CASA Multi-MS Structure and Heuristics of Multi-MS Processing

- Users guide -
v1.2 (CASA 4.3+)

Sandra Castro, Justo Gonzalez

Table of Contents

1.0 INTRODUCTION **2**

2.0 MULTI-MS STRUCTURE AND CREATION **3**

Figure 1. Create a Mulit-MS using partition with scan separation axis. 6

3.0 INPUT MULTI-MS PROCESSING **7**

3.1 Split2 and Cvel2 Heuristics *7*

Figure 2. Regrid a Multi-MS using cvel2. The spws are combined in the output MMS. 9

3.2 Mstransform Heuristics *10*

Figure 3. Treat MMS as a monolithic MS and create an output MMS with a different axis. 11

4.0 ADDING PARALLELISM TO A TASK **12**

Figure 4. How to parallelize task applycal. 12

Figure 5. How to parallelize task cvel2. 13

5.0 RUNNING CASA WITH MPI AND MULTI-MS **14**

1.0 Introduction

Pre-imaging CASA tasks can benefit from parallelization if the Measurement Set (MS) is partitioned in disk. The partitioned MS in CASA is called Multi-MS (MMS). The Multi-MS has been available in CASA since version 4.0, where a major effort was made to improve the MMS format and the cluster infrastructure. Another major milestone was achieved in CASA version 4.1 with the creation of the MSTransform framework, which brought in the lower-level infrastructure necessary to finalize the parallelization of most pre-imaging CASA tasks.

The mstransform task can do everything that split, cvel, hanningsmooth and partition do. It has a more complex interface, which is mostly suitable for advanced users. There are simpler versions of mstransform in each of the individual tasks partition, split2, cvel2 and hanningsmooth2. Every transformation done in an MS can also be done in a Multi-MS almost transparently to the user.

Section 2 of this document will show the structure in disk of a Multi-MS and how to create an MMS from a normal MS. Section 3 shows how mstransform and other parallel tasks process an input MMS. Section 3 also gives details on the heuristics applied to the MMS when certain types of transformations are requested on the data. Section 4 shows examples how to parallelize some tasks in CASA.

2.0 Multi-MS Structure and Creation

A Multi-MS (MMS) is structured to have a reference MS on the top directory and a sub-directory called SUBMSS, which contains each partitioned sub-MS. A Multi-MS can be handled like a normal "monolithic" MS. It can be moved and renamed like any other directory. CASA tasks that are not MMS-aware can process it like a monolithic MS.

The reference MS contains links to the sub-tables of the first sub-MS. The other sub-MSs contain a copy of the sub-tables each. In order to reduce the volume of the MMS, the POINTING and SYSCAL tables (which are read-only in all use cases and identical for all sub-MSs) are stored only with the first sub-MS and linked into the other sub-MSs. The following is an example of a Multi-MS, which has 5 scans and has been partitioned in the scan axis.

1) Looking at the reference MS. Symbolic links have an @ at the end.

```
> ls uid_X2.mms/
```

```
ANTENNA@            ASDM_STATION@       FLAG_CMD@       PROCESSOR@          SYSCAL@
ASDM_ANTENNA@       CALDEVICE@          HISTORY@        SOURCE@             table.dat
ASDM_CALWVR@        DATA_DESCRIPTION@   OBSERVATION@    SPECTRAL_WINDOW@    table.info
ASDM_RECEIVER@      FEED@               POINTING@       STATE@              WEATHER@
ASDM_SOURCE@        FIELD@              POLARIZATION@   SUBMSS/
```

2) Looking at the sub-MSs directory. The sub-MS names have the MMS basename followed by a 4-digit index.

```
> ls uid_X2.mms/SUBMSS/
```

```
uid_X2.0000.ms/  uid_X2.0001.ms/  uid_X2.0002.ms/  uid_X2.0003.ms/  uid_X2.0004.ms/
```

