioner(16, userFunc)
ratings.mapPartitions{...}
       .groupByKey(userPart).mapValues{...}

/**** Original KMeans ****/
while(notConverge && i < maxIter){
  val closest = points.map(...)
  centroids = closest.reduceByKey(...)...
}

/**** KMeans applying CKP ****/
while(notConverge && i < maxIter){
  if (i == j){
    val mapFunc = (a:Any) => {a match {
      case _ => closestPoint(a, centroids)}}
    val part = new ConfluencePartitioner(16,
        mapFunc)
    points = points.reduceByKey(part,
        (a,b)=>1).cache()
  }
  val closest = points.map(...)
  centroids = closest.reduceByKey(...)...
}
```

Fig. 3. Code Comparison after using the ConfluencePartitioner Interface to apply CKP in MovieLensALS and KMeans

Table 3 lists the mapping functions of different distributed operations mentioned above in Scala programming language.

With ConfluencePartitioner, the programmers can apply CKP without knowing the details of implementing the self-defined partitioner and repeatedly writing different self-defined partitioners in each iteration of different distributed operations.

TABLE 4
Benchmark Settings

Benchmark	Input Data	Runtime Setting
MatrixMultiplication	matrix type: $Z^{1000\times1000}$	Nil
MovieLensALS	21,622,187 ratings from 234,934 users on 29,584 movies	32 user blocks and 32 item blocks
MultiAdjacentList	25,000,000 vertexes with average in/out-degree being 2	4 iterations
KMeans	64,534,480 wikipedia page visit records	32 centers, 10 iterations

6 Evaluation

We conduct experiments on a physical testbed to compare the performance of CKP with the default RKP scheme in aspects: the improvement on shuffle sizes of different iterations, the workload skewness of executors, and the scalability.

6.1 Testbed and Benchmarks

The testbed consists of 18 computer nodes of the Gideon-II cluster in HKU [25], where each node is equipped with 2 quad-core, 32 GB DDR3 memory and 2 $\times$ 300 GB SAS hard disks running RAID-1. In the setting of the YARN cluster, one node takes the role of the name node of HDFS and one other acts as the resource manager of YARN. The remaining 16 nodes are configured as both HDFS data nodes and YARN node managers, and are connected to an internal non-blocking switch with GbE ports. Spark is deployed on top of the YARN cluster with 16 executor, where each executor runs with 8 GB memories.

Several benchmarks that are representative ones of their fields are used to evaluate the performance of CKP. Unless further specified, the input data sizes and the running setting of the benchmarks are listed in Table 4. CKP is compared with RKP as the baseline as RKP is the default and pervasive key partitioning scheme in distributed frameworks. Other key partitioning scheme are sometimes used to balance the workload of the executors for specific applications and datasets. These key partitioning schemes are for special purpose and we do not compare CKP with them here.

6.2 Metrics

We measure the shuffle size of each distributed computing iteration (or alternately, stage) of the benchmarks. The shuffle size of each iteration is the sum of the cross-node data transfer size of the shuffle operation of all executors in that iteration. This metric depicts the performance of CKP in reducing the shuffle size. Besides, the standard deviation of input workloads of the executors is measured as the metric to evaluate the workload skewness of the key partitioning schemes. This metric is also measured in each iteration of the benchmark. The scalability of CKP is measured by the overall shuffle size of all iterations that apply key partition binding, with different volumes of input data.

6.3 Shuffle Size Improvement

Results of the shuffle size and the workload skewness of each iteration (or stage) after applying CKP and RKP in all the benchmarks are shown in Fig. 4, respectively.

