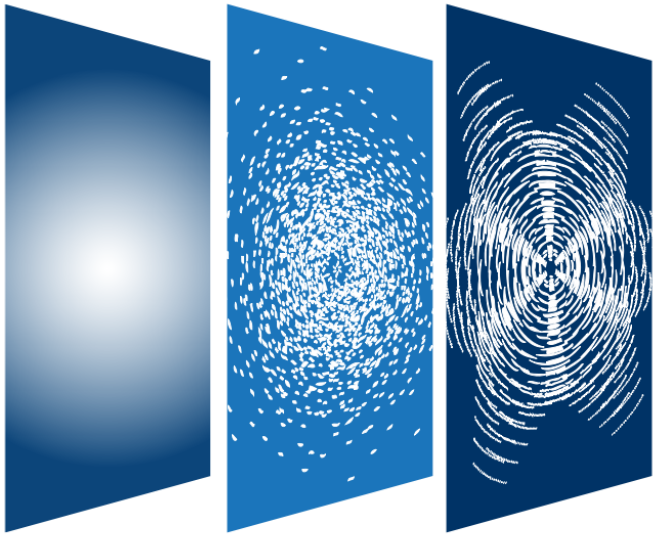


CASA

Common Astronomy Software Applications

# CASA Parallelization and High Performance Computing

## Concepts, Implementation, and Issues


Jeff Kern

CASA Group Lead


Atacama Large Millimeter/submillimeter Array

Expanded Very Large Array

Robert C. Byrd Green Bank Telescope

Very Long Baseline Array

# CASA and HPC

- CASA does not have a HPC issue.
  - We must support efficient operation on the available hardware.
- CASA has focused on two "standard" systems:

| Workstation | Cluster |
| --- | --- |
| • Multi-core system<br>• Local disk<br>• Single shared memory | • Many multi-core nodes<br>• High performance network file system (Lustre)<br>• No shared memory access |

# Definitions

- Core: A single processing element which reads and executes instructions.
- Node: A single host, in modern systems usually has multiple cores.
- Engine: A single instance of CASA connected through a messaging interface, in most cases a single engine consumes a single core.
- Master Engine: The primary engine which is responsible for dispatching jobs to other engines
- Multi-Process: Many independent processes each with their own process space.
- Muti-Threaded: A single process with multiple threads of execution, multiple threads can share the same memory object.

# The processing performance problem

- Many tasks in CASA require traversing the entire data set and are data IO limited.

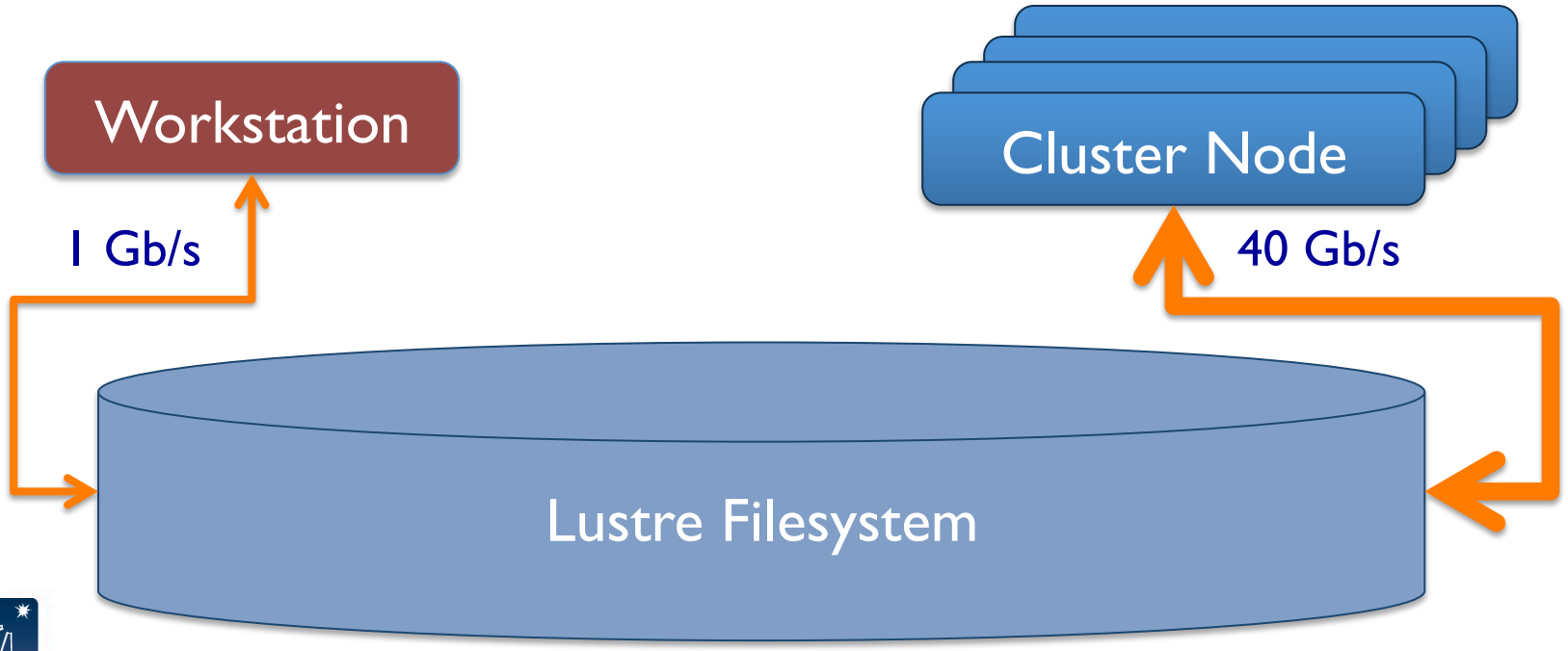
| | Peak Data Rate | OS Realized Data Rate | Time for 500 GB file |
|---|---|---|---|
| SATA Disk | 115 MB/s | 60 MB/s | 2h 22 m |
| Raid | 200-500 MB/s | 375 MB/s | 22 m |
| Lustre (10 GB) | 1.2 GB/s | 700 MB/s | 12 m |

