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 computer nodes, which generates significant network traffic and often dominates completion time. This traffic is particularly pronounced in iterative distributed applications because each iteration will invoke a shuffle operation. ~~Our key observation is that many iterations can aggregate and thus greatly reduce key-value pairs based on some specific keys.~~ Our key observation is that keys are relevant between iterations and key-value pairs of a specific key pattern can be localized in an iteration to avoid shuffling data between nodes in other iterations. For instance, in MatrixMultiplication, vector entries are multiplied, respective, to be addends for the following add operation, and the input vector entries for the same result entry can be organized in the same node so that the multiply and the add operation for one result entry can be finished in one node without shuffling. If we have an efficient key partitioning scheme that partitions key-value pairs based on these specific key patterns on the same node, we can greatly reduce the network traffic.

In this paper, we define these relevant keys as key dependencies, and we present a new data structure called Key Dependency Graph (KDG) to keep track of them for all iterations. We implement a system called Confluence, where developers construct a KDG for a distributed application, and Confluence automatically generates an efficient key partitioning scheme from the KDG. ~~After applying Confluence, the datasets needed by the multiple iterations will be partitioned and assigned to the nodes where they will be needed so that network traffic is eliminated and the overall shuffle workload is decreased.~~ ~~, while keeping the workloads in a balanced state.~~ We demonstrate the practicality of Confluence on diverse applications ~~The experimental results on a Spark cluster~~ and show that Confluence can greatly decrease the shuffle size for the typical iterative distributed applications by up to 50%.

Index Terms—Spark; Shuffle; MapReduce; Iterative Distributed Operation; Key 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], Mesos [9] and Dryad [10] 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 [11], ~~being~~ one of the most popular distributed paradigms, provides ~~the~~ 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 [12]. 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 significant shuffle workloads greatly impacts the performance of ~~shuffle~~ distributed operations. Most of the distributed computing paradigms include the shuffle operation, which transfers the intermediate output data of the map operations to the computer nodes doing the reduce operations. The shuffle operation transfers data from a group of nodes to the other nodes in the same group, which follows a many-to-many communication pattern. The shuffle operation may require a large amount of network bandwidth and sometimes even dominate the job completion time especially for the shuffle-heavy jobs, where the volume of data to shuffle is large comparing to the input data size. The study based on the Yahoo! work trace has revealed as much as 70% of the jobs are shuffle-heavy [13] and the shuffle completion time can account for as much as 33% of overall completion time [14], [15]. The heavy shuffle workload problem is pronounced in iterative distributed operations, where the shuffle operations transfer large volumes of data between every two iterations.

Several iterative distributed computing paradigms [16], [17] have been developed to overcome the disk I/O overheads of the computing tasks that include multiple iterations of distributed operations and thus multiple shuffle operations. For instance, Spark [17] adopts the in-memory approach to reuse data across iterations in the memory to avoid the overhead of transferring the intermediate data into and out of the disk. However, none of them proposes a useful method to minimize the size of the data to be transferred in multiple shuffle iterations.

Our key insight 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

- *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*

decrease the transferred data across shuffle iterations. ~~The dataset entries are generally in the format of key-value pairs. The iterative key dependency relationship reflects the logic of the program: given a key-value data entry as the input, what new key-value data entries will be generated as the output in every iteration of the distributed operation. It represents the transformation process from key-value pairs to new key-value pairs in iterative distributed operations and it can be expressed to computer by means of the Key Dependency Graph . It is special to every iterative distributed operation and is decided by the logic of the code and the input dataset.~~ How the output data of the previous shuffle iterations are partitioned across different locations can greatly affect the data are to be shuffled and its efficiency in the following computing iterations. ~~With the knowledge of the key dependency,~~ 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 the shuffle operations (we call that the "shuffle size" in the rest of this paper). But by 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. We use a simple example to illustrate this idea.