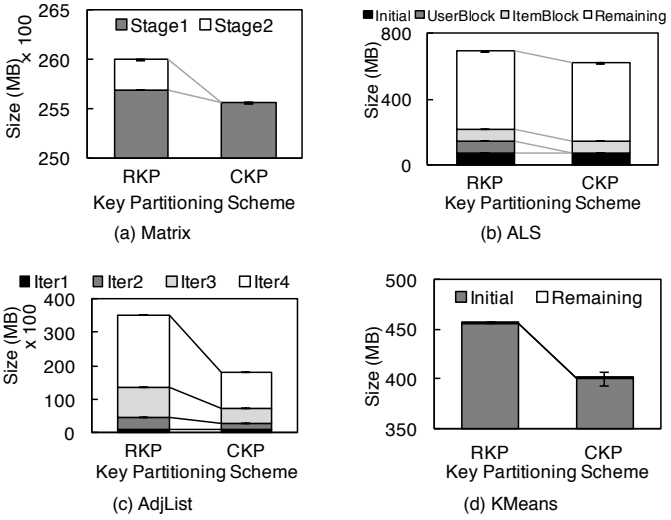


Fig. 4. The Shuffle Size and Standard Deviation of Workloads of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

In MatrixMultiplication, CKP totally removes the shuffle size of stage 2, and the shuffle size of stage 1 remains close to that of RKP (Fig. 4(a)). Note that, regardless of the key partitioning scheme, the volume of data in stage 2 is small compared to stage 1 because data have been locally merged by the distributed framework to compress data in stage 2 [18].

The benefit of applying CKP on iterative distributed operations that reduce the key dimensions can be read in Fig. 4(b). In MovieLensALS, after applying CKP, the shuffle size for generating the user block is 0, while the shuffle size for generating the item block remains almost the same as that of RKP. As a result, the CKP shuffle size for generating these two block is half of that of RKP. Because the other iterations of the distributed operations cannot be improved by CKP, the overall shuffle size CKP can reduce for this application is about 11%.

After applying CKP in MultiAdjacentList benchmark, the shuffle size of each iteration is decreased by half beginning from iteration 2 (Fig. 4(c)). The reason is that by binding partitions of key $(list, tail)$ to key $list$, only data with key $(head, list)$ need to be shuffled in each iteration, which is about 50% of the total data volume.

In KMeans benchmark, points are bound to centers they are grouped into in iteration 1. by partitioning the points into clusters they belong to in the first iteration, CKP decreases the overall shuffle size by 12% (Fig. 4(d)). This decrease in shuffle size is contributed by avoiding the repartition of the points that are stable to their clusters, although not all points are fixed to their clusters in every

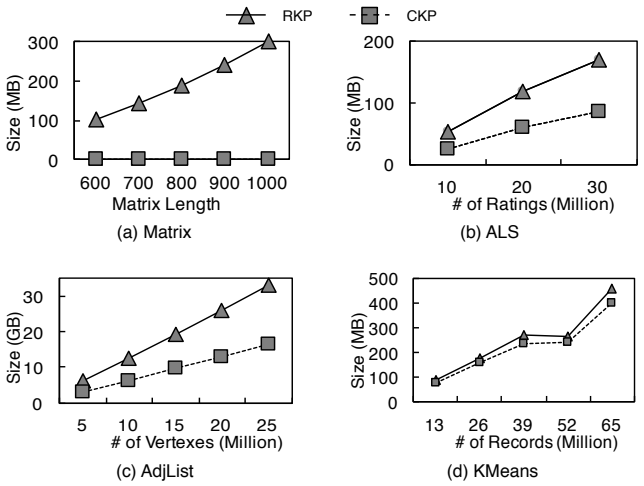


Fig. 5. Overall Shuffle Size of the Key Partition Binding Iterations in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks with Different Input Size

iteration. This shuffle size is influenced by the input dataset, initial centroids of clusters, and from which iteration the points are bound to the centroids. All these factors affect the dependency probability in the key dependency.

Note that the shuffle sizes of the other iterations that do not apply CKP remains almost unchanged, which means that CKP can decrease the shuffle size of distributed iterations that have the confluence key dependency without introducing extra workloads to the other iterations. How CKP can decrease the overall shuffle traffic depends on how large the volumes of data are in iterations that apply key partition binding, compared to the overall data volume of the application. For MultiAdjacentList, the CKP can be applied to half of the operations, either appending to the head or to the tail. While for MovieLensALS, CKP can only be applied to the operation that arrange the user-item blocks, which is a relatively small portion of the overall application. Still, it indicates the potential of applying CKP in the complicated iterative distributed applications.