- Even imaging [in the simplest case] is IO limited, requiring about 50 MB/s per core to prevent the CPU from incurring wait states.

*https://science.nrao.edu/facilities/evla/data-processing/hardware-recommendations*

# Lustre

- Running against Lustre can provide significant performance improvement, but where you run from matters.
- How the data is arranged on Lustre impacts performance as well.



# Embarrassingly Parallel Approach

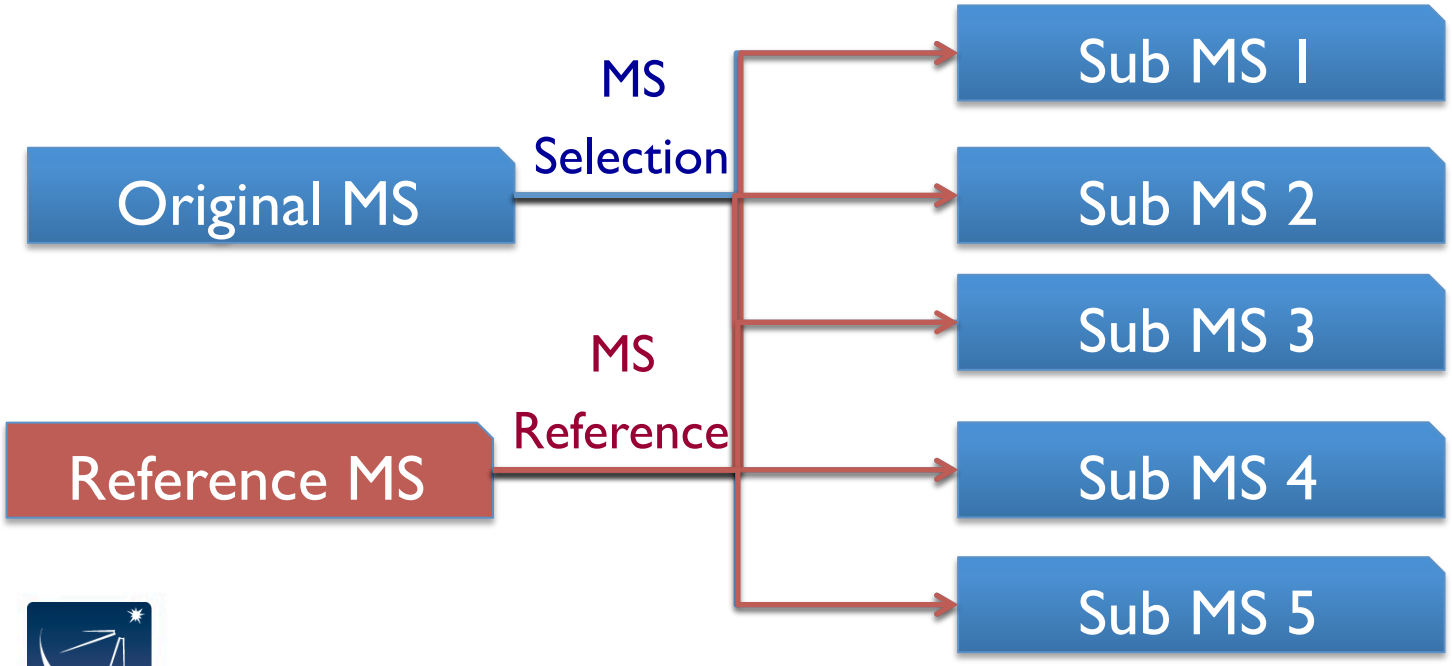
*An embarrassingly parallel workload is one for which little or no effort is required to separate the problem into a number of parallel tasks, this is often the case where there is exists no dependency between the parallel tasks.*

*-Wikipedia*

- Most of the tasks which require access to large amounts of data fit this description.
  - Flagging of Data
  - Applying Calibration
  - Time Averaging Data
- Imaging does not strictly fit this definition but can the communication between processes is only at certain points in the cycle so it is also an easily solved issue.

