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 key-value pairs on many computers, and thus generates significant network traffic and often dominates completion time. This traffic is pronounced in iterative distributed applications where each iteration invokes a shuffle operation. Our key observation is that some iterations process different types of key-value pairs according to relevant keys among these types, and we can partition key-value pairs on local machines by exploiting this relevance, greatly reducing shuffle traffic for the following iterations. For instance, PageRank involves two types of key-value pairs: the URL key and webpage links pair, and the URL key and webpage rank pair. These two pairs' URL keys are relevant because if we partition them with the same URL on the same node, PageRank's *join* operations in the next iteration incur no shuffle network traffic.
We present a new data structure called Key Dependency Graph (KDG) to capture relevant keys for general iterative distributed applications. We implement the Confluence system, where developers construct a KDG for a distributed application, and Confluence automatically generates an efficient key partitioning scheme from the KDG. We apply Confluence on diverse applications and show that Confluence greatly decreases the shuffle network traffic by up to 50%.

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

1 Introduction

Distributed applications consisting of iterative distributed operations are pervasive in the fields of graph computing [1], [2], database query processing [3], [4], [5] and machine learning [6], [7]. To process the big data, distributed computing frameworks like YARN [8], Spark [16] and Dryad [9] are often used. Several distributed computing paradigms have been developed on top of these frameworks to match various styles and scenarios of computing on large-scale data [3], [4]. MapReduce [10], one of the most popular distributed paradigms, provides users a simple map-and-reduce interface that can suit most of these data analysis tasks. Generally, it saves the output data in the file system, such as HDFS [11]. This is inefficient for tasks of the iterative distributed operations, where one iteration reuses the data from the previous iterations. The sharing of data between iterations is usually done through a shuffle operation.

The problem of heavy shuffle network traffic greatly impacts the performance of distributed operations. Most distributed computing paradigms include the shuffle operation, which transfers intermediate output data of map operations to the computer nodes doing reduce operations. 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. The study based on the Yahoo! work trace has revealed as much as 70% of jobs are shuffle-heavy [12] and the shuffle completion time can account for as much as 33% of overall completion time [13], [14]. 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 computing paradigms (e.g., Spark [16]) reuses intermediate data across iterations in memory. However, the volume of intermediate data and their network traffic is still enormous.

Our key observation is that we can exploit the dependency of keys across iterations and partition the data according to the relation between keys of different iterations to greatly decrease the network traffic. How the output data of the previous shuffle iterations are partitioned across different locations can greatly affect the data that are to be shuffled and its efficiency in the following computing iterations. Keys are relevant between different iterations. If we have a key partitioning scheme such that some data needed for the following computing iterations can be assigned to the same computing node after shuffling, we can reduce the data volume of shuffle operations (we call that the "shuffle size" in the rest of this paper). With the default hashed-by-key partitioning scheme, 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 partition-

• *F. Liang, F.C.M. Lau, H. Cui and C.-L. Wang are with Department of Computer Science, The University of Hong Kong, Hong Kong SAR. 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*

Manuscript received January 7, 2017.

ing 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 $A_{ip}B_{pj}$, $p = 1, 2, ..., k$. 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). The boxes having the same color represent the related data entries needed to generate a particular result entry C_{ij}. 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 $(1, 1) \rightarrow 19$, the entry $(1, 1) \rightarrow 14$ in Node 2 needs to be transferred to Node 1 to join the entry $(1, 1) \rightarrow 5$. However, if in the first shuffle iteration, knowing that entries with keys (i, j, p) are expected to all mapped to the key (i, j) in the second iteration (Fig. 1(c)), a better partitioning can be done such that no entry is needed to be transferred 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 from the input entries in the second iteration with no data transfer.

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

We present a new data structure called the *key dependency graph* (KDG) for this purpose. The KDG is a directed acyclic graph which depicts how the keys of dataset entries of each iteration are generated from the previous iteration. From the KDG, we can identify the subgraphs in which dataset entries with the keys in the source nodes will only generate the dataset entries with keys in the downstream nodes. We define such a subgraph the *pure confluence subgraph*. Within a pure confluence subgraph, if all the dataset entries matching the source keys are assigned to the same node, all the generated datasets can be computed locally in the node without shuffling.

To make use of the confluence key partitioning scheme (CKP) in iterative distributed operations, Confluence allows programmers to construct a KDG for a distributed operation. KDG is easy to construct and it does not have to be accurate (Section 3.5). Confluence automatically generates an efficient key partitioning scheme from the KDG. This user-defined key partitioning scheme guides the assignment of data entries to their designated nodes. Most distributed computing paradigms, such as Spark [16] and Twister [17], provide an interface for adding a user-defined partitioner for the shuffle operation. By applying the CKP scheme, shuffle sizes of multiple computing iterations can be reduced significantly. Note that this will not impact the computing workload skew level; that is, the standard deviation of the computing workloads of the nodes in the cluster after applying CKP will not be larger than that with the random hashing scheme. We are not aware of any previous work that also tried to decrease the overall shuffle traffic by a key partitioning scheme across multiple iterations based on the key dependency relationship.