3) Looking at the first sub-MS, which is the only one with a physical copy of the POINTING and SYSCAL tables.

```
> ls uid_X2.mms/SUBMSS/uid_X2.0000.ms/
```

```
ANTENNA/          ASDM_STATION/       FLAG_CMD/       PROCESSOR/          table.dat
ASDM_ANTENNA/     CALDEVICE/          HISTORY/        SOURCE/             table.f1
ASDM_CALWVR/      DATA_DESCRIPTION/   OBSERVATION/    SPECTRAL_WINDOW/    table.f2
ASDM_RECEIVER/    FEED/               POINTING/       STATE/              table.info
ASDM_SOURCE/      FIELD/              POLARIZATION/   SYSCAL/             WEATHER/
```

4) Looking at the second sub-MS, which has symbolic links to the POINTING and SYSCAL tables in the first sub-MS.

```
> ls –l uid_X2.mms/SUBMSS/uid_X2.0001.ms/

 ANTENNA/
 ASDM_ANTENNA/
 ASDM_CALWVR/
 ASDM_RECEIVER/
 ASDM_SOURCE/
 ASDM_STATION/
 CALDEVICE/
```

```
DATA_DESCRIPTION/
FEED/
FIELD/
FLAG_CMD/
HISTORY/
OBSERVATION/
POINTING -> ../uid_X2.0000.ms/POINTING/
POLARIZATION/
PROCESSOR/
SOURCE/
SPECTRAL_WINDOW/
STATE/
SYSCAL -> ../uid_X2.0000.ms/SYSCAL/
table.dat
table.f1
table.f2
table.info
WEATHER/
```

The table.info file inside the reference MS contains information on the axis used to partition the original MS. This information is written by the partition task and carried over by other tasks.

5) Looking at table.info in the reference MS.

```
> cat uid_X2.mms/table.info

Type = Measurement Set
SubType = CONCATENATED

AxisType = scan
This is a MeasurementSet Table holding measurements from a Telescope
This is a measurement set Table holding astronomical observations
Virtual concatenation of the following tables:
  SUBMSS/uid_X2.0000.ms
  SUBMSS/uid_X2.0001.ms
  SUBMSS/uid_X2.0002.ms
  SUBMSS/uid_X2.0003.ms
  SUBMSS/uid_X2.0004.ms
```