# Reference MSs

- The easiest way to parallelize in CASA is to have multiple instances of CASA each running on a subset of the data.
  - To simplify the interface and minimize bookkeeping for the user we use a reference MS.

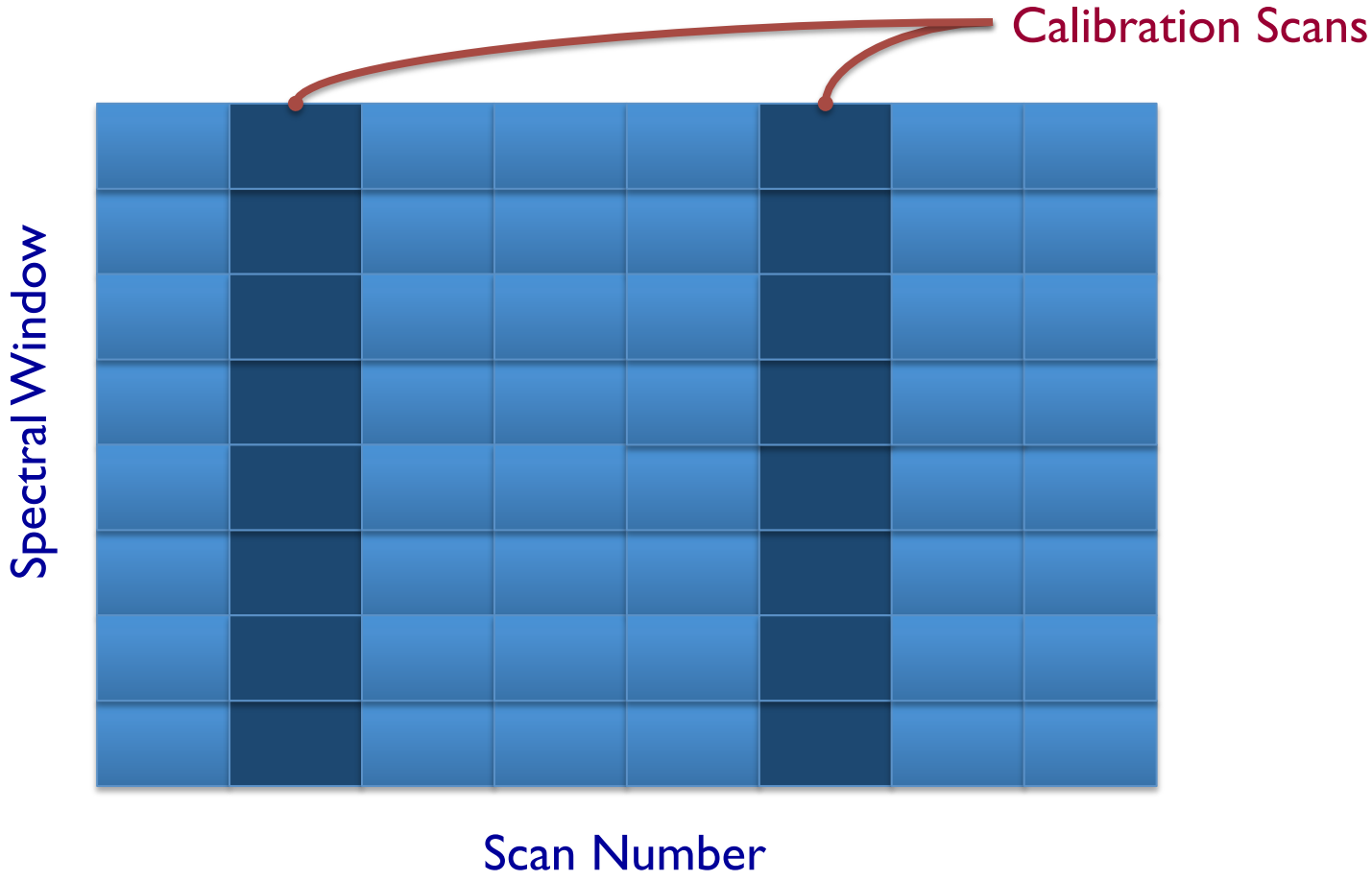


# Partition

- To simplify creation of the reference MS, use the task Partition.
- Contains usual data selection parameters:
  - Data column
  - Field
  - Spw
  - Antenna
  - Scan
  - Scan Intent
  - Array
  - UV-Range
- If *createmms* is true a reference MS is created which contains at least *numsubms* sub MSs
- Partitioning also has the effect of distributing the sub-MSs across multiple raid controllers on Lustre, thus improving I/O performance.