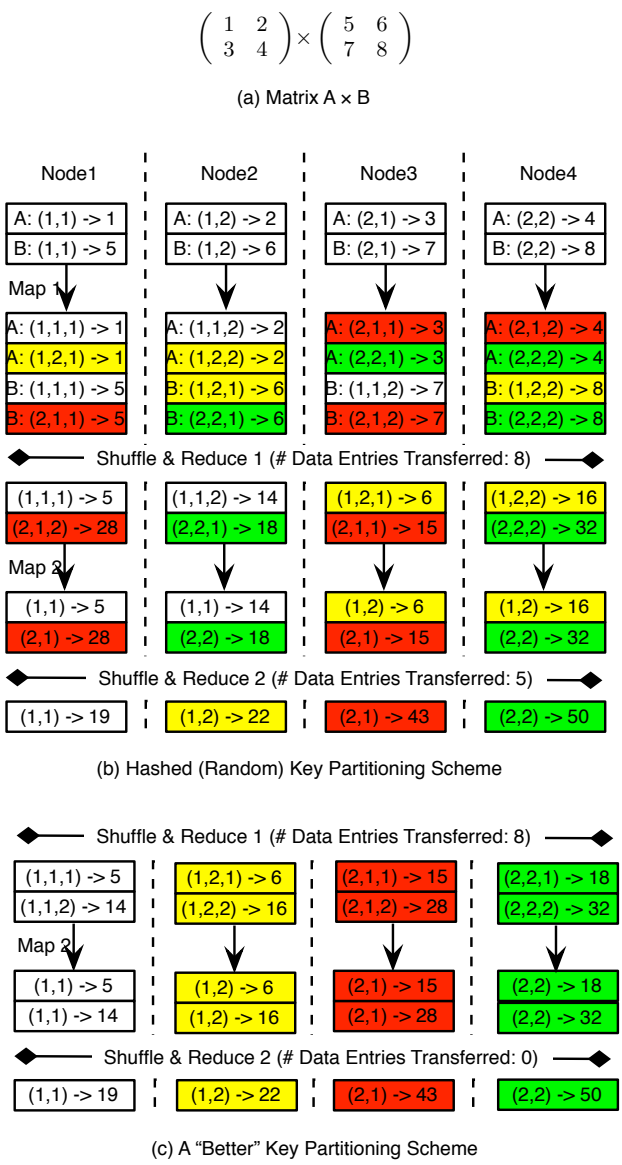


Fig. 1. An Example of the MapReduce-style Matrix Multiplication Algorithm with Different Key Partitioning Schemes

Our major contributions are listed as follows.

- We invented the new data structure Key Dependency Graph to depict the iterative key dependency and used the Confluence Key Partitioning (CKP) scheme to decrease the shuffle traffic across distributed computing iterations.
- We created the method to analyze the data transfer size and the workload skewness when using CKP, which offers the solid shuffle traffic improvement with guarantees.
- We implemented CKP in Spark and illustrated the use of CKP with typical iterative distributed applications from different fields. The experiment on a medium-size cluster demonstrates that CKP reduces the overall shuffle traffic by as much as 50%.

The rest of the paper is organized as follows. Section 2

gives some background information on iterative distributed operations , dependency graph, and the model of the iterative shuffle size. In Section **??**, we show how to construct the KDG and to reduce effectively the shuffle sizes by the Confluence Key Partitioning scheme, as well as analyzing the workload skew level. Section 4 introduces how to apply CKP in the real-life iterative distributed applications and Section 5 discusses the implementation details of CKP in Hadoop and Spark. The evaluation results of the performance 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 Background

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 computing 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 [10], 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 [16] and the distributed database query engine Hive [3] 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.

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

$$A'_i = T_i(A_i) \tag{1}$$

$$A_{i+1} = S_i(A'_i) \tag{2}$$

, 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:

$$G_i = \sum_{a \in A_i} g_i(a, A_{i+1}). \tag{3}$$

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

$$G = \sum_{i=1}^{m} G_i = \sum_{i=1}^{m} \sum_{a \in A_i} g_i(a, A_{i+1}). \tag{4}$$

If values of $G_i (1 \leq i \leq m)$ are independent of each other, S is a linear function and S can be minimized simply by minimizing each $g_i(a, A_{i+1})$. Not correct now because they are not independent However, most usually, G_i and G_{i+1} are correlated. The value of G_i depends on the logic of the transform function T_i in Formula (1), and the partition locality of datasets A_i and A_{i+1}. The partition locality of A_{i+1} will again affect the value of G_{i+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.

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

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 Key Dependency

We use *key dependency* to represent the 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

$$k \Rightarrow_{pr} f(k) \tag{5}$$

, 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.

A programmer is supposed to be well acquainted with the logic of the transform operation t. In the key dependency, we call k the input key, $f(k)$ the mapping key, and pr the dependency probability. 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 [16] 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:

$$\sum_{a \in D_k} g_i(a, A_{i+1}) = \sum_j \frac{n-1}{n} pr_j |D'_k| = \frac{n-1}{n}|D'_k| \approx |D'_k|. \tag{6}$$

In cases that n is large in a large cluster, the approximately equal sign in Formula 6 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 4, 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 some term in the key dependency, we have the intuition that we can leverage the key dependency to discover other key partitioning schemes that can reduce the shuffle size.

In this section, we present the Confluence Key Partitioning scheme that leverages 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 skew level.

3.1 Binding Partition Locations

Binding partition locations of mapping keys to input keys based on the key dependency reduces the shuffle size. In an arbitrary iteration, again assume that we have a set of key dependencies $k \Rightarrow_{pr_j} k_j, \sum_j pr_j = 1$. Now suppose that a key partitioning scheme X always partitions shuffle output data that have key k_x to the same computer node where input data that have key k is partitioned. We say that

Algorithm 1: Confluence Key Partitioning

Input : A Key Dependency Set DK:
$\{k \Rightarrow_{pr_j} f_j(k) | j = 1, 2, ..., \sum_j pr_j \leq 1\}$;
find out J such that $\forall j, pr_j \leq pr_J$;
bind key $f_J(k)$ to key k

key partitioning scheme X binds the partition location of the mapping key k_x to the input key k in this iteration. For shuffle outputs that have other keys, scheme X partitions them by the hash values of their keys.

By the definition of the key dependency, we easily deduct the following theorem.

Theorem 1. In an iteration of a distributed operation, given a key dependency $k \Rightarrow_{pr} f(k)$, if the partition location of key $f(k)$ is bound to k, the shuffle size of transferring transformation output data that have key $f(k)$ after transforming D_k is zero.

In such a scheme X, reusing the symbols in Section 2.4, 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.

3.2 Confluence Key Partitioning

The Confluence Key Partitioning (CKP) scheme

3.3 Confluence Key Partitioning

We demonstrate that the overall data transfer size can be decreased by localizing the pure confluence subgraph, i.e., placing the datasets corresponding to the pure confluence subgraph in a node.