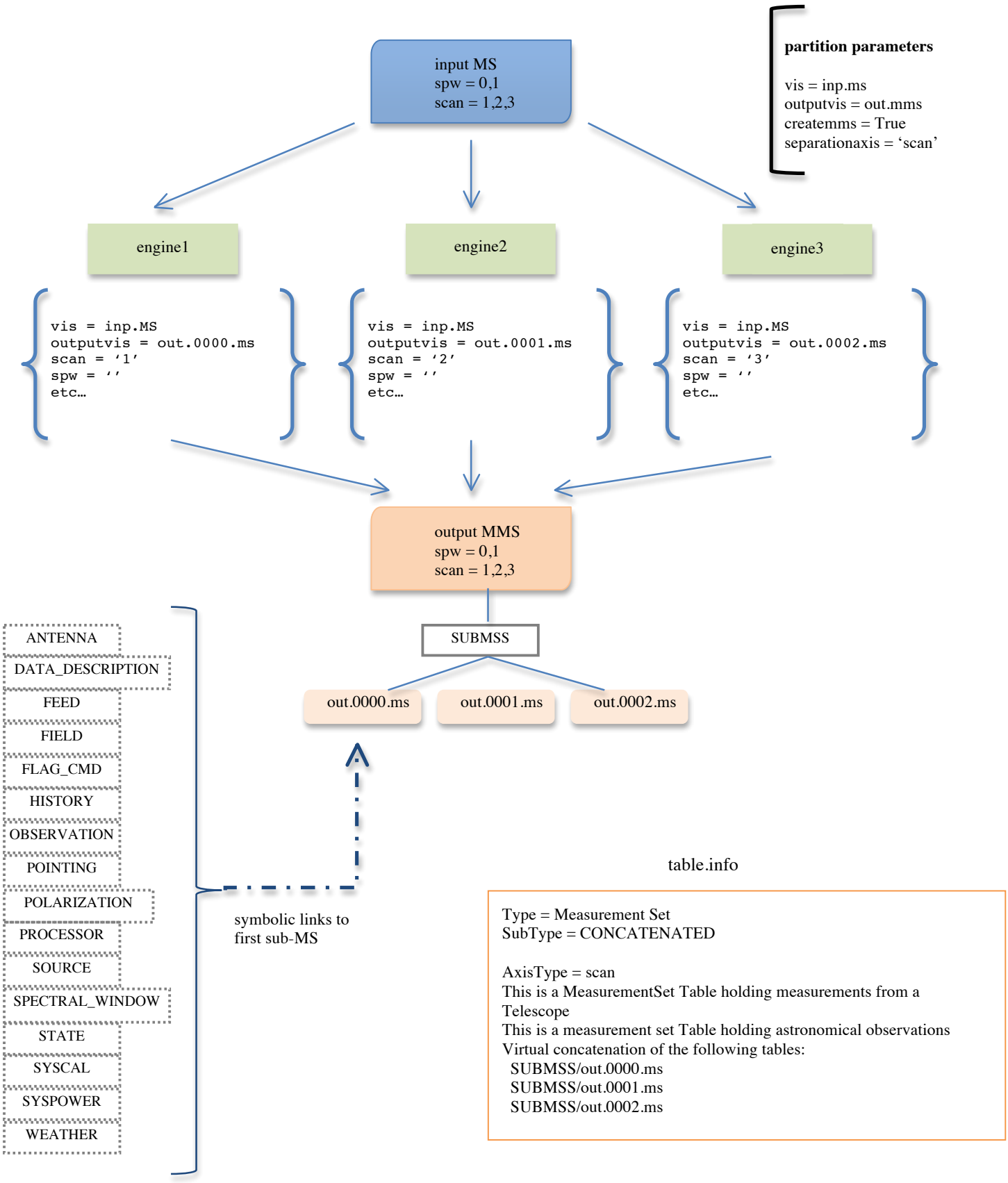
The *partition* task is the main task to create a Multi-MS. It takes an input Measurement Set and creates an output Multi-MS based on the data selection parameters. Partition accepts three axis to do separation across: auto, scan or spw. The default auto will first separate the MS in spws, then in scans. It calculates the minimum between the number of selected spws and the value given in *numsubms* to determine the spws partition. Next it calculates the mathematical ceiling between the *numsubms* and the number of spws. This last calculation will give the number of scan partitions to have for the available scans. Each partitioned MS is referred to as a sub-MS. The user may force the number of sub-MSs in the output MMS by setting the parameter *numsubms*.

It is important to notice that in the ideal case, the number of computer nodes should match the number of sub-MSs so that there is not idle node in the case when there are more nodes than sub-MSs to work on.

Task partition uses two helper classes to handle the parallel jobs; ParallelTaskHelper and ParallelDataHelper, which are discussed in more detail in Section 4. Special care needs to be taken in order to consolidate the sub-tables of the MMS because the spectral window indices in the output are re-indexed in each engine to the same initial index and this needs to be consolidated later. The sub-tables are merged after all engines return for post-processing.

Figure 1 shows a diagram of the creation of a Multi-MS. The example input MS contains 3 scans and 2 spws and will be separated in the scan axis. The output MMS will be created with 3 sub-MSs, each with a different scan.

Figure 1. Create a Mulit-MS using partition with scan separation axis.



3.0 Input Multi-MS Processing

There are essentially two ways of processing an input MMS, these are in parallel using a cluster or as a monolithic MS. Basically every task in CASA can process the MMS as a monolithic MS, because the MMS is made to look like a normal MS. This processing will happen automatically in a transparent way to the user. Examples of such tasks are: listobs, gencal, gaincal, etc.

Other tasks were modified to process the input MMS in parallel such as to speed up the processing, or because they need to modify the input MMS or create a new output MMS. Tasks that only modify the input MMS such as flagdata and applycal may raise NULL MS Selection exceptions depending on the way the MMS was created and the data selection given in the parameters. These exceptions are harmless in these cases and are hidden from the user's terminal. Flagdata's summary mode does not modify the input, but creates output dictionaries in each parallel engine. These dictionaries are consolidated into one single output dictionary, which is returned to the user.