# Separation Axis

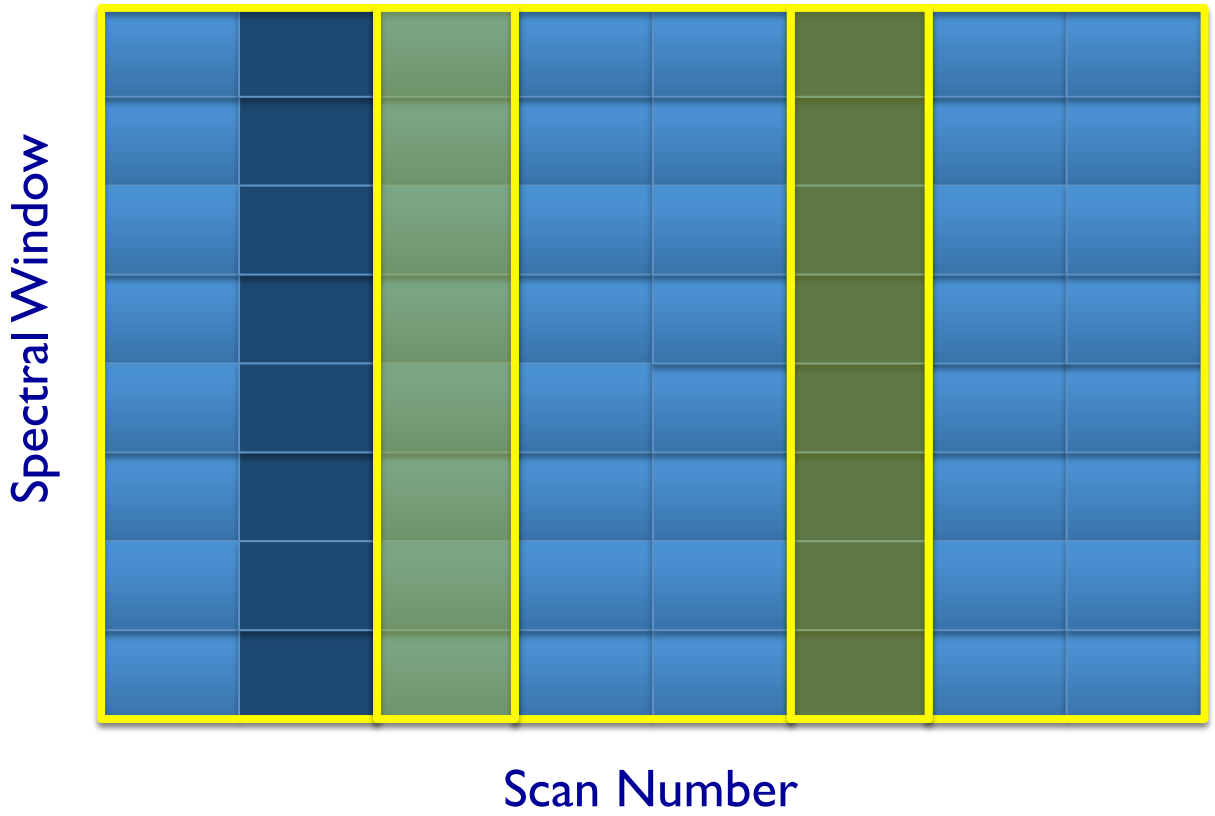
- Partition accepts three axis to do separation across *default*, *spw*, *scan*



# Separation Axis

Separation Axis = 'scan'