Assume that there is a pure confluence subgraph C in the KDG, from level u to level $w (u \neq 0, w > u)$. Let V'_u and V'_w be the source key set and the confluence key set of C, respectively, and B_u and B_w be the upward closure pair datasets. The *Confluence Key Partitioning (CKP)* scheme partitions the dataset B_u in one single node in Iteration u and completes the following $(w-u)$ iterations of operations on B_u and its generated dataset in the same node.

CKP localizes the data in the granularity of a pure confluence subgraph, different pure confluence subgraphs are partitioned randomly.

Although the KDG is the pure confluence subgraph of itself, we do not consider the localizing the whole KDG in the above discussion. The reason is that the raw input dataset is always distributed across the cluster before the first iteration.

We analyze the expected overall data transfer size S' of CKP in comparison with that S'' of the random key partitioning scheme (RKP), which is ignorant of the key dependency and partition the data entries randomly (e.g., partition by the hashed value of keys). S'_i and S''_i denote the expected data transfer size of CKP and and RKP in the i_{th} iteration, respectively.

Before going further, we figure out the function $s_i(a, A'_i)$ in Formula 3 in the case that a is randomly (uniformly) partitioned to a node. In this case, the probability of a is partitioned to any node is $1/n$, where n is the number of nodes in the cluster. We call such an entry that has the same probability being partitioned to any node as the uniformly-partitioned entry. As long as the partition location of an entry is finally hashed to the computer nodes, the entry is a uniformly-partitioned entry.

Suppose that a' is the set of input data entries that have the same key as a right before the shuffle operation and p_j denotes the percentage of data entries of a' that lie in Node j, where $\sum_{j=1}^{n} p_i = 1$. We have

$$s_i(a, A'_i) = \sum_{j=1}^{n} [\frac{1}{n}|a|(1 - p_j)] = \frac{(n-1)}{n} \cdot |a| \qquad (7)$$

, where $|a|$ is the length of the value list of a, which is also the number of data entries in a'. It shows that if a is randomly partitioned to a node, the distribution of a' does not affect the data transfer size for collecting a.

By Theorem **??**, when using RKP, the data transfer size of the i_{th} iteration is $S''_i = \sum_{a \in A_i} \frac{(n-1)\cdot|a|}{n} = \frac{n-1}{n}|A'_i|$, where $|A'_i|$ is the number of data entries in A'_i.

For CKP, we analyze every iteration as follows.

1) From Iteration 1 to $(u-1)$, the key partitioning scheme is the same as RKP: $S'_i = S''_i, 1 \leq i \leq u-1$.
2) In Iteration u, an entry of B_u belongs to a pure confluence subgraph and the pure confluence subgraph is randomly partitioned. The entries in B_u are uniformly-partitioned entries, as well as the entries in $A_u - B_u$. Therefore, $S'_u = S''_u$.
3) From the $(u+1)_{th}$ to w_{th} iteration, the data transfer size of the operations on the datasets generated by B_u is 0, by Theorem **??**. We have $S'_i = S''_i - \frac{n-1}{n}|B'_i|$, where $r_i(B'_i) = B_i, u+1 \leq i \leq w$.
4) From iteration $(w+1)$ onward, the data entries are uniformly-partitioned entries again: $S'_i = S''_i, i \geq w+1$.

In all, the expected overall data transfer size after applying CKP is $S' = S'' - \frac{n-1}{n}\sum_{i=u+1}^{w}|B'_i|$, which eliminates exactly the shuffle workload of the datasets that associate with the keys in the pure confluence subgraphs as compared to RKP, without introducing other shuffle workloads. $\sum_{i=u+1}^{w}|B'_i|$ stands for the shuffle size improvement of CKP as compared to RKP. This result indicates that CKP can reduce more data transfer size if the pure confluence subgraph contains more iterations or the size of the dataset corresponding to the pure confluence subgraph is larger. In Section 4, the value of $\sum_{i=u+1}^{w}|B'_i|$ will be figured out for the specific distributed operations.

3.4 Workload Skew Analysis

The workload of a node is the number of data entries of the map primitive in that node and The workload skew level of Iteration i is the standard deviation of the expected workloads of different nodes in that iteration, which is denoted as L_i. The math symbols in Section 3.3 are reused here without duplicated definition. Only the data entries that associate

with the pure confluence subgraphs are counted as the other data (if any) are randomly partitioned in both RKP and CKP and will not affect the standard deviation of the workloads. We only consider the workload skew levels of the iterations that the pure confluence subgraphs affects (Iteration $u+1$ to $w+1$). The reason is that both RKP and CKP use the same key partitioning policy in the CKP-not-affected iterations and the workload skew levels of these iterations will be the same in both schemes.

In all iterations when RKP is applied, every entry is a uniformly-partitioned entry, the expected workloads of the all the nodes are the same and the expected workload skew level is 0.

When CKP is applied, as each output entry of the shuffle operation is the uniformly-partitioned entry in Iteration u, the workloads of all the nodes are the same in Iteration $u+1$. Therefore, $L_{u+1} = 0$.

In Iterations from $u+2$ to $w+1$, it can be divided into two phases based on the logic of the distributed operations. Suppose that Iteration $u'(u+1 \leq u' \leq w)$ is such an iteration that in every iteration from $u+1$ to $u'-1$, each input entry of the map primitive is expected to generate the same number of output entries of the shuffle operation, and in Iteration u', the expected number of output entries of the shuffle operation generated by each map input entry is unknown or not the same. Note that such Iteration u' may not exist and in this case, we denote it as $u' = w+1$.

Phase 1: From Iteration $u+2$ to u', the workloads are the same in all the nodes and the workload skew level of every iteration is 0.

Phase 2: From Iteration $u'+1$ to $w+1$, the workloads of the nodes in each iteration are unpredictable, which are dependent on the number of data entries that can be generated by the data localized in each node. Therefore, $L_i \geq 0, u'+1 \leq i \leq w+1$.