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

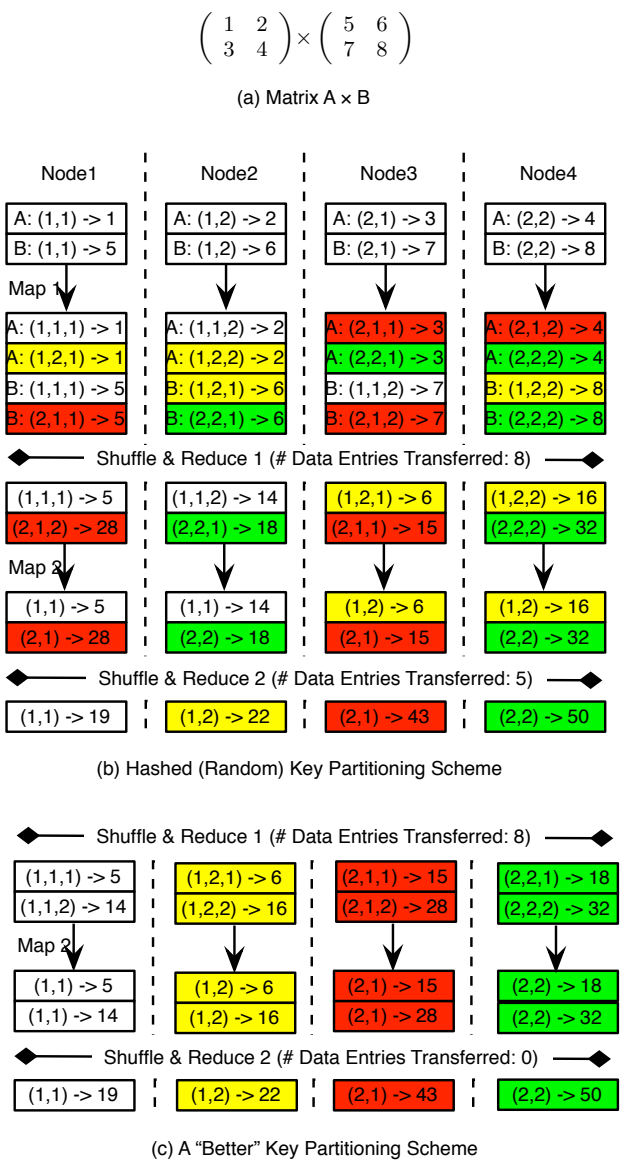


Fig. 1. An Example of the MapReduce-style Matrix Multiplication Algorithm with Different Key Partitioning Schemes

From the KDG, we can identify the subgraphs in which the dataset entries with the keys in the source nodes will only generate the dataset entries with keys in the downstream nodes. We call 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), the programs need to add the corresponding user-defined partitioner to guide the assignment of data entries to their desired locations. Most distributed computing paradigms, such as Spark [17] and Twister [18], provide an interface for adding a user-defined partitioner for the shuffle operation. By applying the CKP scheme, the 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.

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 gave 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 several typical iterative distributed applications from different fields. The experiment on the medium-size Spark cluster demonstrates that CKP could decrease the overall shuffle traffic workload 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 3, 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 volume cannot be stored in a single computer node and are often separated into several partitions, where each partition is distributed to a node in the computer cluster. We refer to a dataset which is separately stored in several nodes as a distributed dataset. The function whose inputs include the 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 the input dataset before the final results are obtained. Each iteration usually takes the outputs of the previous iteration as the input dataset.

One typical example of the distributed operation is MapReduce [11], which divides the operation into two primitives: map and reduce. Many iterative distributed paradigms such as the in-memory paradigm Spark [17] and the distributed database query engine Hive [3] can be equated to multiple iterations of MapReduce operations. For iterative distributed operations, we refer to the general iterative map-and-reduce operation, where the "map" primitive performs one or more transformation operations on the datasets in each local node, and the "reduce" primitive would re-partition the datasets by transferring the data entries with the same keys to the designated locations and group the entries of the same key to one key-value pair, where this *value* is a list of the values that associate with the key. The output of each data entry of the reduce primitive is also called as the key-value-list pair in this paper. The reduce primitive does not involve any transformation on the values of the entries and is also known as the shuffle operation.

Note that division of map and reduce in the matrix multiplication example in Section 1 is slightly different from the definition here. In the example, the reduce primitive aggregate the value list of each key to a value, which follows the traditional style of MapReduce. While by the definition here, the aggregate operations join the other data transformation operation in the map primitive of the next iteration. Each iteration of distributed operation can have a series of transformations plus at most one shuffle operation. In the rest of this paper, the terms of the reduce primitive and the shuffle operation may be used interchangeably.

The traffic pattern of the data to shuffle in each iteration is directly dictated by the key partitioning scheme. A random or careless key partitioning scheme may cause unnecessary shuffle traffic across multiple iterations of distributed operation. This is what we set out to reduce using our partitioning scheme.

