Confluence: Speeding Up Iterative Distributed Operations by Key-dependency-aware Partitioning

Response to Review Comments

In the previous draft, we modeled iterative distributed operations by constructing the key dependency graph (KDG) and applied Confluence key partitioning (CKP) by finding the pure Confluence subgraph, which seemed to be a complex procedure. This intricate model is the root cause of wonders:

(a) how efficient the procedure itself is, as it seems we need make efforts and spend time going through this procedure;

(b) what actual effect of CKP is when applied in a distributed application, as we may not directly deduce it from the KDG;

(c) how CKP can be applicable to complicated distributed applications since KDG would seemed very complicated then, without mentioning to find the pure Confluence subgraph out of it.

We set off by finding solutions to solve the complicated model problem. In this revised version, we formalize data relation between different iterations by the much simpler notion of "key dependency". We find that the key dependency model is not only much simpler in form, but also easily answers the above questions.

A key dependency looks like: $k \Rightarrow_{pr} f(k)$

The key dependency captures the transformation logic of distributed operations. Compared to the method based on KDG, the difference by using key dependency is:

(1) Finding the key dependency need not consider every edge between key nodes individually in the graph. Programmers need only find one or a few key dependencies that are important. In a sense, programmer need not construct the whole KDG at all, but only presenting some edges in the form of key dependency is enough for applying CKP. This change answers question (a).

(2) A key dependency not only roughly indicates two key are related, but also includes a symbol "pr", called dependency probability, that more precisely capture the statistical relation by a probability model. With the dependency probability, programmers can easily deduce the benefit of binding key partitions based on the key dependency. This change answers question (b).

(3) The technique of binding key partitions perfectly match the notion of key dependency. The key dependency is useful because we find that by binding the partitions of key k and key f(k) on both side of a key dependency, the shuffle size almost always decreases compared to the random partitioning. Besides, the key dependency directly maps to the implementation. Programmers can apply CKP in the program with as simple as a single line of code based on the key dependency, without any extra effort. This change answers question (c).

We replace the KDG model with the key dependency model in the revised draft. With the new key dependency model, we find the description of applying CKP becomes tidier and clearer, because of the

precise nature of key dependency. In addition to this, we have made lots of other major changes, which are listed in the summary of changes. They answer most questions in the comments given by reviewers.

Major Response List:

(1) What are the limitations of the proposed mechanism?
Response: We addressed this question in Section 4.5, by stating the limitation circumstances of CKP and suggesting solutions to these circumstances.

(2) How hard to derive a KDG, especially developers may modify the original distributed operations?
Response: We addressed this question in Section 3.2 and 4.5. With the new key dependency notion, the key dependency can be derive easily in most application, specially, O(1) in some extreme cases. Besides, applying CKP based on the key dependency in the program code is simple by the ConfluencePartitioner interface, as shown in Section 5.2.

(3) Given a developer-provided KDG, the following steps, i.e., extracting Confluence Subgraph and generating the CKP scheme, need to be automated. This part is done manually and not clearly described in the manuscript.
Response: The new key dependency notion is simpler than the KDG in the previous draft, and there is no need to extract a subgraph. Though the key dependency is still manually provided by programmers, it is much easily in understanding and in time. It is addressed in Section 3.2.

(4) The end-to-end performance needs to be evaluated to demonstrate the effectiveness of the approach.
Response: We evaluate the end-to-end performance by completion time. Results show that CKP can reduce the job completion time by as much as 23%.

(5) Discussing the related work.
Response: We discussed the related work of "SCOPE" in Section 7, and mention the contrast of shuffle evaluation work in the Introduction section.

Summary of changes:

1. Introduction: we improve the contribution part by the power of key dependency and an emphasis on the effect and applicability of CKP.

2. In the background part, in order better analyze the shuffle behavior in iterative distributed operations and to avoid confusion with the map-and-reduce primitive, we redefine the iterative distributed operation model by the transform-and-shuffle primitive (Section 2.1).

3. In Section 2.2, we adjust the iterative shuffle size model based on the new transform-and-shuffle primitive.

4. Instead of proposing the KDG in the previous draft, we formally define the key dependency by in Section 2.3.

5. We add a section 2.4 to show how the shuffle size of the random key partition scheme can be obtained from the key dependency information.

6. In the previous draft, we localized data corresponding to a pure confluence subgraph in a node. In the revised draft, we propose a similar but simpler technique called “key partition binding” in Section 3.1. The benefit of binding key partition based on the key dependency is obvious and can be easily deduced. This technique along with the notion of key dependency is the foundation of CKP.

7. Rather than merely present the algorithm of CKP and calculate the shuffle size in the previous draft, we discuss and explore more in detail the property, the effect and the potential difficulty of apply CKP (Section 3.2). We discuss the complexity of applying CKP and show that it is easy with almost no cost. We discuss the features of the algorithm of CKP to clarity potential difficulty in practice. We discuss things to note in order to make CKP more effective in distributed applications.

8. In the workload skewness section, rather than calculate the value of workload skewness iteration by iteration, which is incomprehensible, we discuss the conditions for CKP in order not to increase workload skewness. Once the conditions are satisfied, CKP does not introduce extra workload skewness.

9. We re-discuss the impact of inaccurate modeling of key dependency. The notion of key dependency makes the discussion clearer.

10. In the application section, besides adjusting the description of the existing distributed applications with the new key dependency notion, we make two major changes. The first is that we add the PageRank and GoogleAlbum example (Section 4.2), which are representatives of the “join” operation. The second is that we add a subsection that discusses limitations when applying CKP in distributed application and suggests solutions or workarounds to solve these problems (Section 4.5).

11. We remove the map-side combine part in the previous draft, because we found that after using the transform-and-shuffle primitive, the map-side combine is naturally a part of the transform operation and need not additionally be mentioned.

12. In the implementation section (Section 5.2), we show the usage of the ConfluencePartitioner interface by comparing the code after applying CKP in the program. The code comparison shows that the ConfluencePartitioner interface is convenient and applying CKP is very easy.

13. Improve the evaluation (Section 6) add supplementing the completion time evaluation to verify end-to-end performance benefit of CKP.

14. In relate work (Section 7), we add a discussion on similar work on shuffle partitioning based on the data relation between iterations.