Tasks that create a new output such as split2, cvel2, hanningsmooth2 and mstransform will process each input sub-MS in parallel whenever possible. In these cases, the output is a Multi-MS with the same separation axis as the input. In some cases, the heuristics are more complicated and it is not possible to process the MMS in parallel or to create an output MMS. These cases are discussed in the following sections.

The only tasks that can create a Multi-MS from a normal MS are partition and mstransform, as seen in Figure 1. These two tasks have a parameter called *createmms* that controls how to partition the MS. The simple relation between input and output for all tasks is the following:

input MS	→	output MS
input MMS	→	output MMS
input MS	→	output MMS (only partition and mstransform)

3.1 Split2 and Cvel2 Heuristics

Task split2 will work seemingly on both MS and MMS. If the input is a normal MS, the output can only be a normal MS. If the input is a Multi-MS, the task will automatically create an output MMS. The user may override this behaviour by setting the parameter *keepmms* to False, in which case the output will be a monolithic MS. In most use-cases this will not be needed.

In the case of time averaging with combination across scans in split2, the way the input MMS was initially partitioned may interrupt the creation of an output MMS. The correct axis to use when first creating the MMS for this type of transformation is the *spw* axis. When time averaging across scans is requested the task will check if each sub-MS contains the selected scans and if the duration of the scans is >= to the requested time bin. If the sub-MSs are not self-contained, the task will raise an exception and exit. The user may still perform time averaging across scans if the parameter *keepmms* is set to False, which will create a normal monolithic MS as the output. The other option is to use task mstransform in this case, because it has the ability to process the MMS as a monolithic MS and still create an output MMS (See Section 3.2). The following is an example of the error message given in this case.

```
WARN     split2::::casa     Cannot process MMS in parallel when combine='scan' because the sub-MSs do not
                            contain all the selected scans
SEVERE   split2::::casa     Please set keepmms to False or use task mstransform in this case.
SEVERE   split2::::casa     An error occurred running task split2.
```

In a similar way, task cvel2 has a constraint that depends on which axis the input MMS was partitioned. Because cvel2 does a combination of all selected spws before regridding the data, the most feasible partition axis for the input is *scan*. The task will check if each sub-MS contains all the selected spws and if not, it will raise an exception. The user may set *keepmms* to False or run mstransform instead.

