SDP Memo 059: Modeling and Evaluating the IO of MID1 ICAL Pipeline on Spark

Document number……………………………………………………………SDP Memo 059

Document Type……………………………………………………………………MEMO

Revision…………………………………………………………………………..DRAFT

Author……………………………………………………………………….Qiuhong Li, Wei Wang and Yuan Luo

Release Date…………………………………………………2018-12-27

Document Classification………………………………………………….. Unrestricted

<table>
<thead>
<tr>
<th>Lead Author</th>
<th>Designation</th>
<th>Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qiuhong Li</td>
<td></td>
<td>Fudan University</td>
</tr>
</tbody>
</table>

Qiu
hong Li

Document No: SDP Memo 059

Revision: DRAFT

Release Date: 2018-12-27
SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Table of Contents

SDP Memo Disclaimer ........................................................................................................................................ 2
Table of Contents ............................................................................................................................................... 2
1. Introduction ................................................................................................................................................... 4
2. Modeling MID1 ICAL pipeline on Spark ....................................................................................................... 7
   2.1 Cost model on Spark .................................................................................................................................... 7
      2.1.1 Memory cost ......................................................................................................................................... 7
      2.1.2 Shuffle cost .......................................................................................................................................... 7
      2.1.3 Spark task overheads of MID1 ICAL pipeline .................................................................................... 8
   2.2 Current data model of MID1 ICAL pipeline .............................................................................................. 9
   2.3 A simplified Data Model of MID1 ICAL Pipeline ..................................................................................... 11
3 Analysis and Comparisons of Several Implementations of MID1 ICAL Pipeline ............................................. 12
   3.1 Overview .................................................................................................................................................... 12
   3.2 The auto-generated version of Spark ....................................................................................................... 13
   3.2 Partitioning version and Partitioning + Alluxio version ........................................................................... 15
   3.3 Analysis and evaluation of several key stages .......................................................................................... 15
   3.4 Differences between Spark version and StarPU version ........................................................................ 22
4. Evaluation ..................................................................................................................................................... 23
   4.1 Comparisons of different implementations of MID1 ICAL IO ................................................................. 23
   4.2 Comparisons between Java (Scala) API and Python API ....................................................................... 24
   4.3 Wrapping a java implementation of MID1 ICAL IO program on Spark .............................................. 25
5 ICAL pipeline of ARL on Spark ....................................................................................................................... 27
6 Conclusion and future work ............................................................................................................................. 27

Document No: SDP Memo 059
Revision: DRAFT
Release Date: 2018-12-27
1. Introduction

This memo aims to model the IO of running SDP graphs on Spark. We start from a baseline program which is generated for MID1 ICAL pipeline on Spark. MID1 ICAL is an astronomical calibration pipeline which is illustrated in Fig. 1 [1]. In Fig.1, tasks are organized by the logic tasks. In this memo, we construct a cost model and evaluate several different implementations of MID1 ICAL Pipeline on Spark [2]. According to the proposed cost model, we redesign the data model and try to find an efficient implementation. However, the logic tasks are different from the tasks executed by an execution framework (EF for short). An EF task might be a set of logic tasks and it is possible that a logic task is implemented by a set of EF tasks concerning of the computing and IO usage. However, in Fig.1, there exist huge IO usages between logic tasks. Some of them do not cause IO actually. For example, the execution of several logic tasks in a process avoid IO usage by utilizing memory as the data storage. As Fig.2 (a) illustrates, an EF task contains three logic tasks, repre_ifft, degkerupd_deg and pharopre_dft_sumvis. In this way, the temporary data between tasks is handled on the fly. While in Fig.2 (b), Task 1, Task 2 and Task 3 are EF tasks which have one logic task respectively. Therefore inter-process communication is necessary. According to our cluster, the processing speed of inter-process is about 3~4 GB/s for a single node. The speed is lower if multiple nodes concerned. While memory speed in a process can reach about 300~400 GB/s. Disk I/O is about 500 MB/s by average. The network communication speed is about 1GB/s. Because of the huge IO gap, the best way for SDP pipelines is to utilize memory computing in a process as possible as we can.

We start evaluating SDP EFs with Spark and plan to get an optimized version of the baseline program. The reason we consider Spark first because it supports distributed in-memory iterative computing, which is important for large iterative SDP pipelines.