Although the workload skew levels of CKP in Phase 2 is unpredictable, fortunately, such Iteration u' does not exist in many distributed operation, including matrix multiplication and the other distributed applications (MovieLensALS, MultiAdjacentList and KMeans) introduced in Section 4. In these applications, the expected workload skew level of CkP is 0, which is the same as that of RKP. In the distributed application where u' does exist, the workload skew level may not be 0 and specific workload skew analysis is needed for this specific application.

3.5 Affect of Inaccurate KDG

Note that the programmers do not need to change the original logic of programs in order to apply CKP. A wrong or inaccurate KDG will not impact the accuracy of the computation results, but only affect results of key partitioning locations. Unpredictable workload skewness may exist, but as CKP randomly partitions the pure confluence subgraphs, the workloads tend to be balanced in all the nodes.

4 Application

In this section, we show how CKP can be applied in various iterative distributed applications, even when the applications are not well-structured and the pure confluence subgraphs are not obvious, by dividing the operations into multiple sub-operations.

TABLE 2
CKP Shuffle Size Improvements in Different Distributed Operations

Distributed Operations	CKP Shuffle Size Improvement
MatrixMultiplication	$n^3 for R^{n \times n}$
MovieLensALS	# of $(user, item)$ entries
MultiAdjacentList	1/2 the RKP shuffle size in every iteration
KMeans	$\|A\| \cdot \sum_{i=2}^{m}(\sum_{j=1}^{k} p'_{ij} - 1/n)$ upper bound:$\|A\| \cdot (m-1)(n-1)/n$ lower bound:$-\|A\| \cdot (m-1)/n$

In the key dependency graph of the matrix multiplication algorithm, the pure confluence subgraphs are obvious in the second iteration, whose keys follow the many-to-one mapping pattern. In some cases, CKP may not be applicable when the pure confluence subgraphs cannot be found in the straightforward way. In this case, if the iterative distributed operations can be divided into several smaller operations to fit the Confluence model, CKP can be flexibly applied to some of the divided operations. The division of the iterative distributed operations can be based on either the operations or the datasets. In the following, we illustrate the methods to divide the iterative distributed operations with the real-life application examples, namely MovieLensALS, MultiAdjacentList and KMeans.

4.1 MovieLensALS

MovieLensALS [20] is a recommendation application that uses Alternating Least Squares (ALS) method in collaborative filtering in order to recommend the product items to the users based on the user-item rating information. During the execution, it organizes the rating data into the user-item blocks grouped by the keys "$(userId, itemId)$". In the successive iterations, it needs to reorganize the user-item block data by generating the user block and the item block grouped by "$userId$" and "$itemId$", respectively. This operation is done in two iterations as shown in Fig. 2(a). No single key partitioning scheme can remove the shuffle data in order to generate both of these blocks from the user-item block, as the key dependency follows a many-to-many pattern. However, if the programmer regards the iterative distributed operation as two individual iterative operations and the operation is divided as in Fig. 2(b), the pure confluence subgraph can be found in the operation in the dashed box. CKP can then be applied on the key dependency $(userId, itemId) \Rightarrow userId$ for the user block.

As CKP is applied only in one successive iteration in MovieLensALS, the workload skew level is 0.

4.2 MultiAdjacentList

MultiAdjacentList [21] is a graph computing application which generates the heads and tails of lists of various lengths from the information of edges. In every iteration, where the key L_i represents a list of length i and its values are the sets of the head nodes and tail nodes of the list, it 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