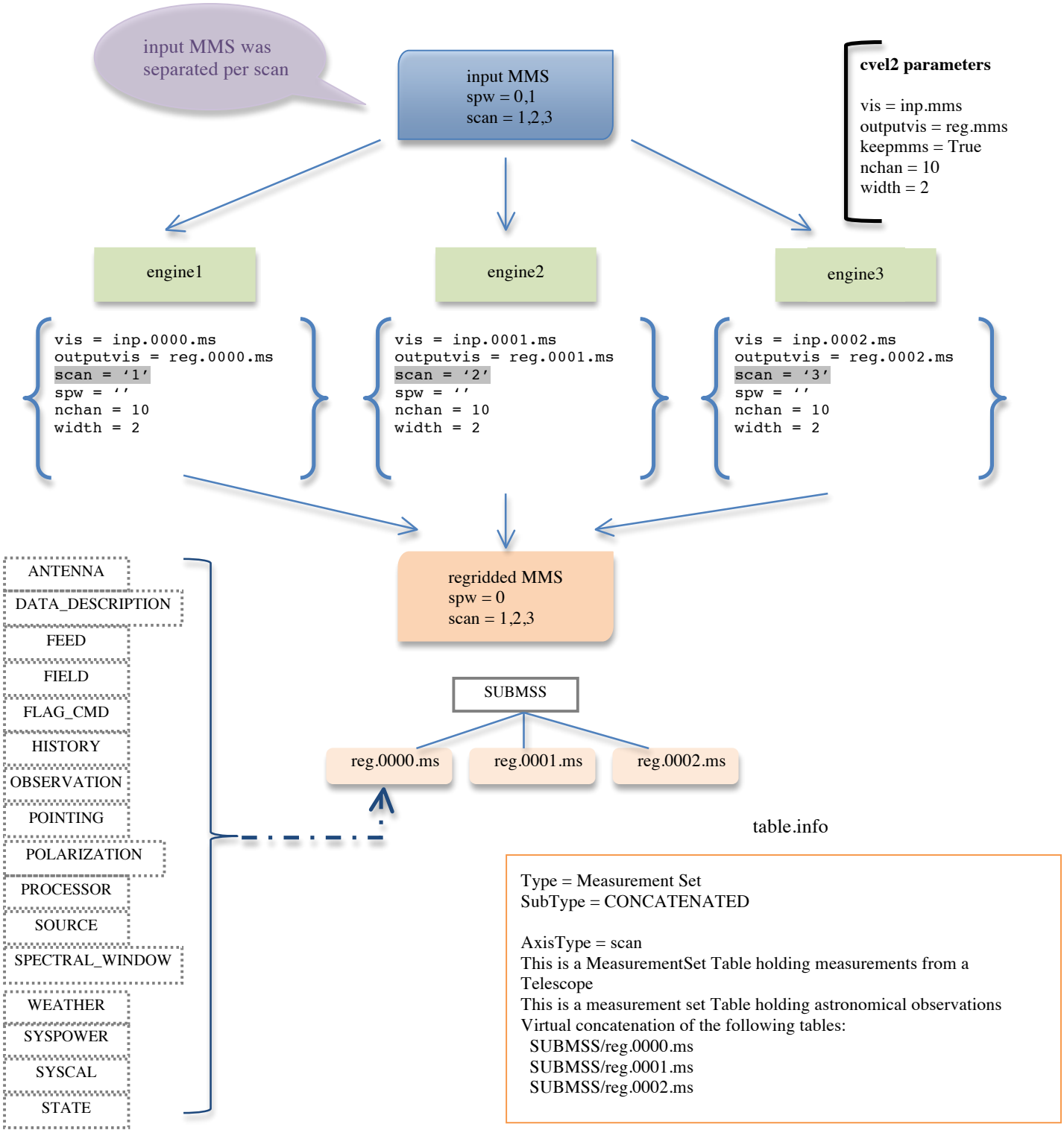
```
WARN     cvel2::::casa      Cannot combine spws in parallel because the subMSs do not contain all the
                            selected spws
SEVERE   cvel2::::casa      Please set keepmms to False or use task mstransform in this case.
SEVERE   cvel2::::casa      An error occurred running task cvel2.
```

Figure 2 shows an example of running task cvel2 to change the channel structure of a Multi-MS. The input MMS in this example has 3 scans and 2 spws which are partitioned in the scan axis into 3 sub-MSs. The input MMS is processed in parallel using 3 engines. Each engine combines the spws of one sub-MS (which contains one scan and all spws) and creates an output sub-MS. All 3 output sub-MSs are then post-processed to create the final MMS.

The most reliable way of obtaining information of a Multi-MS structure is by using the *listpartition* task. This task lists the following properties of a Multi-MS: separation axis, sub-MS name, scan, spw, number of channels per spw, number of rows for each scan and the size in disk. The following is the output of task *listpartition* for the input MMS shown in Figure 2:

```
This is a multi-MS with separation axis = scan
Sub-MS          Scan   Spw    Nchan     Nrows   Size
inp.0000.ms     1      [0 1]  [64 64]   1068    7.8M
inp.0001.ms     2      [0 1]  [64 64]   1080    7.4M
inp.0002.ms     3      [0 1]  [64 64]   1080    7.4M
```

Figure 2. Regrid a Multi-MS using cvel2. The spws are combined in the output MMS.



3.2 Mstransform Heuristics

Task mstransform will process an input MMS in parallel whenever possible. Each sub-MS of the MMS will be processed in a separate engine and the results will be post-processed at the end to create an output MMS. The output MMS will have the same separationaxis of the input MMS, which will be written to the table.info file inside the MMS directory.