2.2 Dependency Graph

The dependency graph is a directed graph usually used for describing the dependency relationship of instructions, tasks, data, etc [10], [19], [20]. The scheduler can use the dependency graph to decide the order the execution of instructions or tasks. In distributed frameworks, the scheduler uses the data dependency graph or task dependency graph to make the tasks execute in parallel in the cluster. As the tasks are meant to process their specific data, the task dependency graph is the data dependency graph as well.

In the data dependency graph, the datasets are divided into graph nodes based on locations and the dependency relationship is constructed upon the execution stages and data locations. The data partition information is specific to the application running situation and the cluster environment. In the key dependency graph proposed in this paper, the construction of the graph does not use any data partitioning information, but considers the key dependency of data, which depends on the application logic.

2.3 Iterative Shuffle Size

We model the overall shuffle size of the iterative distributed operations and discuss the time complexity of obtaining the optimal key partitioning scheme that minimizes the overall shuffle size.

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

$$A'_i = m_i(A_{i-1}) \tag{1}$$

$$A_i = r_i(A'_i) \tag{2}$$

, where A_{i-1} is the output of the $(i-1)_{th}$ iteration as well as the input of the i_{th} iteration, m_i is the map primitive of the i_{th} iteration that operates on A_{i-1} locally without changing the partition locality, and r_i is the shuffle operation of the i_{th} iteration that takes the output A'_i of the map primitive as the input. The keys of A_i and A'_i are the same, but the partition localities are different. In the first iteration, when i is equal to 1, A_0 represents the raw input datasets before any processing.

Suppose in the i_{th} iteration of the shuffle operation, in order to collect an entry $a \in A_i$, the size of data that need to be transferred is a function of a and A'_i: $s_i(a, A'_i)$. The function s_i can be different in different key partitioning schemes. The data transfer size to obtain A_i is:

$$S_i = \sum_{a \in A_i} s_i(a, A'_i). \tag{3}$$

Within the same i_{th} iteration, for $\forall a_p, a_q \in A_i$ where $a_p \neq a_q$, the values of $s_i(a_p, A'_i)$ and $s_i(a_q, A'_i)$ are independent of each other. It means that if the value of $s_i(a_p, A'_i)$ changes due to a different partition location of a_p, the value of $s_i(a_q, A'_i)$ remains unchanged unless a_q is partitioned to a different location.

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

$$S = \sum_{i=1}^{m} S_i = \sum_{i=1}^{m} \sum_{a_i \in A_i} s_i(a, A'_i). \tag{4}$$

If values of $S_i (1 \leq i \leq m)$ are independent of each other, S is a linear function and S can be minimized simply by minimizing each $s_i(a, A'_i)$. However, most usually, S_i and S_{i+1} are correlated. The value of S_i depends on the logic of the function m_i in Formula 1, the partition locality of dataset A_{i-1}, as well as the locations where each entry of A_i is partitioned to. The partition locality of A_i inherited from the i_{th} iteration will again affect the value of S_{i+1}.

If the contents of $A_i (i = 1, 2, ..., m)$ are already known, to find out the optimal key partitioning scheme that minimizes the overall data transfer size S, we need to exhaustively explore the possible key partition schemes of all the key-value pairs across all the iterations. By calculating the overall data transfer size of each scheme, the optimal solution is the scheme with the minimal size. The time complexity of the exhaustive method is $O(n^{\sum_{i=1}^{m} |A_i|})$, 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 an NP-complete problem.

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

3 CONFLUENCE

Even though it is hard to find the optimal partition solution to minimize the overall shuffle size, if the code logic of the specific distributed operations is already known, the shuffle size can be reduced to the maximum extent by exploring the key dependency of different iterations.

This section firstly describes the the construction process of the key dependency graphs, as well as its properties. After that, we present Confluence Key Partitioning Scheme, which decreases the overall iterative shuffle size by applying the properties of the key dependency graph, while not increasing the workload skew level.