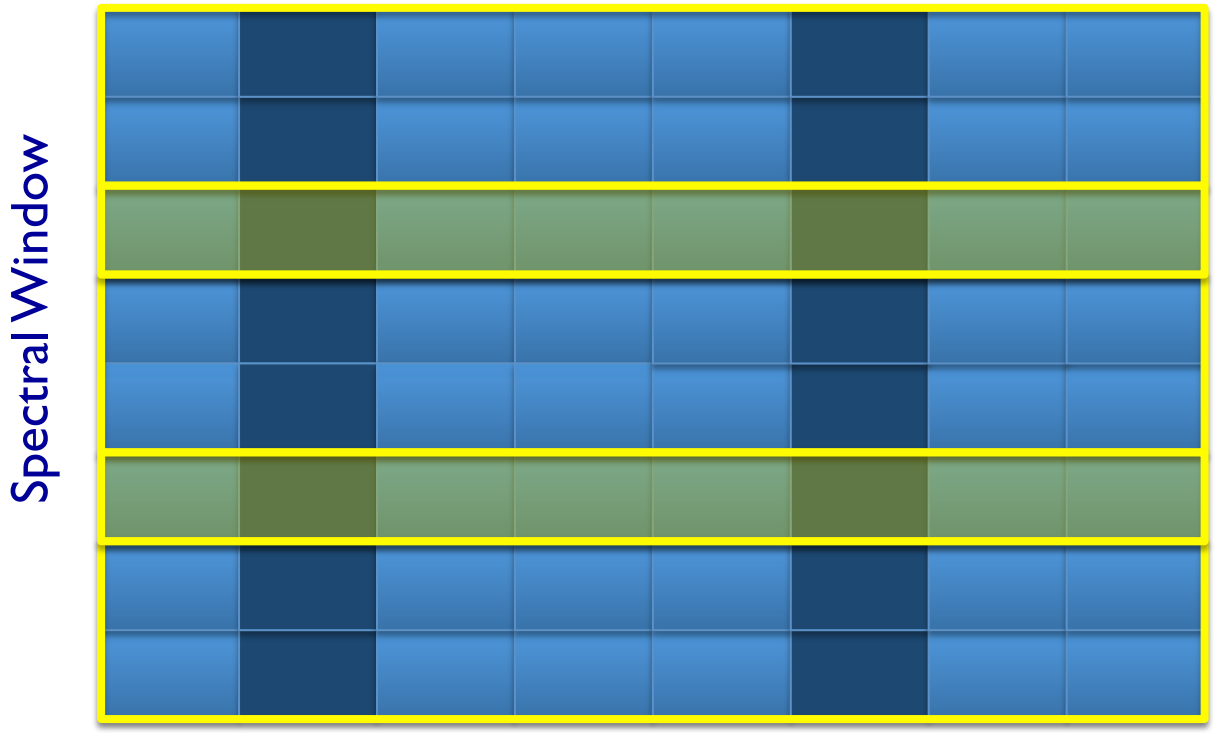
Num Sub MS = 5



# Separation Axis

Separation Axis = 'spw'

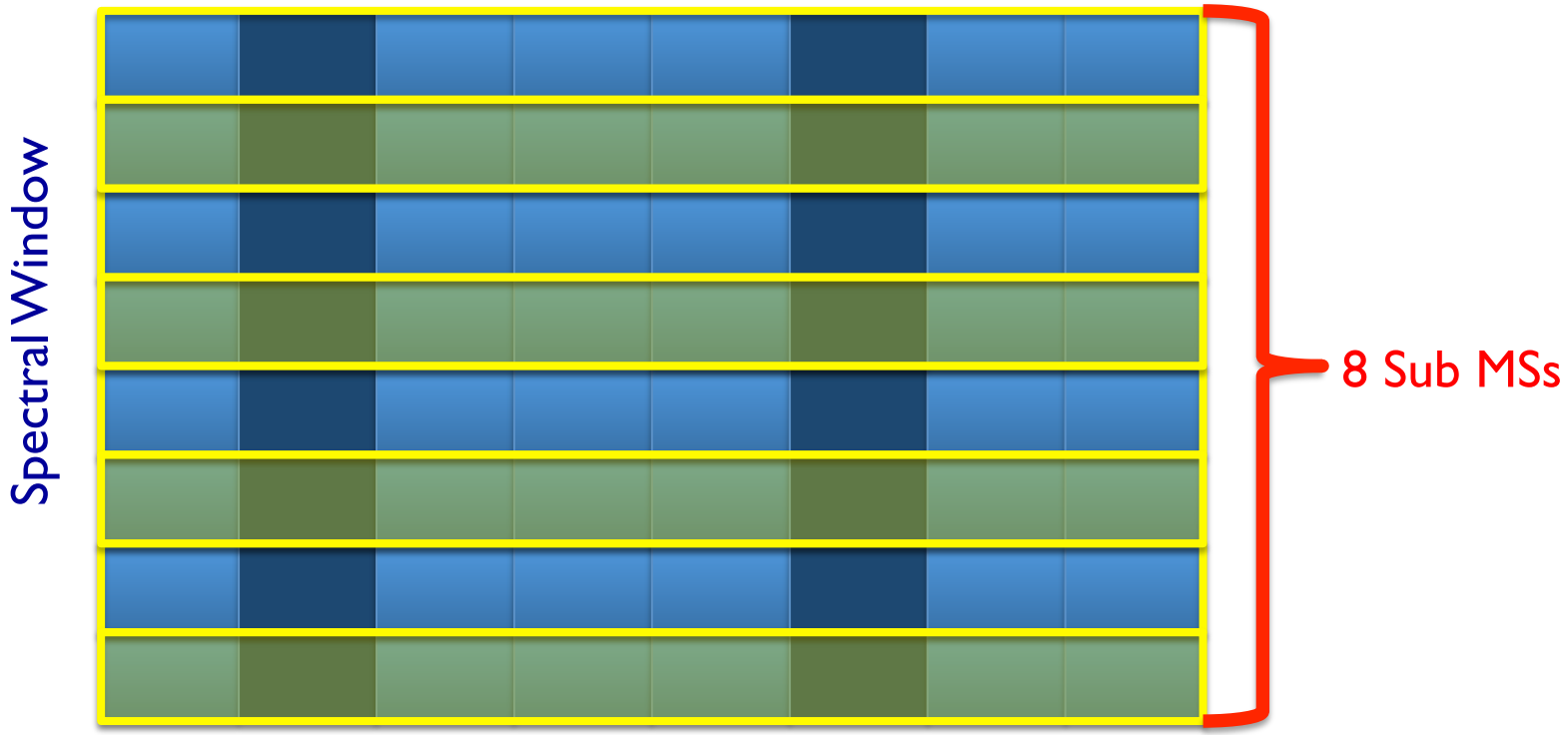
Num Sub MS = 5



# Separation Axis

Separation Axis = 'spw'