6.4 Workload Skew

The workload skewness is denoted as the standard deviation bar in Fig. 4. As expected, the workload skewness (standard deviation of workloads) of the key partition binding stages of MatrixMultiplication (Fig. 4(a)), MovieLensALS (Fig. 4(b)) and MultiAdjacentList (Fig. 4(c)) are all close to 0. In the KMeans benchmark, the workload skewness of CKP is larger than that in RKP (Fig. 4(d)). The reason is that the number of points belonging to each cluster varies but the number of clusters (32) is not large enough compared to the computer nodes (16) to balance the workloads of the nodes. If the number of clusters is larger, the number of points bound to each cluster is smaller. As each cluster centroid is randomly partitioned, the workload skewness will become smaller. Still, the standard deviation of CKP (7 MB) is small as compared to the mean shuffle size of the executors (25 MB).

6.5 Scalability

The overall shuffle sizes of iterations that apply key partition binding with different input sizes in various benchmarks are shown in Fig. 5. In MatrixMultiplication, the overall shuffle size of RKP increases linearly as the matrix length grows, while that of CKP is always 0. CKP can totally remove the shuffle size of stage 2 in MatrixMultiplication. In the other benchmarks, the shuffle sizes of CKP always keep in a fixed (or stable) ratio to those of RKP, e.g., 50% in both MovieLensALS and MultiAdjacentList and around 88% in KMeans. The result shows that CKP scales well with different volumes of input data. The CKP decreases the shuffle size greatly (about 50%) in one iteration out of many iterations in the MovieLensALS application, and a little (about 12%) but in every iteration the KMeans application.

7 Related Work

Iterative distributed computing: Some works [17], [18], [19], [29] improved distributed frameworks for iterative distributed operations by extending programming interfaces to support multiple transform and shuffle phases. The general idea was to cache the data of each iteration in memories instead of in disks to reduce I/O overheads. However, these works on distributed frameworks did not concern with the heavy network workload during shuffle operations. Inheriting the benefit of fast memory I/O rate, CKP decreases shuffle sizes and alleviate network workloads by considering the key dependency in multiple related iterations.

Parallel&Distributed algorithms optimization: Lots of work has been conducted on the optimization of parallel&distributed machine learning algorithms [30] and science computing [31], [32], including matrix multiplication methods [33], [34]. Generally, they focused on optimizing the algorithm itself by decomposing tasks to increase task parallelism or removing the synchronization boundary between I/O-intensive tasks and compute-intensive tasks. Sarma et al. [35] represented the number of data entries generated from the map input for a distributed application with a given parallelism factor as communication cost. To decrease the communication cost for a specific distributed application, one need redesign the logic of the program itself, operating with the proper parallelism factor. The term of communication cost is slightly different from the shuffle size in this paper, where the communication cost is the transfer size of data between different reduce workers (which can be on the same node), while the shuffle size is transfer size of data between different computer nodes. An iterative distributed application can first be designed with the minimum communication cost for each iteration, if possible, and then apply CKP to decrease the shuffle size. ShuffleWatcher [36] scheduled the locality of map tasks and reduce tasks to decrease shuffle size for the single iteration of a MapReduce job, but it did not provide a solution for iterative distributed operations.

Logic-Aware Shuffle Partitioning: Some similar works [27], [28] also looked into the logic of programs and considered the data relation across iterations to avoid unnecessary shuffle network traffic. But the methods either lacked a precise model abstraction so that its applicable area was limited to aggregate-like operations [28], or built

a complicated model that made its usage difficult [27]. The Confluence key dependency model is simple in form and makes it possible to apply CKP in a single line of code. The dependency probability easily predicts the effect of shuffle size reduction.

Datacenter networking for shuffle: In datacenters, various networking scheduling algorithms were proposed to improve the shuffle throughput for different performance goals [15], [37], [38], [39]. CKP does not concern with the underlying network level. However, by decreasing shuffle workloads, shuffle operations applying CKP can work on these shuffle-optimized datacenter networks seamlessly and gain larger performance increase.

Skew&Straggler: Distributed computing paradigms like MapReduce adopted the data locality principle, either to place computing tasks close to locations of the processing data in priority to avoid data migration [12], [40], or to avoid an unbalanced allocation of workloads by considering compute skew problems [41], [42]. They concerned with the placing of map tasks whose required input data are self-contained. As the input data of each shuffle task are distributed across the cluster, such a task-to-data approach cannot help in shuffle tasks. CKP binds key partitions of different iterations based on the key dependency, which eliminates shuffle traffic of multiple iterations while not impacting the balance of workloads. Aaron et al. [43] addressed the problem of stragglers in iterative machine learning. In the case of successive key partition binding, as CKP randomly partitions mapped keys in every iteration whose next iteration does not apply key partition binding, the workload skewness always readjusts to balanced.