Naturally, some transformations available in mstransform require more care when the user first partition the MS. If one wants to do a combination of spws by setting the parameter *combinespws = True* in mstransform, the input MMS needs to contain all the selected spws in each of the sub-MSs or the processing will fail. For this, one may set the initial *separationaxis* to *scan* or use the default *auto* with a proper *numsubms* set so that each sub-MS in the MMS is self-contained with all the necessary spws for the combination.

The task will check if the sub-MSs contain all the selected spws when *combinespws=True* and if not, it will issue a warning and process the input MMS as a monolithic MS. In this case, the separation axis of the output MMS will be set to *scan*, regardless of what the input axis was. This possibility is not available in cvel2.

A similar case happens when the separation axis of the input MMS is per *scan* and the user asks to do time averaging with time spanning across scans. If the individual sub-MSs are not self-contained of the necessary scans and the duration of the scans is shorter than the given *timebin*, the spanning will not be possible. In this case, the task will process the input MMS as a monolithic MS and will set the axis of the output MMS to *spw*. This possibility is not available in split2.

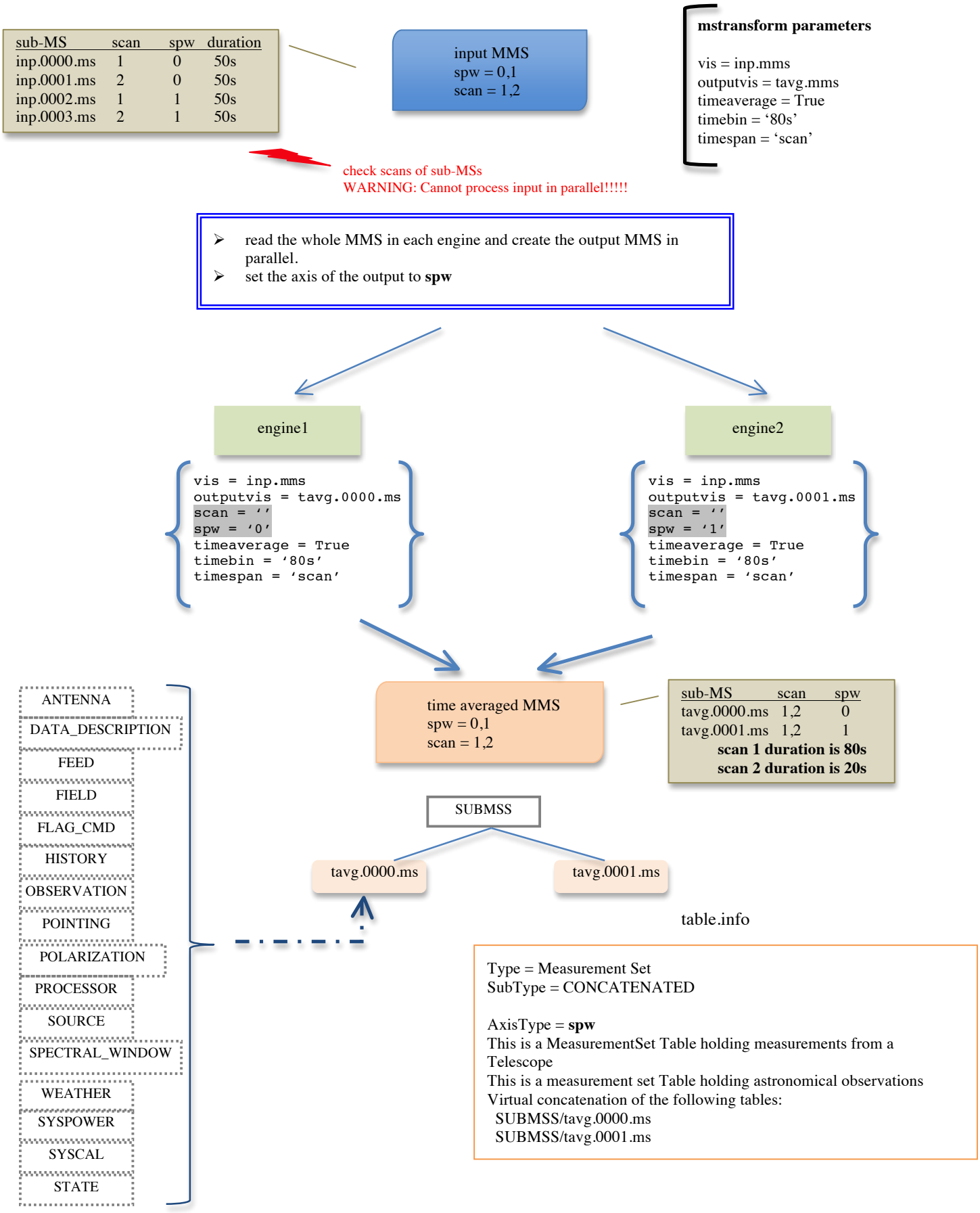
It is important that the user sets the separation axis correctly when first partitioning the MS. See the table below for when it is possible to process the input MMS in parallel or not, using mstransform.