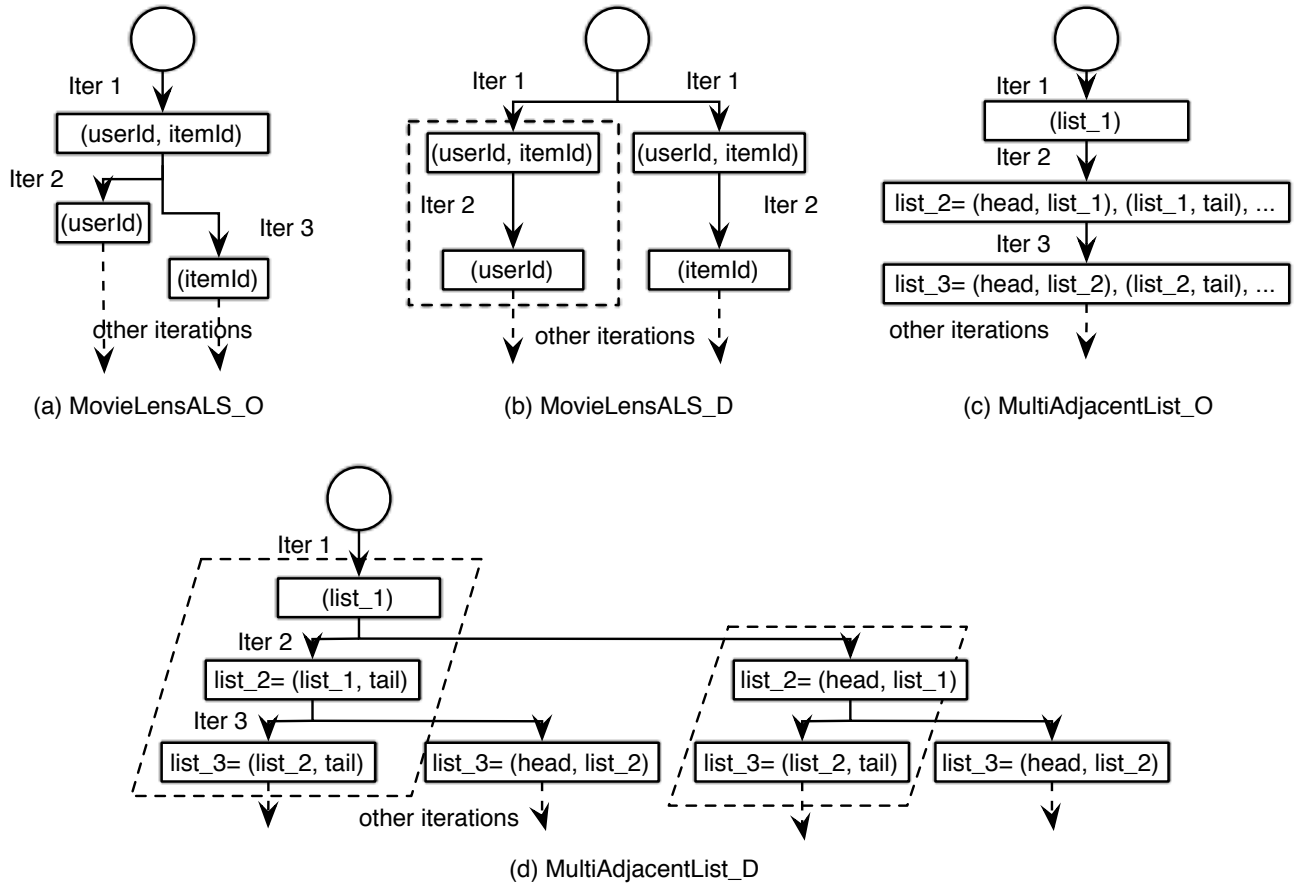


Fig. 2. The Original (Suffixed by _O) and Divided (Suffixed by _D) Iterative Evolution of the Datasets of the MovieLensALS and MultiAdjacentList Applications. The Keys of the Datasets are Marked by Parenthesis. Pure Confluence Subgraphs can be Found in the Computing Iterations in the Dashed Boxes

act as the value. The evolution of the data of multiple iterations is shown as Fig. 2(c). The key dependencies are $L_i \Rightarrow (heads, L_i)$ and $L_i \Rightarrow (L_i, tails)$. We cannot use CKP to localize the keys $(heads, L_i)$ and $(L_i, tails)$ with L_i, as the datasets corresponding to Key $(heads, L_i)$ and Key $(L'_i, tails)$ may overlap for different L_i and L'_i. However, the programmer can divide the iterative operations into several smaller iterative operations by dividing the dataset as in Fig. 2(d). In the iterative operations in the dashed boxes, the pure confluence subgraphs exist in their KDG's by the key dependency $L_i \Rightarrow (L_i, tails)$. Applying CKP to these operations and leaving the other iterative operations that are related to the keys $(heads, L_i)$ to be randomly partitioned, only half of the datasets need to be shuffled.

In the setting that every node has l heads and l tails on average, every map input entry is expected to generate $2 \cdot l$ shuffle output entries in each iteration. Therefore, by the discussion in Section 3.4, the workload skew level of each iteration is 0 when CKP is applied.

4.3 KMeans

KMeans [22] is the popular clustering algorithm in data mining, which groups the vector points into k clusters where each vector point belongs to the cluster with the nearest mean. This process could be done in a dedicated number of iterations. With the initial k chosen points regarded as the centers of the clusters, in each iteration, every point is compared with the k centers and is is grouped into the cluster where the Euclidean distance between the point and the center is the smallest. After all points are grouped, each new cluster center for the next iteration is chosen by the mean of the points in that cluster. The shuffle operation takes place when calculating the mean of each cluster whose points are located in different nodes.

If the points are bound to the same cluster in every iteration, by using the key dependency $points \Rightarrow centers$ and localizing the points in the same cluster at the first iteration, CKP can eliminate the shuffle size in the successive iterations. However, the affiliation of point to clusters are not stable between iterations. One point that is grouped into one cluster in a iteration can be grouped into another cluster in the next iteration. We estimate the shuffle size in this case.

Suppose that p'_{ij} is the percentage of points that are grouped in Cluster j both in the first CKP iteration and Iteration $i (i \neq 1)$, $|A|$ is the total number of points, m is the total number of iterations, and n is number of nodes. The shuffle size of Iteration i is $S_i = |A| \cdot (1 - \sum_{j=1}^{k} p'_{ij})$. The CKP shuffle size improvement in Iteration i is $(n-1)/n \cdot |A| - S_i = |A| \cdot (\sum_{j=1}^{k} p'_{ij} - 1/n)$, and the overall CKP shuffle size improvement is $|A| \cdot \sum_{i=2}^{m} (\sum_{j=1}^{k} p'_{ij} - 1/n)$. The higher

$\sum_{i=2}^{m}\sum_{j=1}^{k} p'_{ij}$ is, the higher the shuffle size improvement. In the best case that points are bound to the same cluster in every iteration, $\sum_{j=1}^{k} p'_{ij} = 1(i = 2, 3, ...m)$ and the shuffle size improvement is $|A| \cdot (m-1)(n-1)/n$. In the worst case, every point grouped into cluster i in the first iteration is grouped to a cluster other than i, the shuffle size improvement is $-|A| \cdot (m-1)/n$. The affiliation of the points to the clusters are more stable after running a few iterations. Instead of localizing the points of the same cluster in the very first iteration of KMeans, localizing them after a few beginning iterations can give higher $\sum_{j=1}^{k} p'_{ij}$.

In every iteration of KMeans, as every input point generates the same point as the input for the next iteration, the workload skew level of every iteration is 0.

A summary of the CKP shuffle size improvements of the distributed operations discussed above is shown in Table 2. All the values of the CKP shuffle size improvement are figured out without consideration to the map-side combine mechanism discussed in Section 5.

5 Implementation

This section introduces the effect of the map-side combine mechanism and the limitations of applying CKP in MapReduce. The implementation details of binding partitions and executors to facilitate CKP in Spark and the discussion on how to automate the confluence approach will also be presented.

5.1 Map-Side Combine

In practice, existing distributed frameworks can do map-side combine for the map output data before the shuffle operation if the reduce operation is an aggregate function, e.g, the sum function of the reduce operation in Stage 2 of the matrix multiplication algorithm as mentioned above. The map-side combine does the local reduce operation on map output data such that the data needed to transfer in the shuffle operation can be greatly decreased in size even when RKP is applied. However, CKP is still beneficial because it eliminates the overhead time of initializing and cleaning up the shuffle connections, which can be time-consuming.

