Confluence: Speeding Up 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, generating possibly a significant amount of network traffic which often dominates a job's completion time. This traffic is particularly pronounced in iterative distributed operations where each iteration invokes a shuffle operation. We observe that data of different iterations are related according to the transformation logic of distributed operations. If data generated by the current iteration are partitioned to the computers where they will be processed in the next iteration, unnecessary shuffle network traffic between the two iterations can be prevented.
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 relations 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 by a predictable percentage. We implemented the Confluence system. Confluence provides a simple interface for programmers to express the Confluence key dependency, based on which Confluence automatically generates efficient key partitioning schemes. Evaluation results on diverse real-life applications show that Confluence greatly reduces the shuffle network traffic, resulting in as much as 23% job completion time reduction.

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

1 Introduction

Distributed applications consisting of iterative distributed operations are pervasive in fields such as 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, which offer various styles of distributed processing for large-scale data [4], [5], [18]. Most distributed paradigms include the shuffle operation, which transfers intermediate output data between computer nodes for next-stage processing.

The shuffle operation may invoke a large amount of network traffic, and often may even dominate the job's completion time, especially for shuffle-heavy jobs. The problem of heavy shuffle network traffic could greatly impact the performance of distributed operations. A study based on a Yahoo! work trace has revealed as much as 70% of jobs are shuffle-heavy [14]. Although some work claimed that the network part of a shuffle may unlikely be a bottleneck [2], other studies showed that the shuffle completion time can account for as much as 33% of the overall completion time [15], [16].

The heavy shuffle traffic problem is especially pronounced in iterative distributed operations, where shuffle operations transfer large volumes of data between every two iterations [18]. In the pervasive hashed-by-key (i.e., random) partitioning scheme, the shuffle size of each map-and-reduce iteration would be almost as large as the map output data [12], leading to excessive shuffle network traffic and deferred job completion time.

Our key intuition is that we can exploit the relation of data between iterations and partition the data in such a way that they would be assigned to locations where they will be processed in latter iterations. Keys of data have relevance over 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 by the following computing iterations have been partitioned to the same node(s) before shuffling, we can reduce or even eliminate the network traffic of shuffle operations (of which the amount of data being shuffled is referred to as the "shuffle size" in the rest of this paper).

For instance, the map-and-reduce-style [12] 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 $A_{ip}B_{pj}$, $p = 1, 2, \ldots, k$. Stage 2 obtains C_{ij} by summing $A_{ip}B_{pj}$, $p = 1, 2, \ldots, k$. 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

• *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

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 the same 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 data with key (i,j,p) in the first iteration and data with key (i,j) in the second iteration. Knowing this relation, partitioning data entries to decrease/minimize the shuffle operation 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 by a predictable percentage, and sometimes can be even eliminated. Compared to other partition optimization approaches [27], [28] which support specific types of data exchange patterns, the Confluence key dependency is more general and can be used in arbitrary types of distributed operations.

To exploit the Confluence key dependency in iterative distributed operations, we present the Confluence Key Partitioning (CKP) scheme. 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 adding just a single line of code in the program. We illustrate by analysis and experiments with real-life applications, showing that by applying the CKP scheme, the shuffle size of multiple computing iterations reduces significantly by a predictable percentage, e.g., 100% shuffle size reduction in an iteration of the MovieLensALS algorithm; as a result, CKP can reduce the completion time of a single iteration and the overall job completion time by as much as 57% and 23%, respectively.

To the best of our knowledge, this is the first proposal to represent the logic of iterative distributed operations by key dependency, and the first attempt to reduce the shuffle size by a predictable percentage by considering the key dependency across multiple iterations in a key partitioning scheme. The technical highlights of Confluence are listed as follows.

- **Precise**: We propose the simple yet powerful notion of Confluence key dependency to precisely capture the logic of data transformation in iterative distributed operations. We define the transform-and-shuffle primitive that can generally describe all iterative distributed operations.
- **Efficient**: We present the Confluence Key Partitioning (CKP) scheme which can greatly reduce the shuffle size in iterative distributed operations. CKP is efficient and would not introduce extra workload skewness.
- **General**: We show that CKP is easily applicable to diverse real-life distributed applications. We provide a simple interface in distributed paradigms, which allows programmers to apply CKP with one single line of code.