There exist several main problems using Spark as the execution framework for MID1 ICAL pipeline. For this pipeline, there are several join operations for several data sources. First, unfortunately, Spark is inefficient in handling data join because the Shuffle operation is very expensive. Second, Spark is implemented in Scala language. The most efficient way is to use JVM API (Scala or Java) as the pipeline language. While many astronomical packages are implemented in C or Python. Spark supports C or Python by utilizing extra process to communicate with the JVM process, which causes performance loss.
Notice that for MID1 ICAL pipeline, most of the reduce operations can be done in an incremental way. Therefore the reduce operation can be done without waiting for the arrival of all of the data items.
Fig. 1. Data flow of MID1 ICAL Pipeline (Referenced from [1])

(a) Temporary data in memory (b) Temporary data needs inter-process communication

Fig. 2. Possible relationships between an EF task and logic tasks

However, the current data model of MID1 ICAL is suitable for a central processing station. Shared memory is utilized to parallelize threads on a machine without considering the data distribution. But it is unsuitable for distributed processing where shared memory cannot be utilized in a simple way. While the network IO for a distributed cluster is much more expensive compared with shared memory in a machine. Furthermore, the current data model exists huge data movements which should be decreased to improve the IO performance. For distributed computing, data locality is quite important. Spark has three locality types, Process-local, Node-local and Any (Any means there exists inter-nodes communications). The StarPU implementation of MID1 ICAL only has one process, it is typical process local. While for the Spark implementation of MID1 ICAL, different stages are executed by different processes. The operations on an RDD may be process-local. The communications between RDDs are implemented by the communications between Spark executor and Spark worker. In a word, the huge gap between the StarPU implementation and the Spark implementation is caused by different execution model. StarPU is a thread execution model, while Spark is a combination of processes and threads. By invoking a java implementation of MID1 ICAL IO in a Spark task, we get almost the same running time as StarPU. However, putting all things in a process sometimes cannot work because of the resource limits. Furthermore, the StarPU implementation by putting all things in a process is not easy to scale out.
We focus on the following issues:

1: The cost model of MID1 ICAL pipeline on Spark
2: Designing RDDs in an efficient way
3: An improved data model considering the data locality for Spark

2. Modeling MID1 ICAL pipeline on Spark

2.1 Cost model on Spark

The cost for the I/O of the MID1 ICAL pipeline on Spark includes task overheads, memory cost, disk cost, network cost, serialization and deserialization cost. We use the time as the measure for the cost analysis. According to our experimental results, the overhead for a Spark task is less than 3 milliseconds. The memory speed is about 300~400 GB in a process. It is about 3~4 GB inter-processes. The disk speed is about 500 MB/s. The network speed is about 1 GB/s. The number of Spark tasks are related with the number of the RDD partitions. A partition is executed by a Spark task.

We propose a cost model as the following:

\[ \text{COST}_{\text{IO}} = \text{MEM}_{\text{cost}} + \text{Shuffle}_{\text{cost}} + \text{Task}_{\text{overhead}} \]  

(1)

\[ \text{MEM}_{\text{cost}} = \text{RDD}_{\text{memory}} + \text{Broadcast}_{\text{memory}} + \text{Execution}_{\text{memory}} \]  

(2)

\[ \text{Shuffle}_{\text{cost}} = \text{Sort}_{\text{cost}} + \text{DISK}_{\text{cost}} + \text{SER}_{\text{cost}} + \text{DESER}_{\text{cost}} + \text{NET}_{\text{cost}} \]  

(3)

For Spark, the shuffle cost is the most expensive. In our cluster, the processing speed for shuffle is about 50 MB/s.

Because the broadcast variables need to be copied from the Spark Worker memory to the Spark Driver memory and are distributed to each computing node, given broadcast variable broad, we need \((m+1)\text{sizeof(broad)}\) memory space for a Spark cluster with \(m\) computing nodes.