Algorithm 1: Construction of the Key Dependency Graph

Input : Key Set K_i, containing all the possible keys (or usually, key patterns) of the output key-value pairs of any Iteration i;
Key Dependency Set D, containing the key dependency of all iterations;
Output: Key Dependency Graph $G = (V, E)$

```
/* Construction of V                                   */
foreach Iterations i do
    foreach k ∈ K_i do
        add vertex v_{i,k} to V
    end
    if any possible key (key pattern) is unknown then
        add vertex v_{i,x} to V
    end
end
/* Construction of E                                   */
foreach Iterations i do
    foreach key dependency v_{i,k1} ⇒ v_{i+1,k2} in D do
        add edge (v_{i,k1}, v_{i+1,k2}) to E
    end
end
```

Fig. 2. Pseudo Code of Constructing the Key Dependency Graph

3.1 Key Dependency

Given a key-value pair as the input to an iteration of the distributed operation, it will generate a (or a set of) new key-value pair(s) based on the logic of the program. The distributed key dependency represents such logic of the generation of key-value pairs from the input key-value pairs in every iteration of the distributed operation. The programmer is supposed to be well acquainted with the distributed operation logic. By saying that we know the key dependency for a specific iterative distributed operation, it means that we know what the output key-value pairs will be for an input key-value pair in every computing iteration. We use the key dependency graph (introduced later) to portrait the iterative key dependency so that it is comprehensible to the computer. Every distributed operation has its own key dependency and it relies on the programmer to understand the iterative key dependency based on the application logic and construct the key dependency graph.

3.2 Key Dependency Graph

The directed graph $G = (V, E)$, which is named as the *key dependency graph* (KDG), addresses the key dependency status of the distributed operation iterations.

The key pattern is the symbol that represents a set of keys that match a specific pattern. The key dependency is denoted with two vertexes $v_{i,k1} \Rightarrow v_{i+1,k2}$ if the output key-value pairs with key matching pattern $k1$ in Iteration i can generate a key-value pair with key matching pattern $k2$ in Iteration $i+1$. The key dependency is specific to the code logic of the distributed operation and it decides which graph node should point to which others. The distributed operation code logic should be known prior by the programmers.

The construction process of the key dependency graph G is shown as in Algorithm 1. Note that when constructing E, for the unknown keys (or key patterns), it is assumed that $v_{i,x} \Rightarrow v_{i+1,k2}, \forall k2$. The unknown key dependency is only raised when there are some key values (or key patterns) the programmer cannot make sure in a computing iteration. In practice, the input set K_i in Algorithm 1 is defined to contain the key patterns instead of the distinguished key values, so that a node in the KDG represents a key pattern. Such practice limits the width of the KDG in each level and our algorithm introduced later can partition the key-value pairs based on their key pattern. In the rest of this paper, unless specified, the input set K_i is the key pattern set, but for generality, we still refer to it as the key set.

This graph is a directed acyclic graph (DAG). The time complexity to construct this graph is $O(\sum_{i=1}^{m} k_{i-1}k_i)$, where k_i is the number of keys (or key patterns) in iteration i and m is the total number of iterations. Usually, k_i is very small in value when it is referred as the key pattern (not the distinguished values of the keys). For example, in the matrix multiplication application, the subgraph from the second level to the third (final level) of the KDG can be denoted as a one node to one node subgraph: $(a, b, [1, n]) \rightarrow (a, b)$, where a and b stand for any row and any column index of the matrix, and $[1, n]$ stands for the range from 1 to the the matrix dimension length n.

The subgraph $C = (V', E')$ of $G = (V, E)$ is called a *confluence subgraph* if it satisfies the following *outward exclusive* requirements: denote V'_i as the set of all the vertexes in V' which are in level i in G. V'_i satisfies $V'_i = \bigcup_{v'_{i-1} \in V'_{i-1}} V_i$, where V_i is the set of vertexes that v'_{i-1} has an edge $e \in E$ points to. The edge $e' = (v'_1, v'_2)$ is in E' iff $v'_1 \in V', v'_2 \in V'$ and $(v'_1, v'_2) \in E$.

In other words, in the confluence subgraph, all the vertexes in the upper level have no edge pointing to the vertexes in the lower level outside the confluence subgraph. Let Level u and Level w be the uppermost level and the lowest level that V' contains, respectively. V'_u is called the *source key set* at level u and V'_w the *confluence key set* at level w of the confluence subgraph $C = (V', E')$.

We call the confluence subgraph $C = (V', E')$ the *pure confluence subgraph* if it further satisfies the *inward exclusive* requirement: $V'_{i-1} = \bigcup_{v'_i \in V'_i} V''_{i-1}$, where V''_{i-1} is the set of vertexes that have an edge $e \in E$ pointing to v'_i.