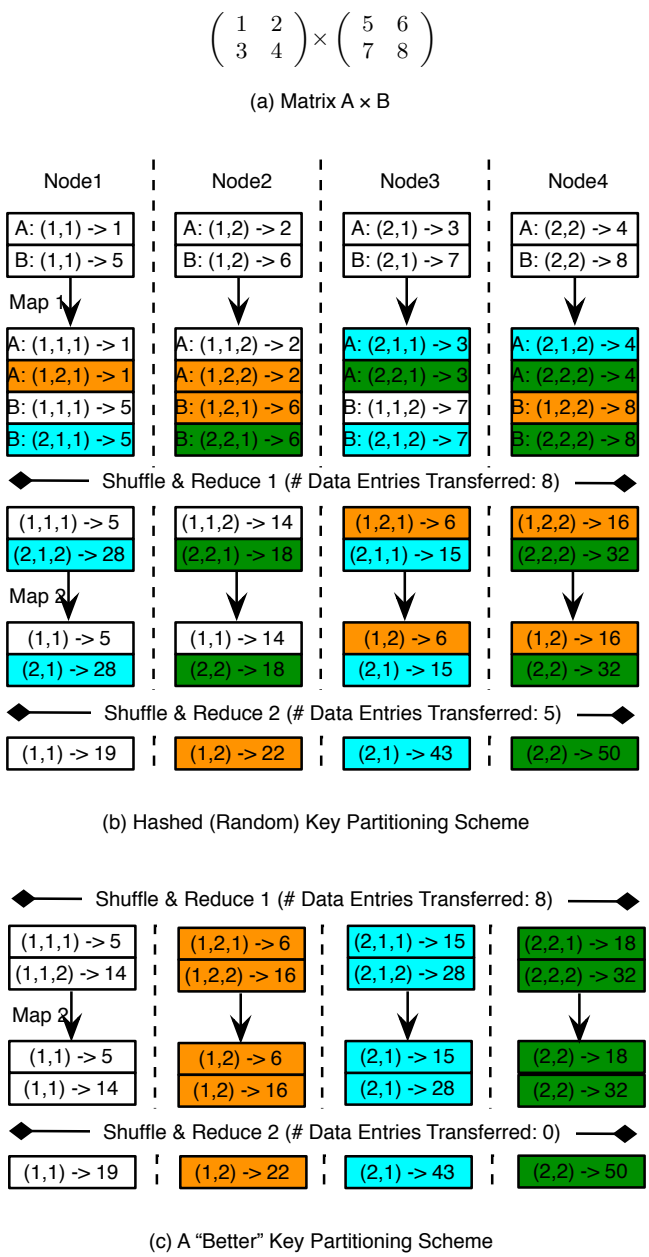


Fig. 1. An Example of the MapReduce-style Matrix Multiplication Algorithm with Different Key Partitioning Schemes

The rest of the paper is organized as follows. In Section 2, we describe the model of iterative distributed operations

and iterative shuffle size, define Confluence key dependency, and analyze the shuffle size of the random key partitioning scheme based on the Confluence key dependency. We present CKP in Section 3. Section 4 introduces the application of CKP and its limitations, and Section 5 discusses implementation details of CKP in distributed paradigms. Evaluation results of CKP are presented in Section 6. We discuss the related work in Section 7 and conclude the paper and suggest some possible 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 input includes a distributed dataset is called a distributed operation. Each entry of the distributed dataset can be represented as a key-value pair.

For iterative distributed operations, we refer to the general iterative transform-and-shuffle operations. In each iteration of the iterative distributed operation, the "transform" primitive performs one or more transformation operations on the input datasets in each local node. The "shuffle" primitive would re-partition the datasets by transferring data entries with the same keys to a designated location and grouping entries of the same key to a key-value pair, forming the input for the next iteration. Each iteration of a 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. There are slight differences, but in the context of iterative distributed operations, they can be treated as equivalent. 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 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 easy 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 $A_i (i = 1, 2, \ldots, m)$ 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 of all the key-value pairs in A_{i+1} 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 data partition for the next iteration until the program has actually finished the i_{th} iteration. Considering this limitation and the time complexity, the idea of finding out the optimal partition schemes for all the iterations in order to minimize the overall data transfer size G is infeasible.

2.3 Confluence Key Dependency

We define the Confluence key dependency (or for convenience, key dependency) for representing 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 $f(k)$ as 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 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 understood as "input key k has chance pr 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, we mean 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: $\forall i, j, p \in domain : (i, j, p) \Rightarrow_1 (i, j)$. Specifically, the key mapping function is $f((i, j, p)) = (i, j)$, and the dependency probability is 1. More details about calculating the dependency probability will be illustrate with application examples in Section 4.