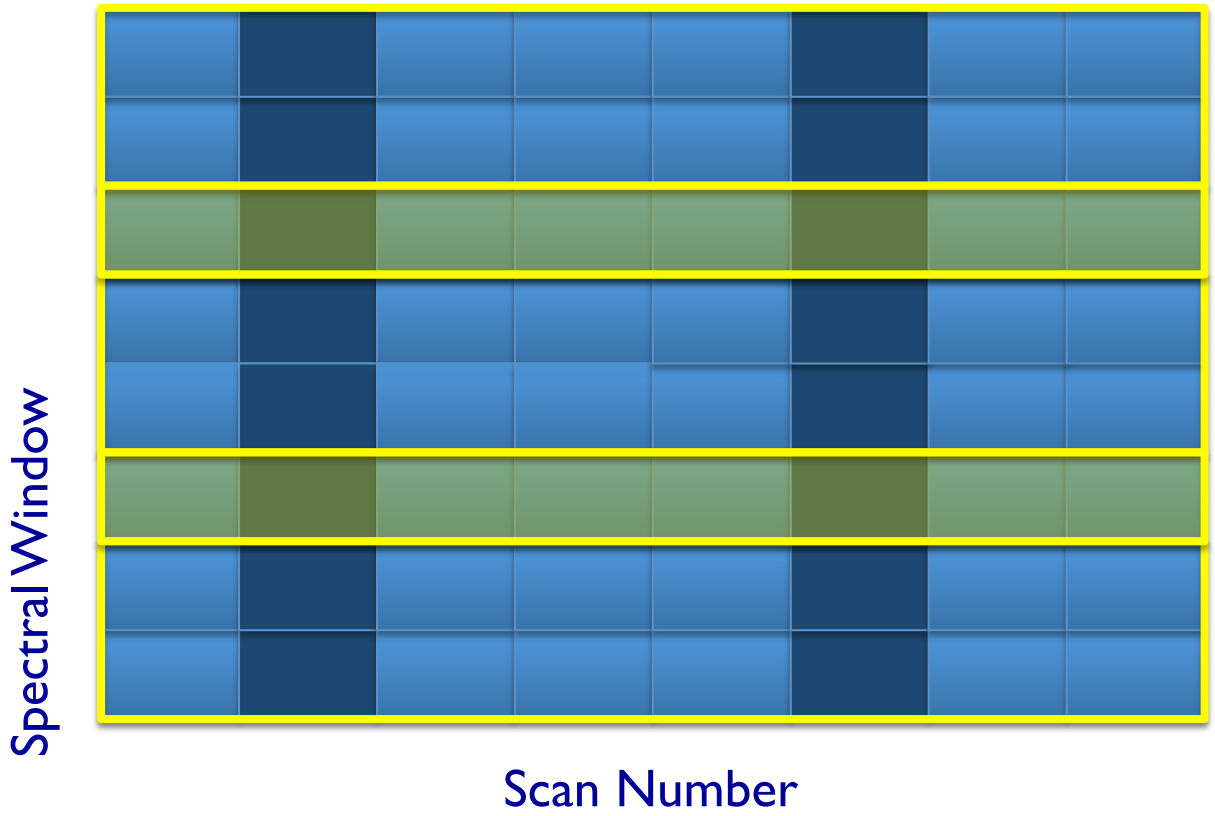
Num Sub MS = 12



# Separation Axis

Separation Axis = 'default'