8 Conclusion and Future Work

We have presented the Confluence Key Partitioning (CKP) scheme, the first work on reducing the shuffle size of iterative distributed operations based on the key dependency. The key dependency precisely captures the logic of iterative distributed operations and CKP partitions data across different iterations by using the technique of key partition binding. CKP greatly reduces the overall shuffle size with a predictable value while not introducing the workload skewness side effect for a variety of distributed operations in fields such as scientific computing, machine learning, data analysis.

In the future, we will try to explore methods to automatically discover the key dependency of data among different iterations. A potential feasible solution is data flow tracking. With a set of sampled data, we can keep track of how data are flowing between nodes across different iterations, and thus find out the key dependency of these sampled data. The challenge is that how to expand key dependency of the sample data to the whole dataset. The unrepresentative sample sample data affects the accuracy of the dependency probability. Luckily, as we have discussed, CKP survives the problem inaccurate dependency probability.

Acknowledgments

This work is supported in part by a Hong Kong RGC CRF grant (C7036-15G).

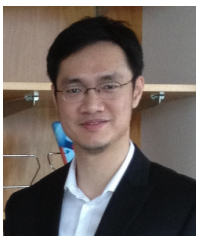
References

[1] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2010, pp. 135–146.
[2] K. Ousterhout et al., "Making Sense of Performance in Data Analytics Frameworks," *NSDI*, vol. 15, pp. 293-307, 2015.
[3] Y. Lu et al., "Large-scale distributed graph computing systems: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 281–292, 2014.
[4] A. Thusoo et al., "Hive: A warehousing solution over a map-reduce framework," *Proceeding of VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.
[5] Y. Yu et al., "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 1–14.
[6] M. Armbrust et al., "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2015, pp. 1383–1394.
[7] Y. Low et al., "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.
[8] T. Kraska et al., "Mlbase: A distributed machine-learning system," in *Conference on Innovative Data Systems Research (CIDR)*, vol. 1, 2013, pp. 2–1.
[9] V. K. Vavilapalli et al., "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th ACM Annual Symposium on Cloud Computing*, New York, NY, USA, 2013, pp. 5:1–5:16.
[10] M. Isard et al., "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, New York, NY, USA, 2007, pp. 59–72.
[11] T. White, "Hadoop: The definitive guide", *O'Reilly Media Inc.*, 2012.
[12] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[13] K. Shvachko et al., "The hadoop distributed file system," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–10.
[14] Y. Chen et al., "The case for evaluating mapreduce performance using workload suites," in *IEEE 19th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, 2011, pp. 390–399.
[15] M. Chowdhury et al., "Managing data transfers in computer clusters with orchestra," in *Proceedings of the ACM SIGCOMM Conference*, New York, NY, USA, 2011, pp. 98–109.
[16] M. Al-Fares et al., "Hedera: Dynamic flow scheduling for data center networks." in *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation*, vol. 10, 2010, pp. 19–19.
[17] Y. Bu et al., "Haloop: efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.
[18] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.
[19] J. Ekanayake et al., "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 810–818.
[20] R. Sakellariou and H. Zhao, "A hybrid heuristic for dag scheduling on heterogeneous systems," in *Proceedings of 18th IEEE International Parallel and Distributed Processing Symposium*, 2004, pp. 111–.
[21] H. Zhao and R. Sakellariou, "Scheduling multiple dags onto heterogeneous systems," in *Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium*, 2006, pp. 14–.
[22] "Movielensals spark submit 2014." [Online]. Available: https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html
[23] "Multiadjacentlist benchmark." [Online]. Available: https://github.com/liangfengsid/MultiAdjacentList
[24] "Spark kmeans benchmark." [Online]. Available: http://spark.apache.org/docs/latest/mllib-clustering.html
[25] "Hku gideon-ii cluster." [Online]. Available: http://i.cs.hku.hk/%7Eclwang/Gideon-II/

[26] J.C. Corbett et al., "Spanner: Googles globally distributed database," *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 251-264, 2012.