By now, we use a simple way to compute the cost:

\[ \text{COST}_{\text{IO}} = \frac{\text{Data}_{\text{rdd}}}{\text{Speed}_{\text{rdd}}} + \frac{\text{Data}_{\text{shuffle}}}{\text{Speed}_{\text{shuffle}}} + \frac{\text{Data}_{\text{broadcast}}}{\text{Speed}_{\text{broadcast}}} + \frac{\text{Data}_{\text{memory}}}{\text{Speed}_{\text{memory}}} + \text{Task}_{\text{num}} \times \text{Task}_{\text{overhead}} \]  

(4)

2.1.1 Memory cost

Memory cost includes RDD cost, broadcast cost and the cost for data processing in memory.

2.1.2 Shuffle cost

The shuffle phase is to sort the results of mappers and transfer them to the reducers. Because the reducers and mappers may not exist on the same computing nodes. Thus the
serialization and deserialization for shuffle data are necessary. Compared with the data processing in the main memory, the shuffle data are expensive to be processed.

The “cogroup”, “groupByKey” and “flatMap” are three expensive Spark transformations which cause shuffle:

1) “cogroup”
When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples.

2) “groupByKey”
When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.

3) “flatMap”
Spark flatMap is a function which expresses a one-to-many transformation. It transforms each element to 0 or more elements.

2.1.3 Spark task overheads of MID1 ICAL pipeline
Spark task overheads per task is less than 3ms. We reduce the data scale to a very small value and treat the execution time as the task overheads. We use 3ms as the average Spark overheads for a task (not include the data processing time). For the auto-generated version, 12964 tasks are launched. The time for extra task overheads is about 12964*3 ms =38.9 seconds. The number of Spark tasks is related with the number of the partitions of RDDs. Thus, coarse granularity of RDD partitions can decrease the extra Spark task overheads. The size of the RDDs and the size of Shuffles are both related with the original data size. The operations in an RDD take place in the Spark worker process by a thread. The processing speed during a process can reach 300 ~ 400 GB/s. The operations among different RDDs need multiple processes, thus inter-process communication and network communication are needed. The processing speed for inter-process in a machine is about 3 ~ 4GB/s (Without shuffle). The processing speed for shuffle is quite slow, less than 50MB/s for our cluster because of sort operations and disk IOs. Furthermore, the flatMap operation causes huge shuffle amount for the auto-generated version. More than 60 GB shuffles are generated. The processing time is more than 60000/50=1200 seconds.
2.2 Current data model of MID1 ICAL pipeline

![Current Data Model of MID1 ICAL Pipeline](image)

We use the data model in Fig. 3 as a baseline to start the modeling work. This data model is extracted from the parameter model. We assume that we have enough memory and computing resources. Different colored nodes represent different logic tasks. All of the edges between nodes represent data communication. While the edges with read lines represent network communications among cluster nodes.

As illustrated in Fig. 3, six join operations of datasets are needed, denoted as $comm1$~$comm6$. To my understanding, we do not need to wait all of the data available to start the processing for $comm1$, $comm3$, $comm4$, $comm5$ and $comm6$. Each two data items can be processed and reduced.

1: $comm1$
In *Predict* phase, the local sky model is divided into facets, each model facet is predicted separately. In the end, the results are collected.

2: *comm2*

Both the observed visibilities and the predicted visibilities are put together to calibrate. The calibration process is independent for each frequency. The results of calibration for all the frequencies are needed to correct the observed visibilities.

3: *comm3*

Visibilities of the specified neighboring frequencies are collected.

4: *comm4*

For each facet, the visibilities of four polarisations are collected to identify component.

5: *comm5*

The visibilities of four polarisations for each facet are collected to subtract image components.

6: *comm6*

The results of “identify component” for all of the 81 facets are collected.

The RDDs in both the auto-generated version and partitioning version are showed in Table 1.

