Note: All the adjustment in the paper in response to the comments are highlighted inside the paper in blue color.

Confluence: Improving Iterative Distributed Operations by

Key-Dependency-Aware Partitioning

**Comment 1:** "Confluence constructs a key dependency graph based on the application semantics to reflect the key dependency relationship of the

datasets of adjacent computing iterations"

also:

"In the key dependency graph proposed in this paper, the

construction of the graph does not use any data partitioning

information, but considers the \*\*application semantics\*\* of the

keys of the data, which depends on the application logic

only.

\*\*  You better define what is "application semantics" earlier.

    Remember I also raised the issue in your thesis review "

"What do you mean “semantics” of the distributed operations? Sorry, I still don't get it.”

Response: I have defined the “application semantics” in the fourth paragraph of the Introduction Section: “The application semantics refers to what new key-value data entries a key-value data entry will generate in a distributed operation, based on the logic of the program.”

But to make it more clear, I now update it to:

“The application semantics

of iterative distributed operations, also called distributed

operation semantics in this paper (Section 3.1), reflects the

logic of the program that decides, 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. The key dependency relationship between diffe-

rent computing iterations are derived from the distributed

operation semantics. It represents the transformation pro-

cess 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 (Section 3.2). It

is special to every iterative distributed operation and is

decided by the logic of the code and the input dataset.”

-----------------------------------------------------------------------------------

**Comment 2:** "In fact, we usually do not know the contents of Ai and

cannot explore the partition scheme for the next iteration

until the program has actually finished the ith 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."

But in your Algorithm 1:

Input : Key Set Ki, containing all the possible keys

(or key patterns) of the output key-value

pairs of any Iteration i;

1. What do you mean "all possible keys" of the output key-value pairs?

2. It seems to me you have to collect ALL the keys from all output key-value pairs.

    Then, reconstruct the Key Dependency Graph. Gathering ALL these information

     from all nodes could be a bottleneck, even you just collect the keys.

3. Why there are "unknown keys" (I guess this is about the “semantics” of the distributed

operations?

Response:

1. “all the possible keys”means all the key values. But I also say that it can be all the possible “key patterns”. When the number of key values is large, the number of key pattern can be very small. I say all the keys here because I do not want to lose the generality of the KDG construction algorithm. I see your wonder. To clarify, I added:

“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.”

I also adjust the definition of the key dependency to include the key pattern.

I think it also answers your second question.

3. 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.

----------------------------------------------------

**Comments 3:** CONFLUENCE

Even though it is hard to find the optimal partition solution

to minimize the overall shuffle size, if the application

semantics of the specific distributed operations (distributed

operation semantics) is already known, the shuffle size can

be reduced to the maximum extent by exploring the key

dependency of different iterations.

Q1: How to obtain "distributed operation semantics"? Reading the Spark code?

\*\* You see all the troubles are from “semantics” of the distributed operations

\*\* You may like to add a special section just to talk about this.

Response: Yes, by reading (usually, when writing) the code.

I add a section (Section 3.1) to explain the semantics issue:

“3.1 Distributed Operation Semantics

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 operation semantics represents such logic of the

generation of key-value pairs from the input key-value pairs

in every iteration of the distributed operation. By saying

that we know the distributed operation semantics for a

specific application, it means that we know what the output

key-value pairs will be for an input key-value pair in a

computing iteration. The programmer is supposed to be well

acquainted with the distributed operation semantics.

We use the key dependency graph (introduced later) to portrait

the distributed operation semantics so that

it is comprehensible to the computer. Every

distributed operation has its own semantics and it relies

on the programmer to understand the distributed operation

semantics and construct the key dependency graph.”

-------------------------------------

**Comments 4:** 4.2 MultiAdjacentList

"However, the iterative operations can be divided 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 Li ) (Li; tails)."

Q1: Who can tell "the iterative operations can be divided into several smaller

iterative operations by dividing the dataset as in Fig. 4(d)" -- Your algorithm or by human?

Response: By human. Not the algorithm. The logic of the algorithm can be unchanged.