2.4 Random Key Partitioning

We will show that the random key partitioning scheme (RKP) generates no smaller 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 being 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 $|D|$ and $|D'|$ denote the volume of dataset D and D' respectively. 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 big 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, $\sum_{a \in D_k} g_i(a, A_{i+1}) \leq |D'_k|$ 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 no less than almost 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 largest possible 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 deduced from the key dependency, we have the intuition that we can use the key dependency to discover other key partitioning schemes that would result in 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

We will show that 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 has always partitioned 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 this iteration i.

By the definition of the key dependency and recalling that the shuffle size only refers to cross-node data traffic, we can easily deduce 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 k was*

Algorithm 1: Key Partitioning Scheme Y

```
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
```

bound to $f(k)$ in this iteration, the shuffle size for transferring transformation output data that have key $f(k)$ after transforming D_k is zero in this iteration.

The insight from 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 reduction of scheme X compared to 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 constraints when binding key partitions. First, key partition binding can 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 constraint is that for a single copy of a dataset, any input key can be bound to at most one mapped key, because a key should have a unique partition. This constraint indicates for all key dependencies with the same input key, only one can be selected for key partition binding in an iteration. These two constraints are considered as common sense when we design new 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 reduction is linearly proportional to the dependency probability. We can further improve scheme X by applying key binding based on the key dependency with the maximum dependency probability for each key in an specific iteration (scheme Y in Algorithm 1).

A problem exists in scheme Y. The key dependency is usually provided by the programmer. In just one iteration, the programmer needs to find a distinguished key dependency set for every input key. The key dependency sets of different input keys can be different. The time complexity of finding the key dependency in an iteration is the size of a key dependency set (usually small and can be considered as a constant) times the number of keys (usually large), or roughly, O(# of keys), which is usually large.

