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

Response to Reviewers' Comments

This revision of the paper focuses on the improvement of major aspects as raised by the reviewers, namely, the skewness analysis, the executor tying issues, and the data compression influence.

1. Comment by Reviewer 1: "The added discussion on **workload skewness** is not clear on the other hand. The use of the word distinguished is confusing in the subsection. There also seems to be no mention about data skew in the values distribution between keys, which is systems like map reduce causes problems regardless of the balanced random hash partitioning. This subsection should be improved. "

Response: the last version where we said "keys of input entries follow a distribution Dist", "Dist" was actually the value distribution of the keys. By considering the key distribution between nodes in joints, we obtained the workload skewness of RKP. We found such a presentation somewhat confusing. Therefore, in Section 3.3 of the revised version, we redefine the distribution model, which divides the value distribution of partition nodes into the joint of three sub-distributions: Dv |><| Dk |><| D_f(k), where Dv is the values distribution between keys, Dk the keys between mapped keys, and D_f(k) the mapped keys between partitions. We found that this redefinition is more precise on one hand, and it can describe both the skewness of RKP and that of CKP on the other hand. Moreover, an observation on the distribution "Dk", which is the only different one, helps us to easily conclude in which condition CKP would not increase the workload skewness as compared to RKP. We highlight this in the last paragraph of Section 3.3 to avoid the misunderstanding that CKP would remove workload skewness. In fact, it only tries not to increase it.

2. Comment by Reviewer 1: "When discussing the concept of key dependencies, probabilities could be explained a bit more. Every key has a probability distribution between zero and one to be mapped to a key in the next stage. The absolute cases of zero and one are clear, but what about more uncertain probabilities? It would be interesting to discuss these elements further, possibly including one illustrative example. "

Response: The calculation of dependency probability would not be clear enough if it is not illustrated with an application example. We found that we have discussed in detail dependency probability for cases where it is not 1, i.e., MultiAdjacentList and KMeans. In the definition section (Section 2.3), we tried to let readers got a first grasp on the concept of dependency probability by explaining it in plain English, "as the chance of input key k transforms to mapped key f(k) after the transform operation", as well as illustrating it with the already mentioned matrix multiplication example. But for more details about the dependency probability calculation, we decide to postpone it to the application section (Section 4) with real application illustrations.

3. Comment by Reviewer 1: "On sec 5.1 the claim that the executor tying implementation does not introduce side effects is not convincing. Whenever the manager executes more than one job at the same time, in particular if their numbers of executors are hashed in a way that brings many conflicts, there would be potentially tasks waiting for other ones to finish, while some other computing nodes remain idle. "

Response: According to this comment, we found that when multiple applications are running, some executors do may have more tasks than the others due to inappropriate hashing. For a concrete example, suppose there are three executors in the Spark cluster, and there are three applications, whose numbers of tasks are 1, 2, 3, respectively, and the task IDs are numbered incrementally from 0, 1, until 2. If we use a mod-by-3 function as the hash function, all of the three applications will compete to tie their task 0 to the executor with hash code 0, but only one application with three tasks will be tied to the executor whose hash code is 2. We address this problem in Section 5.1 in the revised version, and propose to use a hash function that will randomly hash a task to an executor.

4. Comment by Reviewer 2: "However, the comment "Have you utilized the compression file format provided by Spark in your experiment? Can you specify it in your experimental setup part? Can you compare the performance for compressed setting and uncompressed setting?" is not fixed.
This experiment is very important. As you mentioned in the paper, you have observed that the bottleneck is the network shuffle, the compression actually can relieve the load on the network and it is wildly utilized in the current data processing systems like Spark and Hadoop YARN to reduce the disk and network I/O overhead. Besides, recent studies show that the disk and network I/O are not the bottlenecks anymore with the compression techniques (Making Sense of Performance in Data Analytics Frameworks, NSDI 2015). Thus, it is important to show the performance of your system under different file format settings in the paper and what the new observations is by applying your key-aware optimization algorithms. "

Response: In our existing evaluations, data are already compressed in "lz4" in the shuffle network streams. We agree that it is important to mention and discuss this, and compare the effects of CKP with shuffle data compression. We added this comparison in Section 6.6; the finding is that compression does reduce the shuffle size, but CKP can work with compression together to reduce more. We also make note of the potential time overheads of data compression and decompression.

5. Other changes: We have also polished the presentation by fixing some typos or grammar issues.