5.2 Confluence in Hadoop

CKP can be widely applied to decrease the shuffle size of the iterative distributed paradigms that compute and store intermediate datasets in the memory, like Spark and Twister. However, we find it not applicable to Hadoop MapReduce for the following reason. Hadoop MapReduce stores the output result of each distributed operation iteration in the distributed file system, i.e., HDFS, where the partitioned locations of the data blocks are reorganized due to the internal storing mechanism of HDFS. Even if the datasets are partitioned by CKP during the distributed operation, the partitioned locations of the datasets will not be retained in the next iteration of the distributed operation.

An alternative solution is to modify the mechanism of HDFS on choosing the partitioned locations of data blocks when storing data to a file. If the dataset is able to declare the preferred partitioned location as the node where it is kept in the memory and HDFS writes the dataset to its preferred partitioned locations, the partitioned locations of the datasets will remain unchanged and CKP is applicable. We have not implemented this new mechanism in HDFS and the feasibility is yet to be verified.

5.3 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 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.

5.4 Automating Confluence

Currently, programmers need to manually discover the pure confluence subgraphs of the iterative distributed applications from KDG and specify the key partitioning scheme in the program code if they apply CKP. A system which automates this process before the runtime of the application and frees the programmers the extra work may sound attractive. However, there are some difficulties implementing such a system because it is hard for the machines to really understand the application logic to draw the KDG. The reasons are as follows. 1) Difficulty in constructing the vertexes: without the provision of the real meanings of the keys by the programmer, the domain of the values of keys or the patterns of the keys are unknown without visiting the values of the whole dataset. 2) Difficulty in constructing the edges: the key dependency relationship is unclear by merely looking at logic of the program code because the key dependency relationship can also depend on the values, which are unknown until runtime.

TABLE 3
User-Defined Functions of ConfluencePartitioner in Different Distributed Operations

Distributed Operations	Key Dependency	User-Defined Function
MatrixMultiplication	$(i, j, p) \Rightarrow_1 (i, j)$	(a:Any) => a match {case (_ , _, _) => (a._1, a._2)}
MovieLensALS	$(user, item) \Rightarrow_1 user$	(a:Any) => a match {case (_ , _) => a._1}
MultiAdjacentList	$list \Rightarrow_{0.5} (list, tails)$	(a:Any) => a match {case s:String => {s.split(" ")(0) }}
KMeans	$point \Rightarrow_{pr} centroid_1(point)$	(a:Any) => a match {case point:Vector[Double] => closestPoint(point, 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(point, centroids$_1$) is the function which returns the index of the point in "centroids$_1$" that is in the shortest Euclidean distance with "point". The "centroids$_1$" are the cluster centroids of the first iteration.

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

To apply CKP, the powerful partitioner interface offered by the pervasive distributed frameworks allows the programmers to easily implement the specific partitioner for their distributed application. We make one step further by implementing the half-automatic approach so that if the programmers provide the information of the pure confluence subgraphs via a simple interface, the system will generate the corresponding partitioner and apply CKP automatically.

We implement the class named ConfluencePartitioner in Spark, which only requires the programmer to input the user-defined function that defines the mapping of the keys to the pure confluence subgraphs. The shuffle operation uses the ConfluencePartitioner to identify the pure confluence subgraphs that the data entries belong to and partitions each pure confluence subgraph randomly by the hashed value of the subgraph. To apply CKP with ConfluencePartitioner can be as simple as a single line of code in the Scala language:

data.shuffleOperation (new ConfluencePartitioner (numPartitions, user-defined-function))

Table 3 shows the user-defined functions in the Scala codes of different distributed operations mentioned above.

To calculate the partition of a key-value entry in the shuffle operation, the user-defined function is applied to the key of the entry, and the returned result is then hashed by number of total partition nodes. The hashed value decides the partition location of this entry. In this way, the key-value entries whose returned values of the user-defined function are the same are considered to match the same confluence key node and will be partitioned to the same node.

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

6 Evaluation

We conduct experiments on a physical testbed to compare the performance of CKP with the default RKP scheme in various aspects: the improvement on the shuffle sizes of different iterations, the shuffle skew level of the executors, and the scalability via the metrics of the overall shuffle sizes against different input data volumes.

6.1 Testbed and Benchmarks

The testbed consists of 18 computer nodes of the Gideon-II cluster in HKU [23], where each node is equipped with 2 quad-core, 32 GB DDR3 memory and 2 $\times$ 300 GB SAS hard disks running RAID-1. The nodes run with Scientific Linux 5.3 and are connected to an internal non-blocking switch with GbE ports. 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, while the remaining 16 nodes are configured as both the HDFS data nodes and YARN node managers. Spark is deployed on top of the YARN cluster with 16 executor, where each executor runs with 8 GB of 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. Pure confluence subgraphs can be found based on the key dependency relationship indicated in the table, and CKP is applied on these pure confluence subgraphs. CKP is compared with RKP as the baseline as RKP is the default and generally used key partitioning scheme for the distributed operations. Other key partitioning scheme are sometimes used to balance the workload of the executors for specific datasets. These key partitioning schemes rely not only on the application logic but also on the distribution of datasets, 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 benchmarks. The shuffle size of each iteration is the sum of the data transfer size of the shuffle operation of all the executors in that iteration. This metric depicts the performance of CKP in decreasing the data transfer size. Besides, the standard deviation of the shuffle sizes of the executors is measured as the metric to evaluate the workload skew level 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 the iterations that are in the pure confluence subgraphs with different volumes of input data.