<table>
<thead>
<tr>
<th>RDD name</th>
<th>input</th>
<th>kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>reppre</td>
<td>facet of local sky model</td>
<td></td>
</tr>
<tr>
<td>degrid</td>
<td>RDD reppre</td>
<td></td>
</tr>
<tr>
<td>pharotpre_dft</td>
<td>RDD degrid</td>
<td></td>
</tr>
<tr>
<td>visibility</td>
<td>visibility in buffer</td>
<td></td>
</tr>
<tr>
<td>timeslot</td>
<td>RDD pharotpre_dft, RDD visibility</td>
<td>data reduction by time</td>
</tr>
<tr>
<td>solve</td>
<td>RDD timeslot</td>
<td></td>
</tr>
<tr>
<td>correct</td>
<td>RDD predict_observed, solve in Alluxio</td>
<td></td>
</tr>
<tr>
<td>grikerupd_rep</td>
<td>RDD correct</td>
<td></td>
</tr>
</tbody>
</table>
Table 1 RDD Design

<table>
<thead>
<tr>
<th>sum_facet</th>
<th>RDD grierupd_rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>identify_component</td>
<td>RDD sum_facet, RDD subtract_image_component</td>
</tr>
<tr>
<td>subtract_image_component</td>
<td>RDD sum_facet, RDD identify_component</td>
</tr>
<tr>
<td>source_find</td>
<td>RDD identify_component, local sky model</td>
</tr>
</tbody>
</table>

2.3 A simplified Data Model of MID1 ICAL Pipeline

According to the proposed cost model, there are several principles to design a data model of MID1 ICAL pipeline on Spark.

1: The data model should maximize data locality to void inter-process and inter-nodes communications.

For MID1 ICAL pipeline, the visibilities of the neighboring twenty frequencies should be put together. To avoid a huge size, the number of time slot should be decreased. We decrease the number of time slots from 120 to 10. That is, for a processing unit, we increase the number of frequencies and decrease the number of time slots to balance the unit size. By this way, comm3 in Fig.4 can be avoided.

To improve data locality, putting the predicted visibilities and the observed visibilities side by side can avoid huge inter-process communications. The communications between RDD visibility and RDD pharot_dft can be avoided.

Comm4 and comm5 in Fig.4 are caused by collecting the visibilities of the four polarizations. By putting them together can avoid these communication costs.

2: The Spark tasks should be process-local ones as possible as we can. That means, we should decrease the communications between RDDs. Inside an RDD, the tasks are more possible to be process-local ones.
3: The number of Spark tasks should be limited. Besides the task overheads, the management of Spark tasks needs more memory resource and computing resource.

A simplified data model, denoted as new model is illustrated in Fig.4.

3 Analysis and Comparisons of Several Implementations of MID1 ICAL Pipeline

3.1 Overview

According to the codes generated from the parameter model generator [4], there are visibility data of 800 bands of frequency. Each 20 bands can be processed independently.
Thus there are 40 groups of the visibility data. The parallel granularity is flexible by the combination of frequency, time and facet.

We introduce auto-generated version, partitioning version, partitioning + Alluxio version and StarPU version and analyze the differences.

If the six data joins in Fig.4 are implemented by Spark shuffle, the experimental results show the performance is quite poor. However, all of the six join operations can be implemented by the combination of Spark partitioning and Spark broadcast, which cause little shuffle cost. Another method is to use Alluxio to solve the data join problem. However, the current data model has huge communication cost. Finding a simplified data model for Spark is quite necessary. Notice that the comm3 is the most expensive. We can solve it by putting the visibilities of neighboring twenty frequencies together. At the same time, we reduce the number of time slots in visibility buffer to avoid a huge size. A simplified data model is presented in Fig. 5. By putting the predicted visibilities and the observed visibility together, we can save the cost for comm3.

3.2 The auto-generated version of Spark

The auto-generated version uses flatMap to copy data and treat each data block identified by a tuple as a partition. We treat it as a baseline program.

The auto-generated version has the following features:

1: There exists data co-location from both three data sources and two data sources via key exploration. (This feature causes expensive “cogroup” operation of Spark)

2: There are too many stages, which causes huge temporary data contained in RDDs. RDDs need extra serialization and deserialization cost. That is, data storing in RDDs is much more expensive than directly in the main memory. Furthermore, the data exchange between RDDs need inter-process communication.

3: There exist serious data copy problem.

   In auto-generated version, many “flatMap” operations are used to copy data.

To evaluate whether Spark can satisfy the requirements of SDP pipelines, we deploy several Spark clusters in different environments and run the baseline program (denoted as auto-generated version) on them. The baseline program is written in Scala, which generates about 417 G data to mock the data flows for MID1 ICAL pipeline.
By collecting the statistical info during the execution, we conclude the performance bottlenecks listed as below:

1: Too long resilience links

2: Unnecessary join costs for two or three massive RDDs

3: Unnecessary data transfer caused by not considering the data locality

To optimize the expensive “cogroup” operations of the baseline program, we replace the “cogroup” by broadcasting the smaller RDD. By this method, we only need to iterate the larger RDD and avoid the expensive join operation.