Similar problem in 4.1 MovieLensALS

"However, if the iterative distributed operation is regarded

as two individual iterative operations and is divided

as in Fig. 4(b), the pure confluence subgraph can be found"

\*\* Who knows (or can detect "the iterative distributed operation is regarded

as two individual iterative operations and is divided as in Fig. 4(b)"?

Response: The human (the programmer).

Maximizing Shuffle Network Bandwidth Utilization by Application-Level Flow Scheduling

**Comments 1:** \*\* The key work "Application-level" (if important) was not emphasized in Abstract.

Response: I agree that the point of “application-level” should be raised in Abstract. I added:

“Instead of working on the network level that schedules the bandwidth of flows or the routing of data packages, BAShuffler uses the application-level approach that schedules flow source nodes based on the bandwidth allocation estimation without changing the underlying network.”

**Comments 2:** \*\* The "application-level scheduling" was mentioned briefly in Introduction:

"The application-level scheduling approach works by making use of both the shufflerelated information"

  But it is unclear to me what exactly is the application-level approach.

  Please note that the key design of Yarn is to enable multiple applications (MapReduce, Spark, MPI)

  runing on the same cluster at the same time. I am not sure if those shuffle related information

  is about a sigle application (e.g., one MapReduce program), or among all applications (e.g., from Resource

  Manager's view, it knows the status and resource usage for all applicaions running on it -- to me, these are all

  "application-level", too).   So, you better make it clearer.

Response: In the paper, the application level is relative to the network layers in the OSI model, but not the MapReduce applications.

To make it clearer, I explain in Introduction:

“. A central scheduler in the distributed re-

source management platform, e.g., the Resource Manager of

YARN, makes use of the shuffle information and schedules

the flow source nodes for all the MapReduce-like programs

running on this platform.”

And also in Section 2.2 that discuss application-level scheduling:

“``Application-level`` is the term relative to the network layers

in the OSI model, but is not limited to the distributed

application (e.g., MapReduce programs). The distributed

resource management platforms like Mesos and Hadoop

is also working at the application level.”

\*\* Concern: if we have multiple MapReduce programs running at the same time, is your "application-level scheduling"

    applied on individual program separately? Then, without the global information, can you achieve optimal results

    for all 3 programs? In this regard, it seems the network-level approach is better (enable global optimization).

   (I will keep reading the rest of contents and see what you are actually doing -- Later I found your scheduler is running within

Resource Manager. This seems reasonable.).

Response: Yes, the scheduling is in the resource manager with the global information. All programs need to report the status and ask for the decision from the resource manager.

**Comments 3:** \*\* "network is a heterogeneous": usually we won't consider this is a mainstream problem as most clusters are made

    homogeneously (just my own view). We would focus more on the dynamic network traffic in real time computing

    (that is , how to do proper shuffle given the available network bandwidth at run time?)

I feel you are just creating problems to solve.

Response: One of the primary goals of MapReduce is to make use of the off-the-shelf (or commodity) machines to do distributed processing [1]. An important reason that Hadoop becomes popular is that IT companies can setup an available distributed cluster very fast with their machines at hand. Heterogeneous network is unavoidable in this scenario.

I add in Introduction:

“For example, Hadoop is

designed as the general distributed processing platform for

clusters comprising off-the-shelf machines. The physical

Hadoop clusters could be fit with a variety of servers and

network devices of different configurations”

Anyway, the scheduling algorithm itself in the paper did not focus on whether the network is heterogenous or not. It focused on the scheduling based on the estimation of the current network status. It happens to be suppose works even better in the heterogenous network.

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

**Comments 4:** \*\* "The simulation results show that comparing to the random-source-selection method adopted by

     YARN, PGSS can improve the cluster network bandwidth utilization especially in a heterogeneous .."

   ---> Just comparing with a dump solution (random-source-selection) can not show your work is better

     than those state-of-the-art solutions ===> If  I am a reviewer, I will use this reason to kill your paper ^\_^

Response: Unfortunately, the random-source-selection solution is the only existing (as I know) application-level solution that is widely in the popular distributed platforms, including Hadoop and Spark. I could invent other source selection algorithms (actually GSS is one), but I could not exhaust them. I clarify this in Section 2.1.