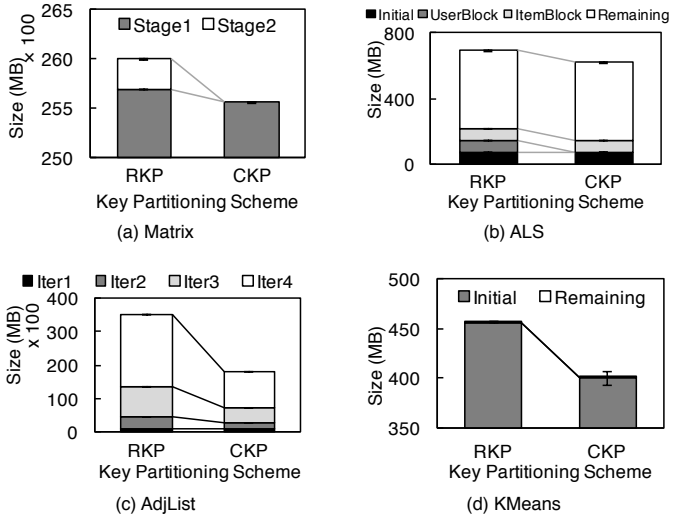


Fig. 3. Shuffle Sizes of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

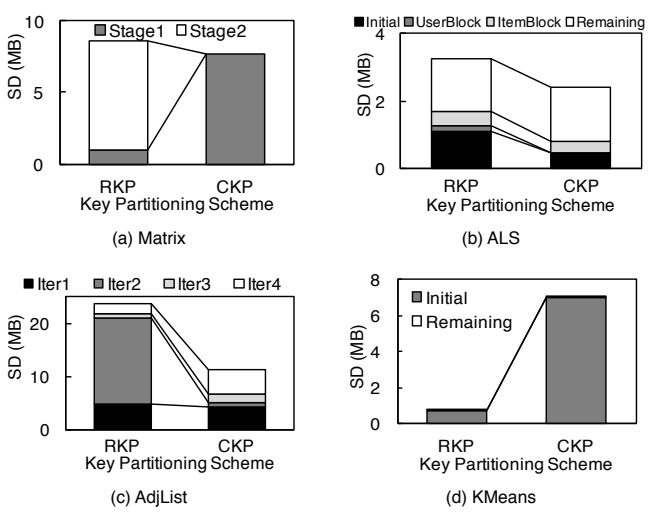


Fig. 4. Standard Deviations (SD) of the Shuffle Sizes of the Executors of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

6.3 Shuffle Sizes Improvement

The results of the shuffle size of each iteration (or stage) after applying CKP and RKP in all the benchmarks are shown in Fig. 3, respectively.

In MatrixMultiplication, CKP totally removes the shuffle operation of Stage 2 whose shuffle size is 0, and the shuffle size of Stage 1 remains closed to that of RKP (Fig. 3(a)). Note that when RKP is applied, the shuffle size of Stage 2 is very small as compared to Stage 1 due to the map-side combine mechanism mentioned above.

The benefit of applying CKP on the smaller iterative distributed operation divided based on operations can be read in Fig. 3(b). In MovieLensALS, after applying CKP, the shuffle size for generating the user block is 0, while that 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 traffic CKP can decrease for the 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. 3(c)). The reason is that by using CKP on divided sub-operations that handle the datasets corresponding to the key $(list, tails)$, only the datasets of Key $(heads, L_i)$ need to be shuffled.

In the KMeans benchmark, by partitioning the points into the clusters they belong to at the first few iterations, CKP decreases the overall cluster shuffle size by 12%. 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.

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 portion of data corresponding to the pure confluence subgraph take in the overall processing 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 only a small portion of the overall application. Still, it indicates the potential of applying CKP in the complicated iterative distributed applications.

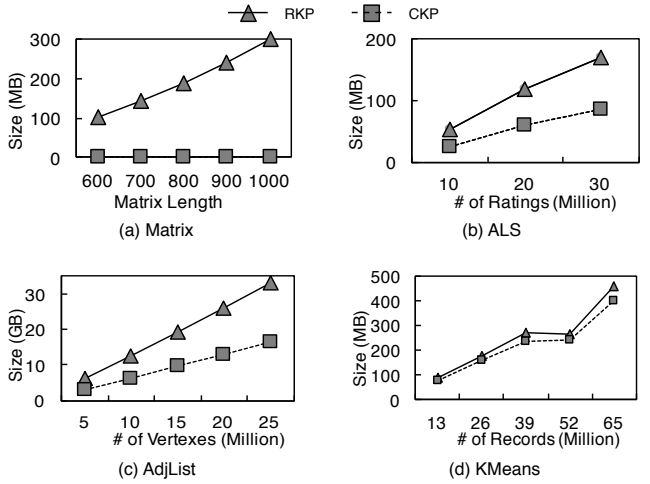


Fig. 5. Overall Shuffle Size of the Pure-Confluence-Subgraphs-Related Iterations in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks with Different Input Size

6.4 Workload Skew

The standard deviations of the shuffle workloads of the executors of different benchmarks are shown in Fig. 4. As expected, the workload standard deviations of the CKP-affected stages of MatrixMultiplication (Fig. 4(a)), MovieLensALS (Fig. 4(b)) and MultiAdjacentList (Fig. 4(c)) are all 0. In the KMeans benchmark, the standard deviation of CKP is larger than that of RKP (Fig. 4(d)). The reason is

that the number of points belonging to each cluster varies but the number of cluster centers (32) is not large enough as compared to the computer nodes (16) to even the workloads of the nodes. If the number of cluster centers is large enough, the actual number of points partitioned to each node will get closer to the expected value, which is the same in every node. Still, the standard deviation of CKP (7 MB) is small as compared to the mean shuffle size of the executors (25 MB).

For the stages that are either before or after the CKP-affected stages, the standard deviations remain almost the same as RKP, or the change is so tiny as compared to the overall shuffle size of the stage that can be ignored.

6.5 Scalability

The overall shuffle sizes of the pure-confluence-subgraph-related iterations 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 workload for Stage 2 of 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 workload greatly (about 50%) in one iteration out of the many distributed operations in the MovieLensALS application, and a little (about 12%) but in every iteration of the distributed operations of the KMeans application.