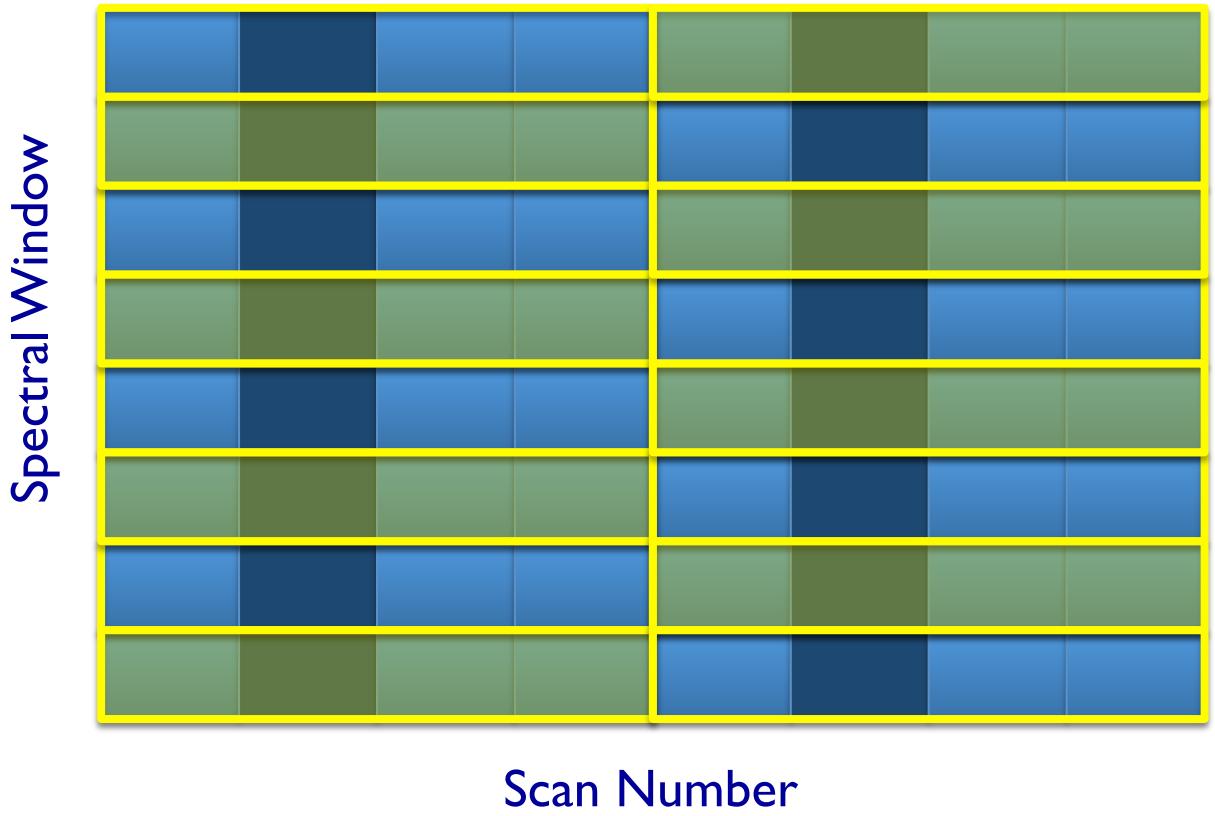
Num Sub MS = 5



# Separation Axis

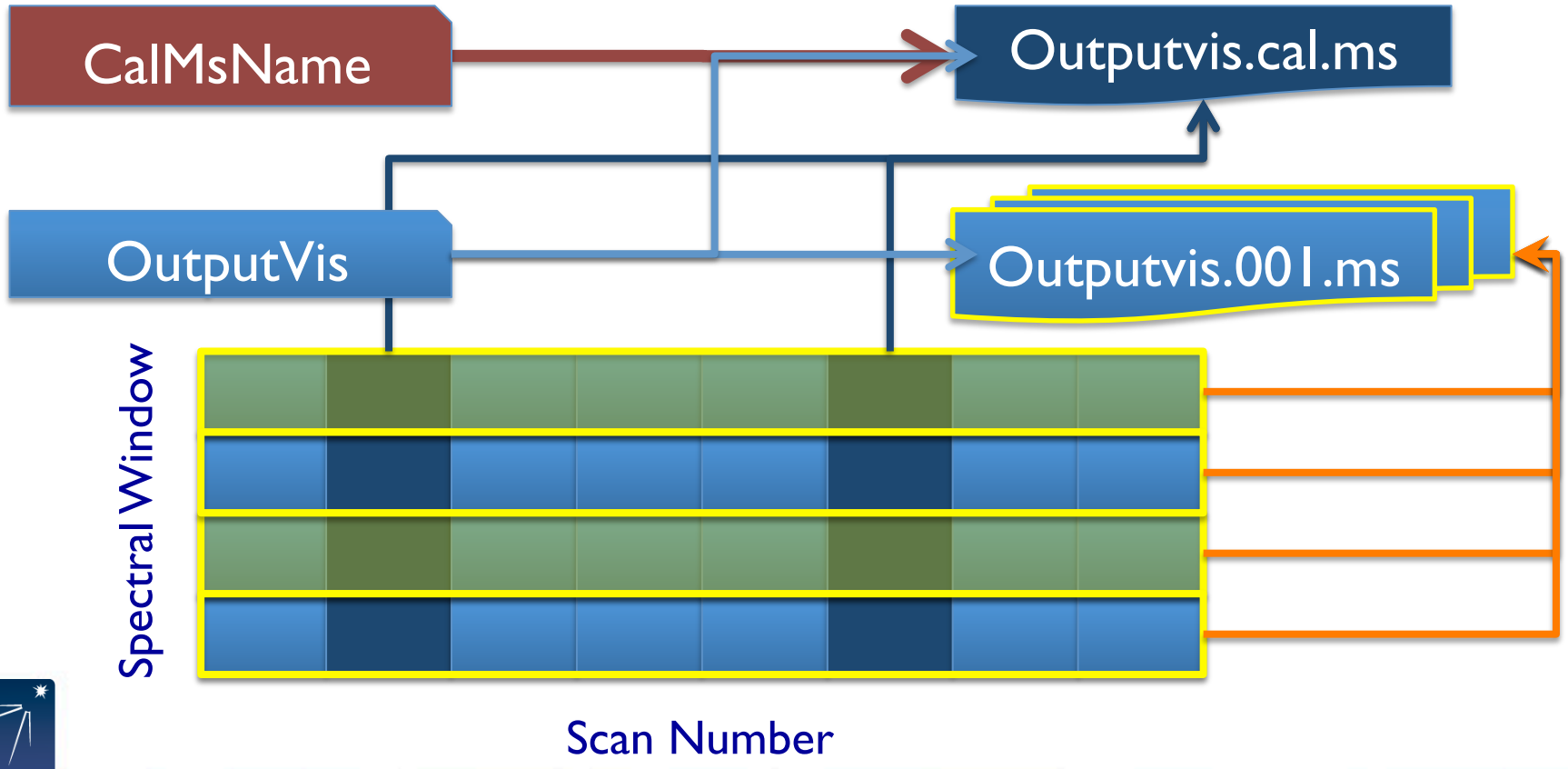
Separation Axis = 'default'