[27] J. Zhou et al., "Incorporating partitioning and parallel plans into the SCOPE optimizer," *IEEE 26th International Conference on Data Engineering*, pp. 1060-1071, 2010.
[28] J. Zhang et al., "Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions," *NSDI*, vol. 12, pp. 22-22, 2012.
[29] T. Gunarathne et al., "Scalable parallel computing on clouds using twister4azure iterative mapreduce," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1035-1048, 2013.
[30] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.
[31] M. Kiran, A. Kumar, and B. Prathap, "Verification and validation of parallel support vector machine algorithm based on mapreduce program model on hadoop cluster," in *IEEE International Conference on Advanced Computing and Communication Systems*, 2013, pp. 1–6.
[32] T. Mensink et al., "Metric learning for large scale image classification: Generalizing to new classes at near-zero cost," in *Computer Vision–ECCV 2012*. Springer, 2012, pp. 488–501.
[33] A. Buluc and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.
[34] G. Ballard et al., "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proceedings of the 24th annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2012, pp. 193–204.
[35] A. D. Sarma et al., "Upper and lower bounds on the cost of a map-reduce computation," *Proceedings of the VLDB Endowment*, vol. 6, no. 4, pp. 277–288, 2013.
[36] F. Ahmad et al., "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *2014 USENIX Annual Technical Conference*, Philadelphia, PA, Jun. 2014, pp. 1–13.
[37] A. Greenberg et al., "Vl2: a scalable and flexible data center network," in *Communication of the ACM*, vol. 54, no. 3, 2011, pp. 95–104.
[38] L. Popa et al., "Faircloud: sharing the network in cloud computing," in *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication*, 2012, pp. 187–198.
[39] A. Shieh et al., "Sharing the data center network," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011, pp. 309–322.
[40] M. Zaharia et al., "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 29–42.
[41] Y. Kwon et al., "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010, pp. 75–86.
[42] Y. Kwon et al., "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012, pp. 25–36.
[43] A. Harlap et al., "Addressing the straggler problem for iterative convergent parallel ml," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, New York, NY, USA, 2016, pp. 98–111.
[44] Spark job scheduling: Dagscheduler. [Online]. Available: https://spark.apache.org/docs/1.4.0/job-scheduling.html

Feng Liang received the BS degree in software engineering from Nanjing University in 2012, and the PhD degree in computer science from The University of Hong Kong in 2017. His research interests are mainly on distributed file systems, distributed computing, machine learning, and formal methods for distributed systems. He is recently undertaking the research project on distributed deep learning. [homepage] i.cs.hku.hk/%7Efliang

Francis C.M. Lau received his PhD in computer science from the University of Waterloo in 1986. He has been a faculty member of the Department of Computer Science, The University of Hong Kong since 1987, where he served as the department chair from 2000 to 2005. He is now Associate Dean of Faculty of Engineering, the University of Hong Kong. He was a honorary chair professor in the Institute of Theoretical Computer Science of Tsinghua University from 2007 to 2010. His research interests include computer systems and networking, algorithms, HCI, and application of IT to arts. He is the editor-in-chief of the Journal of Interconnection Networks. [homepage] i.cs.hku.hk/%7Efcmlau



Heming Cui is an assistant professor in Computer Science of HKU. His research interests are in operating systems, programming languages, distributed systems, and cloud computing, with a particular focus on building software infrastructures and tools to improve reliability and security of real-world software. [homepage] i.cs.hku.hk/%7Eheming

Cho-Li Wang is currently a Professor in the Department of Computer Science at The University of Hong Kong. He graduated with a B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1985 and a Ph.D. degree in Computer Engineering from University of Southern California in 1995. Prof. Wangs research is broadly in the areas of parallel architecture, software systems for Cluster computing, and virtualization techniques for Cloud computing. His recent research projects involve the development of parallel software systems for multicore/GPU computing and multi-kernel operating systems for future manycore processor. Prof. Wang has published more than 150 papers in various peer reviewed journals and conference proceedings. He is/was on the editorial boards of several scholarly journals, including IEEE Transactions on Cloud Computing (2013-), IEEE Transactions on Computers (2006-2010). [homepage] i.cs.hku.hk/%7Eclwang