An example of the KDG of 3 levels and some of its confluence subgraphs are shown in Fig. 3. The subgraph in the dashed boxes C3 is a pure confluence subgraph of C1. The subgraph in C2 is a confluence subgraph but not a pure confluence subgraph, as the source node of the edges $(v0x, v11)$ and $(v0x, v12)$ is not in C2. The subgraph in the dotted box C4 is not a confluence subgraph as the edge

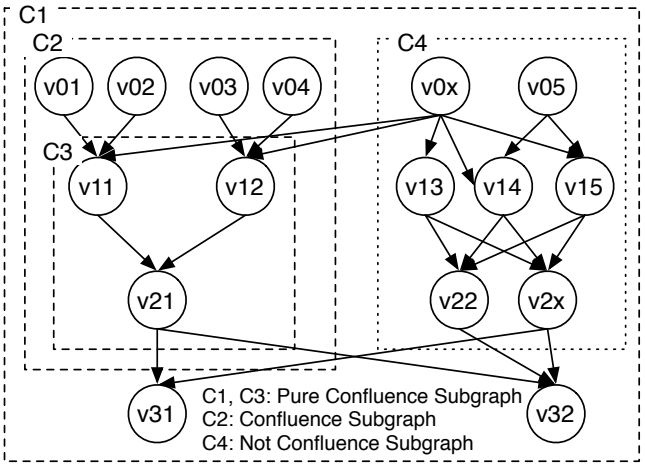


Fig. 3. A Key Dependency Graph Example

$(v0x, v12)$ points to the node $v12$, which is not a vertex in the subgraph C4. Note that the key dependency graph is always the pure confluence subgraph of itself, as show in the dashed box C1.

3.3 Properties of Confluence Subgraph

For the distributed operations of several iterations, suppose that there is a confluence subgraph C of its KDG with the source key set V'_u at level u and the confluence key set V'_w at level $w (w > u)$. Let B_u be a part of the output dataset of the u_{th} iteration and B_w be the dataset generated by B_u after $(w - u)$ iterations. We have a theorem that can easily be conducted by the definition of the confluence subgraph.

Theorem 1. If the keys of all the entries of B_u are in the source key set V'_u, the keys of all entries of B_w are in the confluence key set V'_w.

We call the datasets B_u and B_w in Theorem 1 the *upward closure pair* datasets of the confluence subgraph. It can be further conducted from Theorem 1 to get:

Theorem 1.1. If the subgraph C is a pure confluence subgraph and the keys of all the entries of B_u are in the source key set V'_u, when B_u is stored in one single node, the minimum data transfer size for obtaining B_w is 0.

Theorem 1.1 indicates that if the datasets B_u and B_w are the upward closure pair of a pure confluence subgraph and B_u is partitioned in one node, we can finish all the following $(w-u)$ iterations of distributed operations in the same node to obtain A_w, without any data transferring. We name this key partitioning scheme the **Confluence Key Partitioning** scheme.

3.4 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

$$s_i(a, A'_i) = \sum_{j=1}^{n} [\frac{1}{n}|a|(1-p_j)] = \frac{(n-1)}{n} \cdot |a| \qquad (5)$$

, 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.

***Theorem* 2.** The data transfer size for collecting the uniformly-partitioned entry in the shuffle operation is decided only by the number of nodes in the cluster and the length of this entry.

By Theorem 2, 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: $S'_i = S''_i, 1 \le i \le u-1$.
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 1.1. We have $S'_i = S''_i - \frac{n-1}{n}|B'_i|$, where $r_i(B'_i) = B_i, u+1 \le i \le w$.
4) From iteration $(w+1)$ onward, the data entries are uniformly-partitioned entries again: $S'_i = S''_i, i \ge w+1$.

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.

3.5 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 3.4 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. And 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 ~~semantics~~ logic of the distributed operations. Suppose that Iteration $u'(u+1 \le u' \le w)$ 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, $L_i \ge 0, u'+1 \le i \le w+1$.

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.

3.6 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.

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 [21] 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. 4(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. 4(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.