input MMS axis	combinespws=True	nspw > 1	timeaverage=True, timespan=scan
scan	yes	yes	no
spw	no	no	yes
auto	maybe	maybe	maybe

In the event that the user requests two transformations at the same time: combination of spectral windows and time averaging across scans on an input MMS, similar checks will be applied in order to determine if it is possible to process the input in parallel. First, the task will check if each sub-MS contains the selected spws and only in case of success, it will check if it contains all the scans with proper duration. If the checks are unsuccessful, the input MMS will be processed as a monolithic MS and the output will be in this case a normal monolithic MS.

Figure 3 gives an example of an input MMS processed as a monolithic MS when doing time averaging across scans. The MMS in this example was partitioned using the default *auto* axis and contains spw=0,1 and scan=1,2; each scan has a duration of 50s. Each input sub-MS has one scan and one spw, which is not enough to perform time averaging across scans with a timebin=80s.

Figure 3. Treat MMS as a monolithic MS and create an output MMS with a different axis.



4.0 Adding Parallelism to a Task

In order to make a CASA task process an MMS in parallel, a few lines of code need to be added to the Python code. The Class ParallelTaskHelper will take care of creating the jobs that will be sent to each parallel engine. Each engine will call an instance of CASA with the task parameters. This example is for tasks that do not create an output MS. They either create a calibration table or merely add/change something in the input MS, such as the CORRECTED_DATA column. Figure 4 shows how task *applycal* is modified to process an MMS in parallel. The text in blue shows what is added to the task code.

Line 2 : import the ParallelTaskHelper class.
Line 9 : test if input is a Multi-MS and execute the if statement if True.
Line 11 : add the absolute path to the filename given in parameter *gaincal*. This is so that each engine finds the filename in the correct place.
Line 12 : initialize the ParallelTaskHelper with the task to run and the parameters of the task.
Line 13 : call the go() method, which will call the following methods:
initialize(): add an absolute path to parameter *vis*;
generateJobs(): generate a call to the task for each sub-MS in the MMS and add to a list of jobs to be executed by the cluster;
executeJobs(): execute the list of jobs;
postExecution(): consolidates list of return values from all engines.
Line 17 : this is the start of the normal *applycal* task, which will be executed in each engine.

Figure 4. How to parallelize task applycal.

```
1  from taskinit import *
2  from parallel.parallel_task_helper import ParallelTaskHelper
3
4  def applycal(vis, field, spw, intent, ….)
5
6      casalog.origin('applycal')
7
8      # Process in parallel if input is a Multi-MS
9      if ParallelTaskHelper.isParallelMS(vis):
10         # To be safe convert file names to absolute paths.
11         gaintable = ParallelTaskHelper.findAbsPath(gaintable)
12         helper = ParallelTaskHelper('applycal', locals())
13         helper.go()
14         return
15
16       # Start of normal task
17       try:
18           mycb = cbtool()
19           if ((type(vis)==str) & (os.path.exists(vis))):
20               # Add CORRECTED_DATA column
21               mycb.open(filename=vis,compress=False,
22               addcorr=True,addmodel=False)
23           else:
24               raise Exception, 'Visibility data set not found - please
25                                 verify name'
   ….
```

When the task creates an output we use a different helper class. The ParallelDataHelper class inherits from ParallelTaskHelper and is used to handle the heuristics of processing an MMS. The example in Figure 5 shows how task *cvel2* implements parallelism.