However, this kind of optimization is not suitable for the “cogroup” operation for two or more massive RDDs. To solve this problem, we use Alluxio to serve as a distributed cache to avoid broadcasting a large RDD and the experimental results are quite good. Alluxio is a product from AMP Lab, which is the birthplace of Spark. Alluxio can provide data sharing across different jobs and different systems with in-memory speed.
3.2 Partitioning version and Partitioning + Alluxio version

Partitioning version avoids copy data by using Spark `collect` and Spark `broadcast` to communicate among Spark RDDs. Spark executor gets the data from Spark workers by `collect` operation and broadcasts the data to the workers which need the data. The collect operations for the six joins in Fig.4 are the bottlenecks of the partitioning version. To improve the communication performance further, partitioning + Alluxio version uses Alluxio as a communication tool between RDDs.

It is quite simple to install and use Alluxio to store the data of the pipeline. The following scripts is used to create the data storage for tasks. For MID1 ICAL pipeline, almost all of the combinations of data items can be determined beforehand. Thus it is quite suitable to use Alluxio to solve the join problem.

```bash
sudo ./alluxio fs rm -R /pharotpre_dft_sumvis
sudo ./alluxio fs mkdir /pharotpre_dft_sumvis
sudo ./alluxio fs chmod 777 /pharotpre_dft_sumvis
sudo ./alluxio fs rm -R /cor_subvis_flag
sudo ./alluxio fs mkdir /cor_subvis_flag
sudo ./alluxio fs chmod 777 /cor_subvis_flag
sudo ./alluxio fs rm -R /visibility_buffer
sudo ./alluxio fs mkdir /visibility_buffer
sudo ./alluxio fs chmod 777 /visibility_buffer
sudo ./alluxio fs rm -R /solve
sudo ./alluxio fs mkdir /solve
sudo ./alluxio fs chmod 777 /solve
sudo ./alluxio fs rm -R /reppre_ifft
sudo ./alluxio fs mkdir /reppre_ifft
sudo ./alluxio fs chmod 777 /reppre_ifft
```

3.3 Analysis and evaluation of several key stages

We use scale=1/10 to compare the different implementations of Spark. We evaluate the IO of MID1 ICAL for twenty neighboring frequency bands of visibilities. We analyze and evaluate several key stages.

**Reppre_ifft & Degrid**
Different implementations have different mapping strategies. We compare auto-partitioning version, partitioning version, partitioning+Alluxio version and new model version for these two stages. We compare the number of Spark tasks, the amount of Shuffle and the execution time respectively.

Each partition of an RDD is processed by a Spark task. The auto-partitioning version defines a partition with a data block identified by a six-tuple (beam, major_loop, frequency, time, facet, polarisation). For \textit{reppre\_ifft} stage, beam : 0, major_loop : 0, frequency : 1\textasciitilde5, time : 0, facet : 1 \textasciitilde 81, polarization: 1\textasciitilde4. For \textit{degrid} stage, beam : 0, major_loop : 0, frequency : 1\textasciitilde20, time : 0, facet : 1 \textasciitilde 81, polarization: 1\textasciitilde4. Thus for the auto-generated version, there are 1620 Spark tasks for \textit{reppre\_ifft} stage and 6480 Spark tasks for \textit{degrid} stage. For partitioning version, if the default parallelism is set to 20, there are 20 tasks for \textit{reppre\_ifft} stage and \textit{degrid} stage respectively. The new model version merges \textit{reppre\_ifft} stage and \textit{degrid} stage. Therefore there are 20 tasks for these two stages. The partitioning+Alluxio version is almost the same as partitioning version except that it adopts Alluxio to save the outputs for both of the stages.