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
```

We present the efficient Confluence Key Partitioning (CKP) scheme that 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.

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 for a specific iteration whether key partition binding should be applied or not. In some iterations, it may happen that the programmer fails to find the key dependency, or the maximum dependency probability of the key dependency set is very small, and thus applying CKP would not help much. In these cases, the programmer may 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 would the programmer figure out the key dependency set in an iteration. The key dependency is derived from the transform operation logic. In some cases, for input key-value entries with a specific key, the transform operation always generates output entries with predictable keys, regardless of what the actual values are. The matrix multiplication algorithm falls in this case. Finding key dependency in 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, the programmer needs also consider the key-value entries as a whole to deduce the mapped keys and the probability 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 assumption 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 binding iteration compared to RKP is $\Delta \approx \sum_k pr_J|D'_k|$ (recall the shuffle size reduction by key partition binding in Section 3.1). For other iterations that do not apply key partition binding, the CKP shuffle size is the same as that of RKP. All 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 partition is the number of input entries for the transform operation in that partition and the workload skewness of an iteration is the standard deviation of the workloads of different partitions in that iteration. As shuffle operations only change data partitions, but do not change key-value entry formats, the workload skewness is also the standard deviation of the shuffle input value distribution between the partitions of their mapped keys, where we denote this distribution as $V|P$.

Suppose in a distributed iteration, the shuffle input value distribution between input keys is $V|K \sim D_v$, the shuffle input key distribution between mapped keys is $K|f(K) \sim D_k$, and the mapped key distribution between partitions is $f(K)|P \sim D_{f(K)}$. Therefore, the workload skewness of an iteration is decided by the joint distribution $D_v \bowtie D_k \bowtie D_{f(K)}$ of this iteration. As D_v is decided by the nature of input data, and CKP and RKP both hashed mapped keys to their partitions, CKP and RKP differ only in distribution D_k. In RKP, the mapped key can be considered as the shuffle input key itself, and input keys are uniformly (one-to-one) distributed on mapped keys.

In a nutshell, if input keys are uniformly bound to mapped keys in each iteration, 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 equal to 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 each mapped key $userId$ is equal to the average number of items all users have bought. In a few cases (e.g., KMeans in Section 4) where this condition does not hold, we find that the workload skewness difference of CKP and RKP is so small that it could be ignored.

Note that both RKP and CKP do not try to decrease the skewness of input value distribution between keys, i.e., D_v. The skewness of D_v refers to the case that some keys have much larger number of values than others, and the balanced partitioning of keys would not erase the skewness. The decrease of skewness of values between keys can be done by other techniques [41], [42], and CKP does not try to decrease or erase this skewness, but tries not to increase the workload skewness of each iteration compared to RKP.

3.4 Inaccurate Key Dependency

Note that programmers do not need to change the original logic of applications in order to apply CKP. Applying any key partitioning 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. A dependency probability of a key dependency implies how much CKP reduces the shuffle size as compared to RKP. When an inaccurate key dependency is applied in CKP, the shuffle size reduction is related to the actual dependency probability, not to the inaccurate one. At all events, the shuffle size of CKP is no greater than that of RKP.

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	$\|A\| \cdot \sum_{i=j+1}^{m} p_{ij}$ [1] upper bound: $(m-j)\|A\|$ 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 capitalizing on the key dependency with a large dependency probability in some iterations. We also discuss limitations of CKP when in some situation it is not applicable, and solutions to handle these limitations.

We find that many applications have key dependency with a large dependency probability in some iteration(s). For example, 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), $(i, j, p) \Rightarrow_1 (i, j)$, which has 100% dependency probability for all key dependencies. In some applications, the dependency probability is not 100% but is still large enough to apply CKP. We illustrate the techniques of finding the key dependency of iterative distributed operations with real-life application examples from different fields. 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 users based on user-item rating information. During the execution, it organizes the rating data into user-item blocks

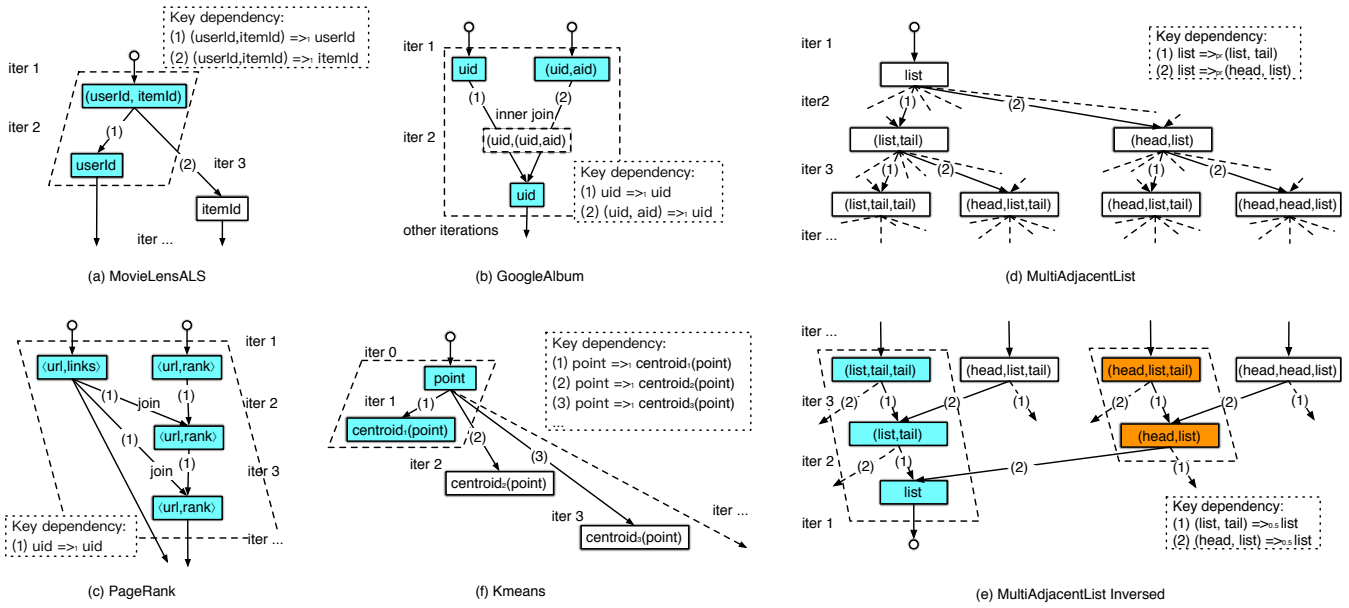


Fig. 2. The Transformation Lineage of Keys between Iterations and the Key Dependency of Different Applications. Key Partition Binding is Applied to Keys in Iterations Wrapped in Dashed Rhombus Boxes

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).

Key dependencies from input key $(userId, ItemId)$ exist in iteration 2 and iteration 3, respectively. As these two iterations use the same copy of input dataset, by the second constraint of key partition binding, we can apply key partition binding in either iteration, but not in both. If we select key dependency $(userId, itemId) \Rightarrow_1 userId$ 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 as that of RKP.

4.2 GoogleAlbum and PageRank

CKP can usually be applied to *join* operations in distributed databases. For example, in the 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 means 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 the same node before such an "inner join" operation. Spanner [26] uses a similar notion of hierarchical schema to maintain data locality in a distributed database. The hierarchical schema specifies data locations once when persisting data in storage, while CKP specifies 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 $\langle url, rank\rangle$, where url is the key for both datasets. These two datasets are joined to entries in form of $\langle url, (links, rank)\rangle$ in multiple iterations (Fig. 2(c)). 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 $join$ 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 in 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 via 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 head nodes and tail nodes to each list. Each new list acts as the new key, and its new head nodes and tail nodes as the new value. The iterative transformation of keys in MultiAdjacentList is shown in Fig. 2(d).