It would be good to compare PGSS with other shuffle scheduling algorithms at the network level. But I am afraid that it would be hard to achieve, with the difficulty to get the source code of these algorithms (although we can re-implement them based on their paper) and the lack of configurable switches. One of the design goal of BAShuffler is to keep the network layer unchanged. It enables the cluster manager to own a faster platform for shuffle with the ordinary operating system and network devices.

I emphasize this again in Section 2.2:

“The benefits of application-level shuffle scheduling are:

1) Besides bandwidth capacities of the links, it can make

use of the dynamic application-level shuffle information,

including the pattern of the arriving time of the flows and

their sources and destinations; 2) The distributed platforms

can work on the ordinary operation systems and network

devices.

**Comments 5:** \*\* "Minimizing the completion time of the coflow is NOT the

same thing as minimizing the shuffle completion time."

Be reminded, you have not define "shuffle completion time" earlier.

The readers may not get what you mean, and why this was emphasized.

Response: What I want to emphasize is that the coflow is not the exact model for the shuffle operation with the fact that not all the shuffle fetch flows are known in batch to become a coflow.

**Comments 6:** \*\* "To obtain the optimal scheduling solution that minimizes the shuffle completion time,"

  I believe we have raised the question during your oral exam: if you just optimize shuffle completion time

  will you sacrifice the total execution time. I believe the key objective is to minimize the total execution time, right?

  Please add your justification ! Is it only valid if the application is

  Again, "shuffle completion time" was not defined. So I won't be able to

  give further comments. Surely, if you only refer to "shuffle-heavy" MapReduce tasks,

  reducing shuffle time can help to improve the total time (since the shuffle time dominate the total execution time).

Response: Yes, to minimize the total execution time would be the perfect goal. It will face lots of other issues to solve, including the resource allocation problem and task assignment problem for the other map tasks, reduce tasks and any other task that is involved. It would be a very complex target, and it involves the collaboration of all component in the system. Therefore, no distributed system claim it to be optimal, though the algorithms focuses on a specific smaller target can. In this paper, these issues are considered as controlled variables, and I focus on the shuffle part.

Luckily, (if the scheduling overhead can be ignored,) the methods used in BAShuffler do not occupy any more configurable compute resources (CPU and memory) nor change the task assignment mechanism. It is not supposed to negatively impact the total execution time. The only affect may be faster shuffle. How much it can improve the total execution time depends on how the job is affected by faster shuffle. Of course when shuffle-heavy jobs can enjoy more benefit, shuffle-light jobs will not sacrifice its total execution time.

To justify this, instead of discussing this issue in the application level scheduling section, I prefer to add in the design consideration section 3.1.1:

“”

**Comments 7:** 3.1.1 Design Considerations

\*\* What are the actual problems behind the scene?

I found you keep discussing the number of fetchers and the communication pattern

of the fetch flows. Are there other factors, like (1) nodes are busy with the map tasks, thus can not help to distribute data;

 (2) reducers were activated too early thus occupy resources but not receiving data?

Response: Yes, there are many other factors that can influence the shuffle. The above two factors are the ones that concerns the starving of the computational resources for the shuffle tasks. This is supposed to be avoided in the distributed platforms with a good resource isolation mechanism, although the current system has not done very well in this aspect. To recognize these factors, I add:

“For example, the CPU time slices for source nodes to send the map outputs via the fetch flow and

the straggler of a map task that prevents the shuffle operation having all the map outputs ready.

These factors are related to the dynamic environment of the distributed management platform.

Some of them can be avoided by a better resource isolation mechanism.

For instance, a good resource isolation mechanism should make sure the CPU time slices guaranteed for

sending the flow data will be stolen by other compute-intensive jobs.

BAShuffler does not consider these factors.”

Anyway, I raise the two factors because it directly relates to our algorithm:

1. The number of fetchers: it discusses the rationality of fixing the number fetchers in the system, but not make it be a input variable for the scheduling algorithm.
2. The communication pattern: This is what BAShuffler want to schedule.

----------------------------

**Comment 8:** Minor:

Gideon-II cluster [33] having a heterogeneous network ??

Response: I change it to “Gideon-II cluster [33] in the heterogeneous network configuration.”

Gideon II cluster originally does not have a heterogeneous network. As I want to see BAShuffler’s performance in the heterogenous one, it is configured by “tc”.