<table>
<thead>
<tr>
<th></th>
<th>Auto-generated</th>
<th>Partitioning</th>
<th>Partitioning+Alluxio</th>
<th>New model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task number</td>
<td>8100</td>
<td>40</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>Shuffle amount(MB)</td>
<td>75</td>
<td>3.9</td>
<td>0</td>
<td>2.6</td>
</tr>
<tr>
<td>Execution time(second)</td>
<td>89</td>
<td>11</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 2  Comparisons of different implementations of \textit{reppre & degrid}
The input of this stage is the local sky model. The results of `reppre_ifft` are kept in memory. The RDD `degrid` is organized as `<key,value>` pairs. The key is a six tuple of (beam, major_loop, frequency, time, facet, polarization), where beam=0, major_loop: 1~9, frequency: 1~800, time: 1~36, facet: 1~81, polarization: 1~4. Each value is of 401.4 MB. For 9 iterations, each iteration has an RDD `degrid` of size $800 \times 81 \times 4 \times 401.4 \times 36 = 104042$ TB.

If a collection of tasks are executed within a process, the cost of communication can be saved. As Fig.2 shows, in the first three stages of MID1 ICAL pipeline, the data dependency is represented as edges between the nodes, which represent tasks of different stages. A solid purple circle represents a task of the `reppre` stage. A solid green circle represents a task of `degrid` stage. A solid yellow circle represents a task of `pharopre_dft` stage. For short, we denote a task of `reppre` stage as a purple task, a task of `reppre` stage as a green task and a task of `pharopre_dft` stage as an orange task. If we have enough memory for a single node, the temporary results between the tasks in a process can be in memory directly without the communication cost.

In the new model version, we use the following scripts to merge the two stages together.

``` scala
sc.parallelize(initset).flatMap(ix=>reppre_ifft_degrid_kernel(ix,broads_input_telescope_data,broadcast_lsm))
```

Here `initset` is a six-tuple of (beam, major_loop, frequency, time, facet, polarization). Local sky model is shared by Spark broadcast variable. The results of `reppre_ifft` are kept in memory and the output of `degrid` is stored in RDD `degrid`. 

---

Fig. 6 Data dependency of `reppre` & `degrid` (for neighboring 20 frequencies)
For the four different implementations, Table 2 illustrates the differences. The auto-generated version launches too many tasks and has shuffles.

**Pharotpre\_dft\_sumvis**

In this stage, the predicted visibilities of facets belong to the same frequencies are collected and processed.

The auto-generated version makes 324 copies of RDD *degrid* by Spark *flatMap* operation. This is the reason of the inefficiency.

The partitioning version uses Spark *partitionBy* to reorganize RDD *degrid* and broadcasts local sky model, which has a small size.

The partitioning+Alluxio uses Alluxio to store the outputs.

For the new model version, the input of this stage is RDD *degrid*. We use Spark *partitionBy* operation to collect the facets of each 20 neighboring frequency bands and 10 time slots. As the following scripts illustrates, in each partition, both *pharotpre\_dft\_sumvis* kernel and visibility kernel are executed. The output is RDD *Pharotpre\_dft\_sumvis* including both the predicted visibilities and the observed visibilities. By this way, the predicted visibilities and the observed visibilities are put side by side and avoid huge communication cost putting them together. To avoid a huge size of each item of RDD *Pharotpre\_dft\_sumvis*, we decrease the number of time slots from 120 to 10. Thus each item of RDD *Pharotpre\_dft\_sumvis* including the visibilities of twenty frequency bands and 10 time slots.

RDD *Pharotpre\_dft\_sumvis* is generated by the following scripts.

```scala
reppre\_ifft\_degerudp\_deg.partitionBy (new SDPPartitioner\_pharo(12)).mapPartitions(pharotpre\_dft\_sumvis\_visibility\_kernel)
```