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 the key flow as in Fig. 2(e), 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 cutting off 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(f)). In CKP, we choose a specific iteration j, and bind key partitions of the points based on the key dependency $point \Rightarrow_1 centroid_j(point)$. 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 iterations.

We can predicate the shuffle size after applying CKP. Suppose in iteration $i, i > j$, pr_{ij} is the probability 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 the j-th iteration, partitions of points are bound to their 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 of j and make a trade-off then. In our evaluation in Section 6, we choose j to 1.

4.5 Limitations Applying Confluence

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

First, CKP is not usually applicable for single-iteration distributed operations. CKP requires that each key of a transformation input dataset has a partition location. But in single-iteration distributed operations, input datasets usually come from a distributed file system (e.g., HDFS [13]), where data locations are not organized by their key partitions. An alternative solution is to modify the mechanism of distributed file systems on choosing the partitions of data blocks when storing data to a file. If a data entry is able to declare the preferred location as its key partition, and if the distributed file system writes the entry to its preferred location, the key partitions 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 their partition location during transformation, which should be a local processing of data in each node without cross-node data traffic. This assumption is violated when performing iterative distributed operations in distributed paradigms that relocate 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 intermediate transformation data in local nodes.

The third circumstance is that programmers fail to define the key dependency or the maximum dependency probability of the key dependency set is trivially small 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. A solution to the former case 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. The latter case happens when input keys are diversely mapped to lots of mapped keys during transformation. All dependency probabilities are usually small, and the shuffle size improvement of CKP is limited in this case.

Besides, to apply CKP, all a programmer need to manually do is to provide a key dependency by the procedure in Section 3.2, and the rest is automated by the ConfluencePartitioner interface introduced later in Section 5. After the algorithm of an application is changed, programmers need to manually update the key dependency being applied accordingly. Even if the key dependency was outdated, it only affects the shuffle size reduction, as discussed in Section 3.4.

5 Implementation

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

5.1 Tying Partition and Executor in Spark

In the implementation of Spark, a partition is only a logical location where a task executes the dataset, while the logical

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.

locations of tasks are not tied to 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, where data shuffling is still needed.

To amend the mismatch of the logical partition and the actual position of the executor, we implement a new feature in Spark to allow the tying of the task partitions and the executors. The tying can be done by hashing. In the default setting of the task scheduler of Spark, when an executor becomes free, the task scheduler selects a task with whichever partition from the head of task queue. The selected task will run in that executor. Now, if the partition-executor-tying feature is turned on, when the task scheduler selects a task for the free executor, it selects the one whose task partition hash code is the same as the hash code of the free executor. The hash based tying ensures that tasks of the same partition will always be allocated to the same executor in different iterations. Finding the first task that hashes 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 a computing iteration.

A dense hashing conflict problem may exist if inappropriate hash functions are used when multiple distributed applications are running. For a concrete example, suppose there were three executors in the Spark cluster, and there were three applications, whose number of tasks is 1, 2, 3, respectively, and the task IDs are numbered incrementally from 0, 1, until 2 if possible. If we use a *mod-by-3* function as the hash function, all of the three applications will compete to tie their task 0 to the executor with hash code being 0, but only 1 applications with 3 tasks will be tied to the executor whose hash code is 2. In general *mod* hash cases, the distribution of task hash codes of multiple applications will concentrate on smaller values, but the distribution of executor hash codes tends to be normal. Therefore, tasks with small hash codes will be waiting for the competition for free executors, while executors with larger hash code may keep idle. A solution to this problem is to choose a hash function that make the distribution of task hash code of multiple applications also normal, so that the expected number of tasks assigned to each executor are the same. For example, instead of using the incremental task ID to directly *mod*, every application can use a different hash function that transform the task ID to a random value before using *mod*.

The partition-executor-tying implementation will not introduce any negative side effects into the system for the following three reasons: 1) The order to run pending tasks of the same iteration does not affect the completion time of each iteration; 2) with random hashing, each executor is expected to run the same number of tasks in each iteration; 3) the task scheduling overhead is often trivial and negligible, which is only linear in the number of executors.

5.2 Confluence Partitioner Interface

We provide an interface in Spark to allow programmers to apply CKP in a distributed application with as little as 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 programmers to provide the key mapping function of the key dependency selected for key partition binding. 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 an 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 then 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 mapped key is the same will be partitioned to the same node.

To apply CKP in a shuffle operation, programmers only need add a ConfluencePartitioner with a 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 repartitions distributed data, such as $reduceByKey$ or $join$. Fig. 3 shows the code of using ConfluencePartitioner to apply CKP in MovieLensALS and KMeans. Only a single line of code is added in MovieLensALS to apply CKP. Line 7 defines the key mapping function from key $(userId, itemId)$ to the first element of the tuple: $userId$. In KMeans, the key mapping function in Line 21 binds the partition of key a, which is a point, to its cluster centroid in a specific iteration j.

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

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