Figure 5. How to parallelize task cvel2.

```
1  from taskinit import *
2  from parallel.parallel_data_helper import ParallelDataHelper
3
4  def cvel2(vis, outputvis, keepmms, passall, field, spw, ….)
5
6      # Initialize the helper class
7      pdh = ParallelDataHelper("cvel2", locals())
8
9      casalog.origin('cvel2')
10
11     # Validate input and output parameters
12     try:
13         pdh.setupIO()
14     except Exception, instance:
15         casalog.post('%s'%instance,'ERROR')
16         return False
17
18     # Input vis is an MMS
19     if pdh.isParallelMS(vis) and keepmms:
20         status = True
21
22         # Work the heuristics of combinespws=True
23         retval = pdh.validateInputParams()
24         if not retval['status']:
25             raise Exception, 'Unable to continue with MMS processing'
26
27         pdh.setupCluster('cvel2')
28
29         # Execute the jobs
30         try:
31             status = pdh.go()
32         except Exception, instance:
33             casalog.post('%s'%instance,'ERROR')
34             return status
35
36         return status
37
38     # Start of normal task
39     mtlocal = mttool()
40     try:
41         # Gather all the parameters in a dictionary.
42         config = {}
       ….
```

The following describes the above code:

Line 2 : import ParallelDataHelper class, which inherits from ParallelTaskHelper;
Line 7 : instantiate the ParallelDataHelper class with the task name and the local parameters;
Line 13 : check if parameter *vis* exist and if *outputvis* is given;
Line 19 : process in parallel only if input is a Multi-MS and parameter *keepmms* is True;
Line 23 : work out the heuristics, which will check if it is possible to process in parallel based on the separation axis of the input MMS. In the case of *cvel2* it will check if the sub-MSs contain all the selected spws in order to combine them before regridding.

Line 27 : initialize the ParallelTaskHelper, which will create a cluster using MPI (if available) or it will fallback to the default simple_cluster.
Line 31 : call the go() method, which will call the following overloaded methods:
initialize(): create a temporary directory for the output sub-MSs;
generateJobs(): decide how to create jobs for input MMS in parallel or as a monolithic-MS. It also works out the job commands depending on the separation axis of the output.
executeJobs(): execute the list of jobs;
postExecution(): consolidates the output sub-MSs created by each engine. This means, creates the final structure of the MMS and consolidates the sub-tables.
Line 39 : this is the start of the normal *cvel2* task, which will be executed in each engine.

5.0 Running CASA with MPI and Multi-MS

The following document shows how to setup your environment in order to run CASA with MPI. https://svn.cv.nrao.edu/svn/casa/trunk/gcwrap/python/scripts/mpi4casa/CASA-4.3-MPI-Parallel-Processing-Framework-Installation-and-advance-user-guide.pdf. Once the installation of CASA is all set, start it by running:

```
>mpirun –n 11 casapy –nogui –log2term
<CASA>partition('inp.ms', outputvis='out.mms', separationaxis='spw', numsubms=10, flagbackup=False)
```

alternatively, you can run your script on two or more computers, if they have been properly configured as explained in the above document.

```
>cat config.hostfile
almahpc01 slots=6
almahpc02 slots=5

>mpirun –hostfile config.hostfile casapy
<CASA>flagdata('out.mms', mode='manual', spw='*:1~10')
```

Running CASA functional tests with MPI.

```
>mpirun –n 4 casapy –nogui –log2term –c $TESTDIR/runUnitTest.py test_mstransform_mms
```