TABLE 1
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	$\lvert A\rvert \cdot \sum_{i=2}^{m}(\sum_{j=1}^{k} p'_{ij} - 1/n)$ upper bound:$\lvert A\rvert \cdot (m-1)(n-1)/n$ lower bound:$-\lvert A\rvert \cdot (m-1)/n$

4.2 MultiAdjacentList

MultiAdjacentList [22] 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 act as the value. The evolution of the data of multiple iterations is shown as Fig. 4(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. 4(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 3.5, the workload skew level of each iteration is 0 when CKP is applied.

4.3 KMeans

KMeans [23] 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 $points \Rightarrow centers$

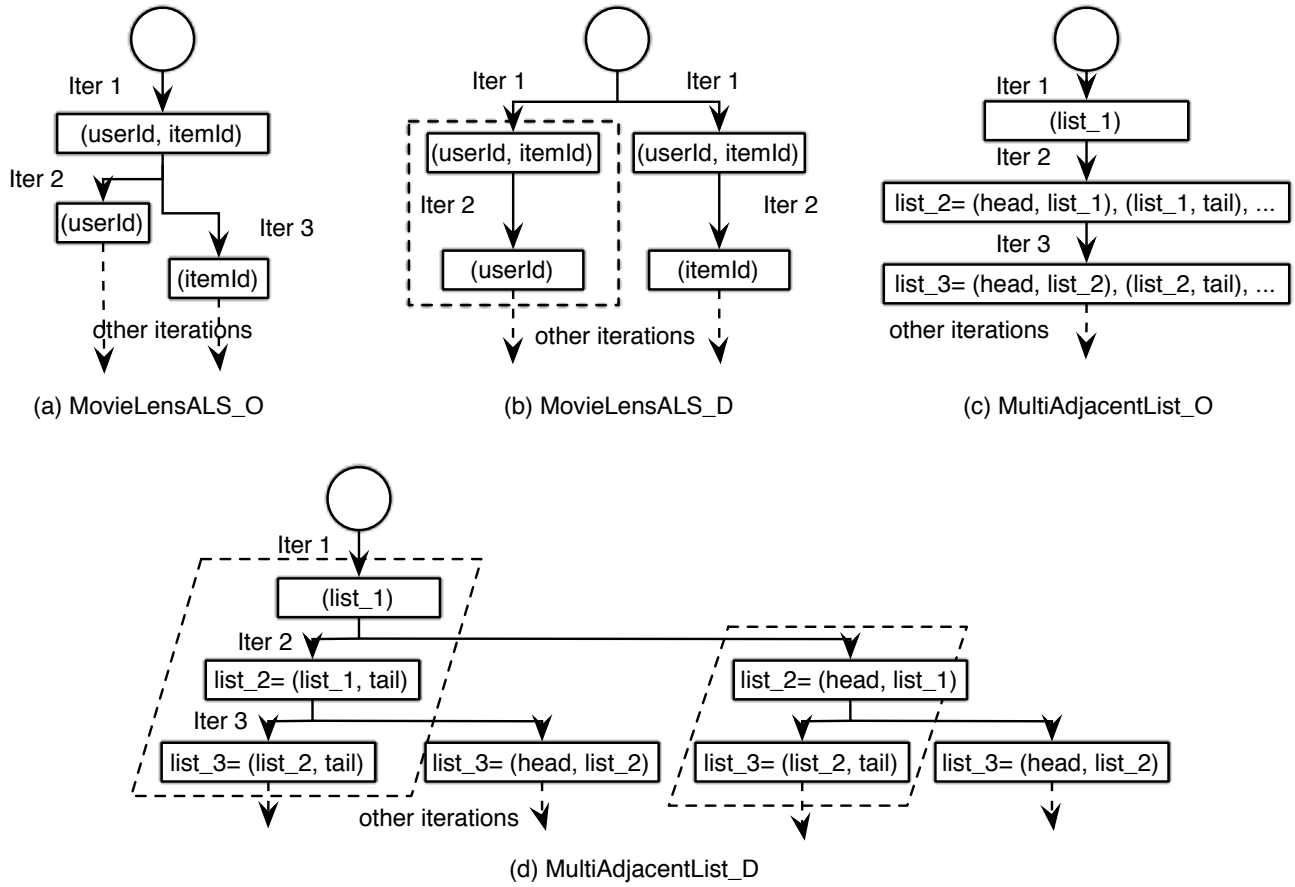


Fig. 4. The Original (Suffixed by _O) and Divided (Suffixed by _D) Iterative Evolution of the Datasets of the MovieLensALS and MultiAdjacentList Applications. The Keys of the Datasets are Marked by Parenthesis. Pure Confluence Subgraphs can be Found in the Computing Iterations in the Dashed Boxes

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 1. 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.