```
/**** 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,
        (x,y)=>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

can apply CKP without knowing the details of implementing the self-defined partitioner or repeatedly writing different self-defined partitioners in different iterations.

6 Evaluation

We conduct experiments on a physical testbed to compare the performance of CKP with the default RKP scheme in aspects of: the improvement on the shuffle size, the workload skewness of executors, the completion time 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 × 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. When sending shuffle data through network, Spark network streams use *Lz4*, one of the fastest (de-)compression algorithm, to serialize and deserialize shuffle data by default.

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 traffic size of the shuffle operation of all executors in that iteration. Note that the shuffle size is the compressed one after shuffle network streams serialize the raw shuffle data using *Lz4*. This metric depicts the performance of CKP in reducing the shuffle size. Besides, the standard deviation of input workloads of executors is measured as the metric of workload skewness in binding iterations. The completion time measure how CKP improves the job completion time by shuffle size reduction. The scalability of CKP is measured by the overall shuffle size of all binding iterations, 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.

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 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

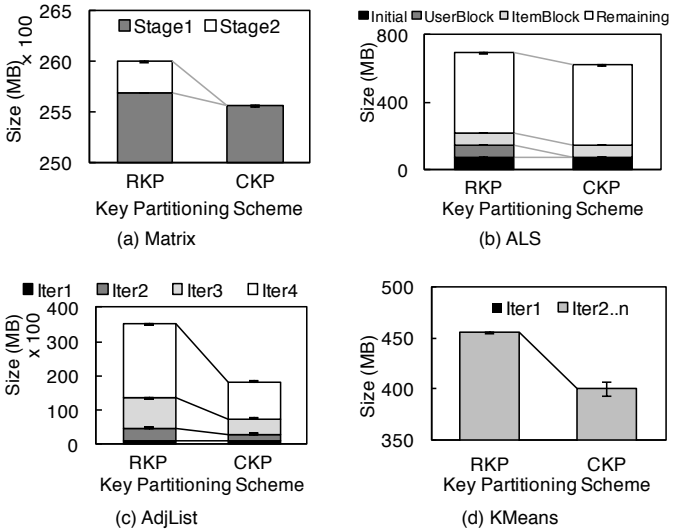


Fig. 4. The Shuffle Size and Standard Deviation of Workloads of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

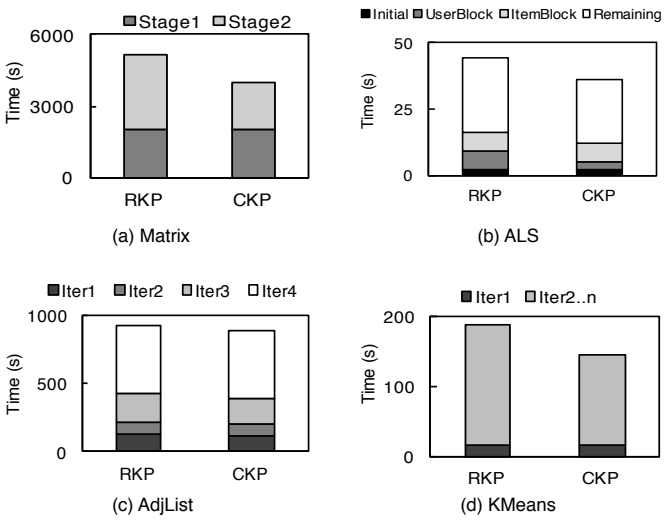


Fig. 5. Completion Time of Multiple Iterations in Different Benchmarks

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. The other iterations do not apply key partition binding, and the overall shuffle size that CKP can reduce for this application is about 11%.