7 Related Work

Iterative distributed computing: Some works [15], [17], [24] improved distributed frameworks for iterative MapReduce-like computing by extending programming interfaces to support multiple map and reduce phases. The general idea was to cache the data of each iteration in memories instead of in disks to reduce I/O overheads. Spark [16] provided more kinds of operation interfaces and supported a series of transformations on distributed input datasets in the memory by the concept of resilient distributed dataset. However, these works on distributed frameworks did not consider the heavy network workload during shuffle operations. Inheriting the benefit of fast memory I/O rate, Confluence decreases shuffle sizes and alleviate network workloads by considering the key dependency of multiple related iterations.

Parallel&Distributed algorithms optimization: Lots of work has been conducted on the optimization of parallel&distributed machine learning algorithms [25] and science computing [26], [27], including matrix multiplication methods [28], [29]. Generally, they focused on optimizing the algorithm itself by decomposing tasks to increase the task parallelism and removing the synchronization boundary between I/O-intensive tasks and compute-intensive tasks. Sarma et al. [30] represented the map-and-reduce communication cost as the number of data entries that need to be generated from the map input for a distributed application with a given parallelism factor. To decrease the communication cost for a specific distributed application, one need redesign the distributed 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 transfer size of data between different computer nodes. An Iterative distributed application can be first designed with the minimum communication cost for each iteration, if possible, and then apply Confluence to decrease the shuffle size. ShuffleWatcher [31] scheduled the locality of map tasks and reduce tasks to decrease shuffle inter-node traffic for the single iteration of a MapReduce job, but it did not provide a solution for iterative distributed operations.

Datacenter networking for shuffle: In datacenters, various networking scheduling algorithms were proposed to improve the shuffle throughput for different performance goals [13], [32], [33], [34]. Confluence does not concern with the underlying network level. However, by decreasing shuffle workloads, shuffle tasks applying the confluence key partitioning scheme 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 [10], [35], or to avoid an unbalanced allocation of workloads by considering compute skew problems [36], [37]. They concerned with the placing of map tasks where the 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. Confluence localizes data and tasks by partitioning the data corresponding to each pure confluence subgraph with multiple shuffle iterations in the same node, which eliminates the shuffle traffic of multiple iterations while not impacting the balance of workloads. Aaron et al. [38] addressed the problem of stragglers in iterative machine learning. If the unbalance in the granule of pure confluence subgraphs does exist in some particular dataset or algorithm, the partitioning scheme of the pure confluence subgraph can be manually specified by the user to prevent stragglers.

DAG scheduling: Similar to other DAG task scheduling algorithms [9], [18], [19], [39] that schedule tasks to based on the task dependency graph, Confluence optimizes the partition location based on the key dependency graph. The difference is, in terms of the goal, that the DAG task scheduling algorithms tried to parallel the execution of jobs that are not depended on each other to increase parallelism, while Confluence isolates the partitions of the key-related data in multiple shuffle iterations to reduce network traffic.

8 Conclusion and Future Work

We have presented Confluence Key Partitioning scheme, the first work on decreasing the shuffle size of iterative distributed operations leveraging the key dependency. Confluence guides the data partitioning across different iterations based on the key dependency graph, which is constructed

based on the iterative key dependency. Confluence greatly decreases the overall shuffle size while not introducing the workload skew side effect for a variety of distributed operations in fields of 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 sample data, we can keep track of how data are flowing between nodes across different iterations, and thus find out the key dependency of these sample data. The challenge is that how to expand key dependency of the sample data to the whole dataset. If the data flow tracking generates the dependency of specific keys, there can be difficulties to generalize it to the dependency of key patterns. Even when we have the key pattern dependency of the sample data, we need to consider whether the key pattern dependency is applicable to the whole dataset, or how to refine the key pattern dependency for the whole dataset if necessary.

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] 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, Nov. 2014.

[3] 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.

[4] 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.

[5] M. Armbrust et al., "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1383–1394.

[6] 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.

[7] T. Kraska et al., "Mlbase: A distributed machine-learning system," in *Conference on Innovative Data Systems Research (CIDR)*, vol. 1, 2013, pp. 2–1.

[8] 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.

[9] 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.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] K. Shvachko et al., "The hadoop distributed file system," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies*. IEEE, 2010, pp. 1–10.

[12] 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.

[13] M. Chowdhury et al., "Managing data transfers in computer clusters with orchestra," in *Proceedings of the ACM SIGCOMM Conference*. New York, NY, USA: ACM, 2011, pp. 98–109.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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–.

[19] 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–.

[20] "Movielensals spark submit 2014." [Online]. Available: https://databricks-training.s3.amazonaws.com/movie-recommendation-with-mllib.html

[21] "Multiadjacentlist benchmark." [Online]. Available: https://github.com/liangfengsid/MultiAdjacentList

[22] "Spark kmeans benchmark." [Online]. Available: http://spark.apache.org/docs/latest/mllib-clustering.html

[23] "Hku gideon-ii cluster." [Online]. Available: http://i.cs.hku.hk/%7Eclwang/Gideon-II/

[24] 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.

[25] R. Bekkerman, M. Bilenko, and J. Langford, *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] F. Ahmad et al., "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 1–13.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

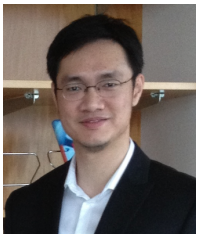
[36] Y. Kwon et al., "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 75–86.

[37] 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.

[38] A. Harlap et al., "Addressing the straggler problem for iterative convergent parallel ml," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: ACM, 2016, pp. 98–111.
[39] Spark job scheduling: Dagscheduler. [Online]. Available: https://spark.apache.org/docs/1.4.0/job-scheduling.html

Feng Liang is currently a PhD candidate in Department of Computer Science, The University of Hong Kong. He received his bachelor's degree in software engineering in Nanjing University in 2012. His research interests are mainly on distributed file systems, distributed computing, and machine learning. 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