TABLE 2
User-Defined Functions of ConfluencePartitioner in Different Distributed Operations

Distributed Operations	Key Dependency	User-Defined Function
MatrixMultiplication	$(i, j, p) \Rightarrow (i, j)$	(a:Any) => a match {case (_ , _, _) => (a._1, a._2)}
MovieLensALS	$(user, item) \Rightarrow user$	(a:Any) => a match {case (_ , _) => a._1}
MultiAdjacentList	$list \Rightarrow (list, tails)$	(a:Any) => a match {case s:String => {s.split(" ")(0) }}
KMeans	$points \Rightarrow cluster$	(a:Any) => a match {case point:Vector[Double] => closestPoint(point, cluster_centers)[1]

[1] closestPoint(point, cluster_centers) is the function which returns the index of the point in "cluster_centers" that is in the shortest Euclidean distance with "point". The "cluster_centers" are the cluster centers of a fixed iteration, e.g., the first iteration.

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.

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

TABLE 3
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

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 2 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.

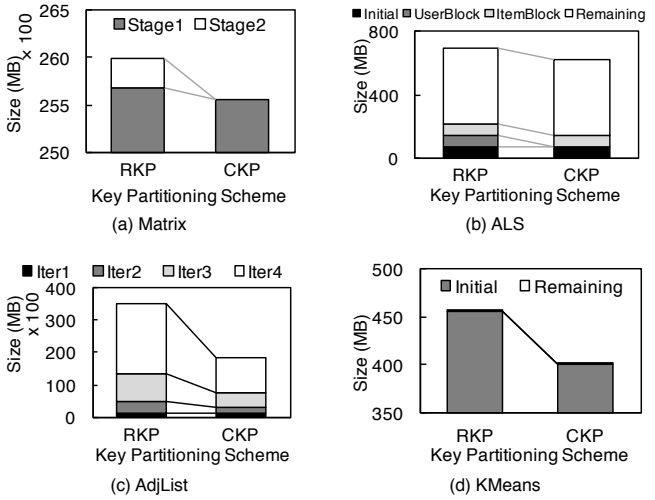


Fig. 5. Shuffle Sizes of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

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 [24], 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 3. 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.

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. 5, 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. 5(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.

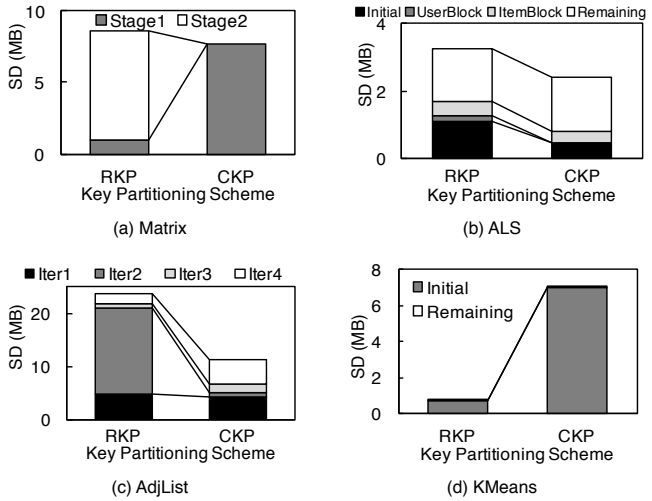


Fig. 6. Standard Deviations (SD) of the Shuffle Sizes of the Executors of Multiple Iterations (or Stages) in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks

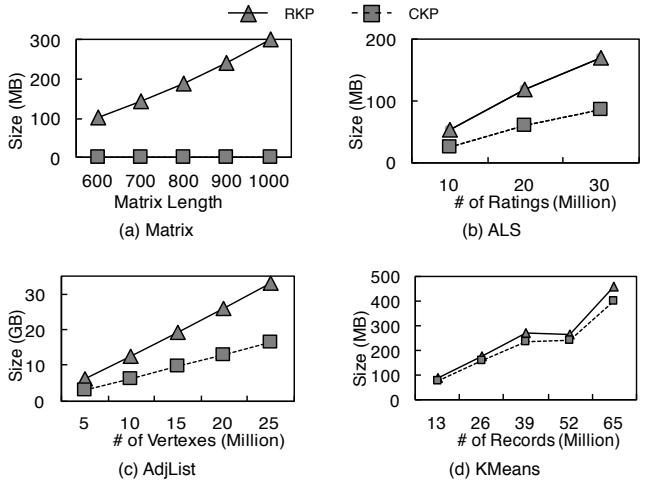


Fig. 7. Overall Shuffle Size of the Pure-Confluence-Subgraphs-Related Iterations in MatrixMultiplication (Matrix), MovieLensALS (ALS), MultiAdjacentList (AdjList) and KMeans Benchmarks with Different Input Size

The benefit of applying CKP on the smaller iterative distributed operation divided based on operations can be read in Fig. 5(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. 5(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.

6.4 Workload Skew

The standard deviations of the shuffle workloads of the executors of different benchmarks are shown in Fig. 6. As expected, the workload standard deviations of the CKP-affected stages of MatrixMultiplication (Fig. 6(a)), MovieLensALS (Fig. 6(b)) and MultiAdjacentList (Fig. 6(c)) are all 0. In the KMeans benchmark, the standard deviation of CKP is larger than that of RKP (Fig. 6(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. 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 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 [16], [18], [25] improved the distributed frameworks for the iterative MapReduce-like computing by extended the programming interfaces to support multiple map and reduce phases. The general idea was to cache the output data of each iteration for the next iteration. Spark [17] provided more kinds of operation interfaces and supported a series of transformations on the distributed input datasets in the memory by the concept of resilient distributed dataset. These iterative distributed frameworks tried to minimize the I/O overhead by storing the intermediate data of each iteration in the memory, instead of in the disk like Hadoop. However, they did not consider the heavy network workload during the shuffle operations. Inheriting the benefit of fast memory I/O speed, Confluence decreases the shuffle sizes and alleviate the network workload by considering the key dependency of multiple relational iterations.

Parallel&Distributed algorithms optimization: Lots of work has been conducted on the optimization of parallel&distributed machine learning algorithms [26] and science computing [27], [28], including the matrix multiplication methods [29], [30]. Generally, they focused on optimizing the algorithm itself by decomposing the tasks to increase the task parallelism and removing the synchronization boundary between the I/O-intensive tasks and compute-intensive tasks. Sarma et al. [31] represented the map-and-reduce communication cost as the number of data entries needed to generate from the map input for the 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, while the shuffle size transfer size of data between different computer nodes. The algorithms with the minimum communication cost can further decrease the shuffle size by considering the key dependency of the datasets in multiple iterations. ShuffleWatcher [32] scheduled the locality of the map tasks and the reduce tasks to decrease shuffle inter-node traffic for the single iteration of the MapReduce jobs, but it did not provide the 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 [14], [33], [34], [35]. Confluence does not concern with the underlying network level. However, by decreasing the shuffle workload, the shuffle tasks applying the confluence key partitioning scheme can work on these shuffle-optimized datacenter networks seamlessly and gain more performance increase.

Skew&Straggler: The distributed computing paradigms like MapReduce adopted the data locality principle, either to place the computing tasks closed to data locations in priority to reduce the traffic workloads of data migration [11], [36] or to avoid the unbalanced allocation of workloads by considering the compute skew problems [37], [38]. 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 task-to-data approach cannot help in the shuffle tasks. Confluence localizes the 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 the following iterations while not impacting the balance of workloads. Aaron et al. [39] addressed the problem of stragglers in iterative machine learning. If the unbalance in the granule of pure confluence subgraph does exist for 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 the DAG task scheduling algorithms [10], [19], [20], [40] that schedule the 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 data with dependency relationship in multiple shuffle iterations to reduce the 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 the fields of scientific computing, machine learning, data analysis, etc.

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] B. Hindman et al., "Mesos: A platform for fine-grained resource sharing in the data center." in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, vol. 11, 2011, pp. 22–22.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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–.

[20] 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–.

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

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

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

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

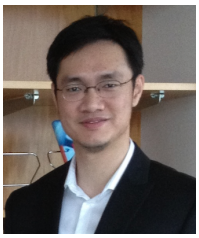
[38] Y. Kwon et al., "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012, pp. 25–36.

[39] 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.

[40] 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