After applying CKP in MultiAdjacentList, 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 centroids 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 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 reduce the shuffle size in the binding iterations without introducing extra workloads to the other iterations. How CKP can reduce the overall shuffle traffic depends on how large the volumes of data are in the binding iterations, compared to the overall data volume of all iterations. In MultiAdjacentList, the CKP can be applied to half of the iterations, by either appending nodes to the head or to the tail. While in MovieLensALS, CKP can only be applied to the iteration that arrange the user-item block, which is a relatively small portion of the overall application. Still, it indicates the potential of applying CKP in complicated iterative distributed applications.

6.4 Workload Skew

The workload skewness is indicated by the standard deviation bars 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 slightly 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 tends to be smaller. Still, the standard deviation of CKP (7 MB) is small compared to the mean shuffle size of the executors (25 MB).

6.5 Completion Time

We evaluate the completion time of each iteration to see how shuffle size reduction actually improves the performance of distributed applications. The result of different benchmarks is shown in Fig. 5. CKP reduces the overall job completion time by about 23%, 19%, 6% and 23%, respectively. In key partition binding iterations, the completion time reduction is even greater. For example, the completion time reduction of the second iteration in MatrixMultiplicaiton is about 37% (Fig. 5(a)), and that of the UserBlock iteration in MovieLensALS is about 57% (Fig. 5(b)). We conclude that the overall completion time improvement depends on how large portion the completion time of binding iterations occupy.

Another observation is that how much the overall completion time is improved depends on whether the network performance for shuffling is the bottleneck for an iterative distributed operation. For example, in MultiAdjacentList, the shuffle size of every iteration is reduced by 50%, but the average completion time reduction of all iterations is only about 6%, resulting in about 6% overall completion time

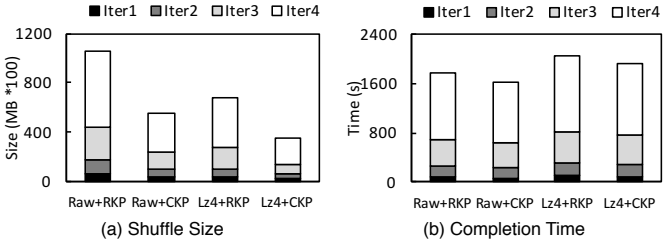


Fig. 6. Shuffle Size and Completion Time of Each Iteration in MultiAdjacentList Benchmark Using Different Compression and Key Partitioning Schemes. "Raw" Stands for Raw Data without Compression.

reduction. The reason is that this algorithm spends more time generating dozens of new lists from every shorter list, where CPU is the bottleneck most of the time and network is unlikely a bottleneck.

6.6 CKP vs. Compression

Data compression is another approach often used to reduce the shuffle size. CKP and compression are not mutually exclusive and they can be used at the same time, as what we have been doing so far by apply CKP along with the default *Lz4* shuffle streamer. Still, we are interested on how CKP and data compression, individually, affect the shuffle performance. We selectively run the MultiAdjanceList benchmark, which has many key binding iterations, and double the input size to amplify the effect. Different combinations of compression policies and key partitioning schemes are used, i.e., with or without compression, and with RKP or with CKP. Results of the shuffle size and the shuffle completion time of each iteration are shown in Fig. 6.

CKP and compression can work together to reduce shuffle sizes by a larger margin. Compression is supposed to reduce the shuffle size by a relatively stable ratio, e.g., about 37% regardless of the key partitioning scheme in this benchmark, though the theoretical *Lz4* compression ratio is about 2.2. The shuffle size reduction of CKP is about 50% of the according one of RKP, and can be very different in different benchmarks, as we have shown. The shuffle size with both CKP and compression used is only about 33% of the one with RKP and no compression used (in Fig. 6(a)).