class SDPPartitioner\_pharo\_newmodel(numParts: Int) extends Partitioner {
  override def numPartitions: Int = numParts
  override def getPartition(key: Any): Int = {
    key.toString.split(',')(3).toInt / 10
  }
}
```

<table>
<thead>
<tr>
<th></th>
<th>Auto-generated</th>
<th>Partitioning</th>
<th>Partitioning+Alluxio</th>
<th>New model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task number</td>
<td>20</td>
<td>20</td>
<td>40</td>
<td>12</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Shuffle amount (MB)</th>
<th>511.5</th>
<th>1.9</th>
<th>0</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time (second)</td>
<td>144</td>
<td>18</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3 Comparisons of different implementations of *pharotpre_dft_sumvis*

**Timeslots**

The visibilities including both the predicted and the observed of each twenty neighboring frequencies for a time slot are averaged in this stage.

![Diagram of timeslots](image)

*Fig. 8 timeslots*

**Correct**

The input of this stage is *RDD pharotpre_dft_sumvis, RDD visibility* and *RDD solve*. Because the *RDD solve* is needed by all of the correct tasks, it can be shared by a Spark broadcast variable. Fig.9 illustrates the data dependency of this stage.
The auto-generated version makes 324 copies of RDD `cor_subvis_flag` using `flatMap` operation. The partitioning version uses `collect` and `broadcast` to avoid data copies. According to Formula (4), the partitioning version has broadcast cost, which is inter-process communication between Spark driver and Spark worker. The partitioning + Alluxio version stores the data treating a facet as a unit and avoids huge data movements. The new model version does the data reduction first. Fig.11 illustrates the processing.

<table>
<thead>
<tr>
<th></th>
<th>Auto-generated</th>
<th>Partitioning</th>
<th>Partitioning+Alluxio</th>
<th>New model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task number</td>
<td>20</td>
<td>20</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>Shuffle amount(MB)</td>
<td>402.9</td>
<td>1.9</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Execution time(second)</td>
<td>228</td>
<td>18</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4 Comparisons of different implementations of Correct

Grierupd_rep

The data dependency is illustrated in Fig.10.
![Diagram](image)

**Fig. 10. Grikerud_rep**

<table>
<thead>
<tr>
<th></th>
<th>Auto-generated</th>
<th>Partitioning</th>
<th>Partitioning+Alluxio</th>
<th>New model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task number</td>
<td>20</td>
<td>20</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>Shuffle amount (MB)</td>
<td>402.5</td>
<td>2.0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RDD Input (MB)</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>RDD output (MB)</td>
<td>324000+15811</td>
<td>15811</td>
<td>15811</td>
<td>15811</td>
</tr>
<tr>
<td>Execution time (sec)</td>
<td>96</td>
<td>17</td>
<td>16</td>
<td>13</td>
</tr>
</tbody>
</table>

**Table 5** Comparisons of different implementations of *Grikerud_rep*
3.4 Differences between Spark version and StarPU version

For the temporary data generated during the execution of the pipeline, different data storage methods lead to different costs. The most efficient way is to store data in memory directly. The implementation of StarPU version adopts this way. Almost all of the stages are executed in a process with multiple threads. But in this way the execution almost has no fault tolerance. There are many temporal data for SDP pipelines. The data in Spark RDD needs the serialization and deserialization and other costs concerning of the lineage. By now, the current cost model for the SDP pipelines does not consider the cost caused by the fault tolerance. For distributed computing, it is necessary to consider the cost.
4. Evaluation

4.1 Comparisons of different implementations of MID1 ICAL IO

We use a cluster of three nodes to perform the experiment to compare the auto-generated version, the partitioning version and partitioning + Alluxio version. Each node has 64 GB memory and we set the data scale with 1/10. The results are illustrated in Table 6. We vary the number of computing nodes with 1, 2 and 3. As explained before, the auto-generated version is the slowest because of the data copies and huge amount of shuffle. By now, for partitioning version, the bottleneck exists in five collection operations, which are caused by the data movements between the Spark worker and the Spark executor.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Auto-generated (minutes)</th>
<th>Partitioning (minutes)</th>
<th>Partitioning+Alluxio (minutes)</th>
<th>New model (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35</td>
<td>7.4</td>
<td>1.3</td>
<td>1.6</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>4.0</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>3.8</td>
<td>2.1</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 6  Comparisons of different implementations of MID1 ICAL (data scale=1/10)

Fig.10  Execution statistics of partitioning version
4.2 Comparisons between Java (Scala) API and Python API