However, compression can increase the completion time of the shuffle. In Fig. 6(b), with compression, the overall completion time is about 14% to 18% higher than that without compression, regardless of the key partitioning scheme. Even when CKP is used along with compression, the completion time is still about 7% higher than that using RKP without compression. The reason is that compression increases the time to serialize and deserialize shuffle data in network streams, and (de-)serialization overhead time can overtake the benefit of transferring less shuffle data through the network. The overhead of CKP is only the increased calculation time of the mapping function for each entry, as compared to the hash function in RKP. CKP reduces the shuffle completion time in all cases we have seen.

6.7 Scalability

The overall shuffle sizes of iterations that apply key partition binding with different input sizes in different benchmarks are shown in Fig. 7. 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, i.e., 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.

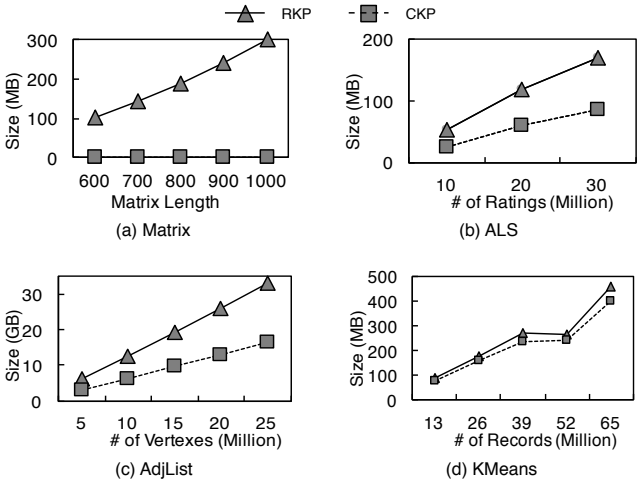


Fig. 7. Overall Shuffle Size of the Key Partition Binding Iterations in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks with Different Input Size

7 Related Work

Iterative distributed computing: Some works [17], [18], [19], [29] improved distributed paradigms for iterative distributed operations by extending their programming interfaces to support multiple transform and shuffle phases. The general idea was to cache the data of each iteration in memory instead of disks to reduce I/O overheads. However, these works focused on the paradigm and did not address the issue of heavy network workload during shuffle operations. Riding on the benefits of fast memory I/O rate in iterative distributed paradigms, CKP reduces shuffle sizes and alleviate network workloads by considering the key dependency in multiple iterations.

Parallel&Distributed algorithms optimization: Lots of work have 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 needs to redesign the logic of the program itself in order to operate with a proper parallelism factor. Their "communication cost" is different from the shuffle size in this paper, where communication cost is the transfer size of data between different reduce worker processes (which can be on the same node), while the shuffle size is traffic size 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 reduce 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 their approaches either lacked a precise model abstraction so that their target applications were limited to aggregate-like operations [28], or they have to be modeled on specific types of data exchange patterns [27]. The Confluence key dependency model is simple in form and can be applied to general iterative distributed operations. The dependency probability precisely predicts the effect of shuffle size reduction.

Datacenter networking for shuffle: In datacenters research, various networking scheduling algorithms were proposed to improve the shuffle throughput for different performance goals [15], [37], [38], [39]. CKP does not concern itself 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 are concerned with placing 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 usually does not introduce new workloads skewness. 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 for which the next iteration does not apply key partition binding, the workload skewness always readjusts to a balanced state.

8 Conclusion and Future Work

We have presented the Confluence Key Partitioning (CKP) scheme, which based on the key dependency can effectively reduce the shuffle size of iterative distributed operations. The key dependency precisely captures the logic of iterative distributed operations, and CKP then partitions data across different iterations by using the technique of key partition binding. CKP greatly reduces the overall shuffle size by a predictable percentage while not introducing any workload skewness as a side effect. CKP is applicable to a variety of distributed applications in fields such as scientific computing, machine learning, and 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 direction is data flow tracking. With a set of sample data, we can keep track of how data are flowing between nodes across different iterations, and thus find out the key dependency of the sample data. The challenge is how to expand the key dependency of sample data to the whole dataset. Unrepresentative sample data affect the accuracy of the dependency probability. Luckily, as we have discussed, CKP survives the problem of inaccurate dependency probability.

Acknowledgments

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

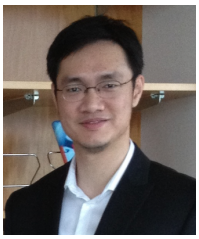
References

[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.

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