We rewrite the partitioning version into the Python version and compare the performance. With scale=1/1000, the partitioning version needs 18 seconds while the Python version needs 27 seconds. With scale=1/100, the Python version fails because the system is out of memory. The Python version needs more memory because Spark workers needs to copy the data to the Python workers. Spark supports Python by RPC mechanism. The codes are executed in JVM actually, which is illustrated in Fig.12. Notice that RPC communications occur inter-process and network communication cost, which is more expensive than that in memory.
4.3 Wrapping a java implementation of MID1 ICAL IO program on Spark

Notice that process-local tasks are the most efficient for almost all of the execution frameworks. We just wrap a java implementation of MID1 ICAL IO program into a Spark task. In this way, it is similar with the implementation of StarPU, which is only one process concerned.

Spark’s “map” is based on Spark’s “RDD”, which is a distributed data set across many computing nodes. While the buffer of StarPU is for just one computing node. These two concepts do not belong to a same level. However, StarPU does not provide built-in communication among computing nodes. You need to use MPI to implement the communication yourself. Concerning of the parallel executions of a computing node, a buffer can be used in a map inside. That means the codes for a map could include multiple processes to share a buffer too. Table 4 illustrates the experimental results. When the parallelism is set to 1, the overall time as a Spark task is about 1.47 seconds. While the time is about 3 seconds running as an independent task. When the parallelism is set to 5, the overall time as 5 Spark tasks are 9.7 seconds. While the serial execution time is about 15 seconds. The scalability
needs to be explored further. The system resources and parameter adjustments affect the results.

<table>
<thead>
<tr>
<th>Number of parallel tasks</th>
<th>Parallelism</th>
<th>Running time as a task of Spark (seconds)</th>
<th>Running time of serial implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4.47</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>9.7</td>
<td>15</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>39.3</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 7 Wrapping a java implementation in Spark tasks (data scale=1/40)

<table>
<thead>
<tr>
<th>Number of parallel tasks</th>
<th>Parallelism</th>
<th>Running time as a task of Spark (seconds)</th>
<th>Running time of serial implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4.47</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9.7</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 8 Wrapping a java implementation in Spark tasks (data scale=1)

To evaluate the new model version for larger data scales, we use a machine with 1.5 TB memory, 80 CPU cores of 2.2 GHZ. We compare the new model version and the Java version of one process. The results are illustrated in Table 8.

When data scale is set to 1/10 (1/3000 for the SKA full scale), the new model version takes 11 seconds and the Java version takes 7 seconds. With the increase of data scale, both versions take more time. According to the cost model of Formula 4, the time includes three parts, the RDD usage, the Shuffle usage, the memory-execution time and the overheads of Spark tasks. The memory-execution time is almost the same of the Java version. For data scale is set to 1, there are data of 268.68 GB generated. The Java version just affects the execution memory cost. According to the cost model represented by Formula 4, the RDD cost and the broadcast cost can explain the gap between The Spark implementations and the Java version.
### Table 9  Running time of a new data model

<table>
<thead>
<tr>
<th>Data Scale</th>
<th>New model (second)</th>
<th>Java version (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/10</td>
<td>26</td>
<td>7</td>
</tr>
<tr>
<td>1/5</td>
<td>40</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>113</td>
<td>11</td>
</tr>
</tbody>
</table>

5 ICAL pipeline of ARL on Spark

We implement a version of ICAL in ARL on Spark and treat it as a baseline. The source codes are available in [5].

```
spark-submit --name dask_spark --properties-file spark-arl.conf --files
sc512,LOWBD2.csv,SKA1_LOW_beam.fits --conf spark.executorEnv.PYTHONHASHSEED=353 --total-executor-cores 80 --executor-memory 100G --driver-memory 100G --py-files arl.zip pipeline-partitioning_auto_spark.py --nfreqwin 512 --ntimes 7 --context 2d --nfacets 1 --nthreads 8 --nworkers 8 --parallelism 64 >full_python 2>&1
```

6 Conclusion and future work

By modeling and evaluating the IO of MID1 ICAL pipeline, we conclude that a data model considering data locality is quite critical. Spark has three levels of data locality for tasks, Process-local, Node-local and Any. By utilizing an extreme case, we put everything into a Spark task, the running time is close with that of StarPU. For distributed computing, it is important to decrease the amount of data communication.

Because network communication cost is expensive for distributed computing, it is important to do data reduction before data transport.

By now, only the IO cost is considered. However, the computing cost caused by EFs should be considered. The next prototyping work needs to include the real computing.

Reference


2012.