Num Sub MS = 12



# Separation Axis

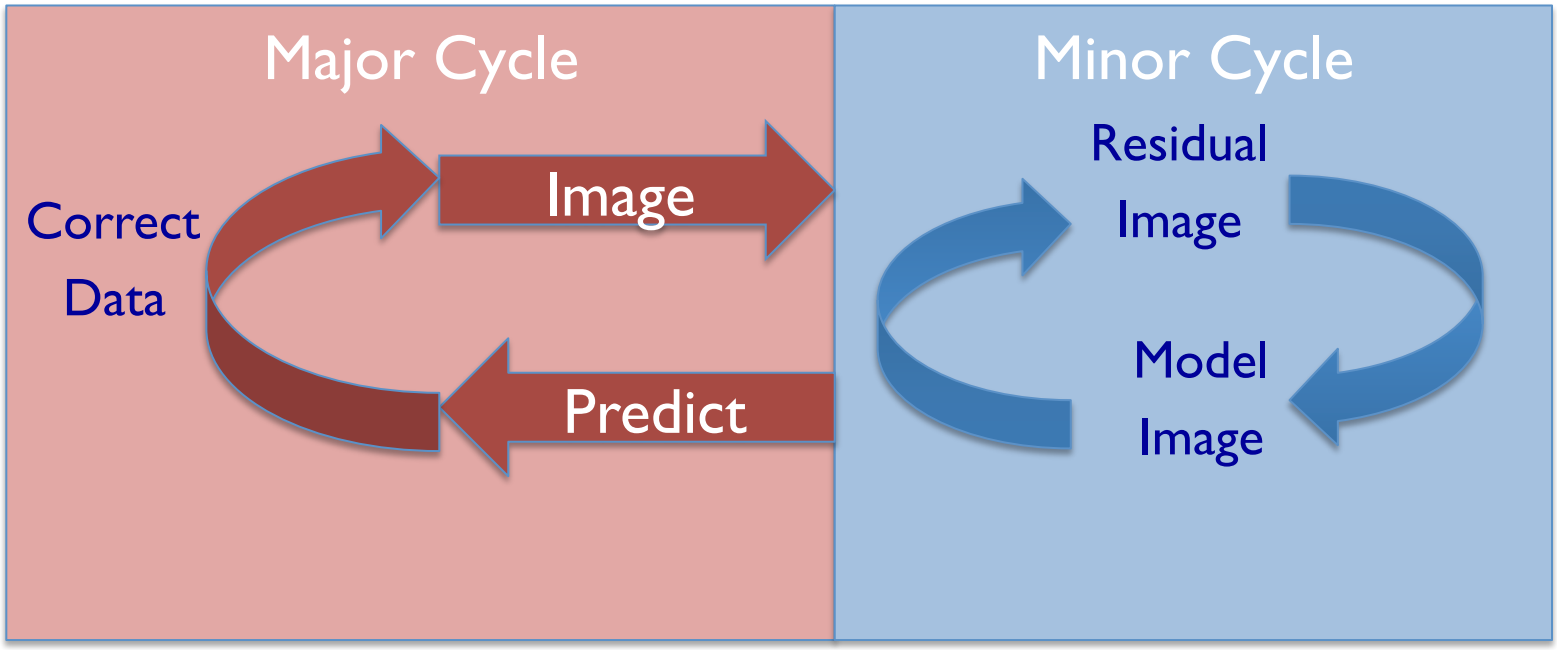
- Setting *calmsselection* to 'manual' allows selection on Field, Scan, or Intent.
- 'auto' does a selection based on scan intent



# Pre-Imaging Tasks

- After partitioning, any task which works on the MS in place (e.g. does not create a new MS) and does not modify the subtables can be parallelized fairly easily.
- So far I have only done those tasks which offered the most significant performance improvement for minimal investment :
  - ClearCal
  - FlagData
  - ApplyCal
- More tasks can easily be added as we identify which are most necessary
  - Flag_cmd
  - SetJy

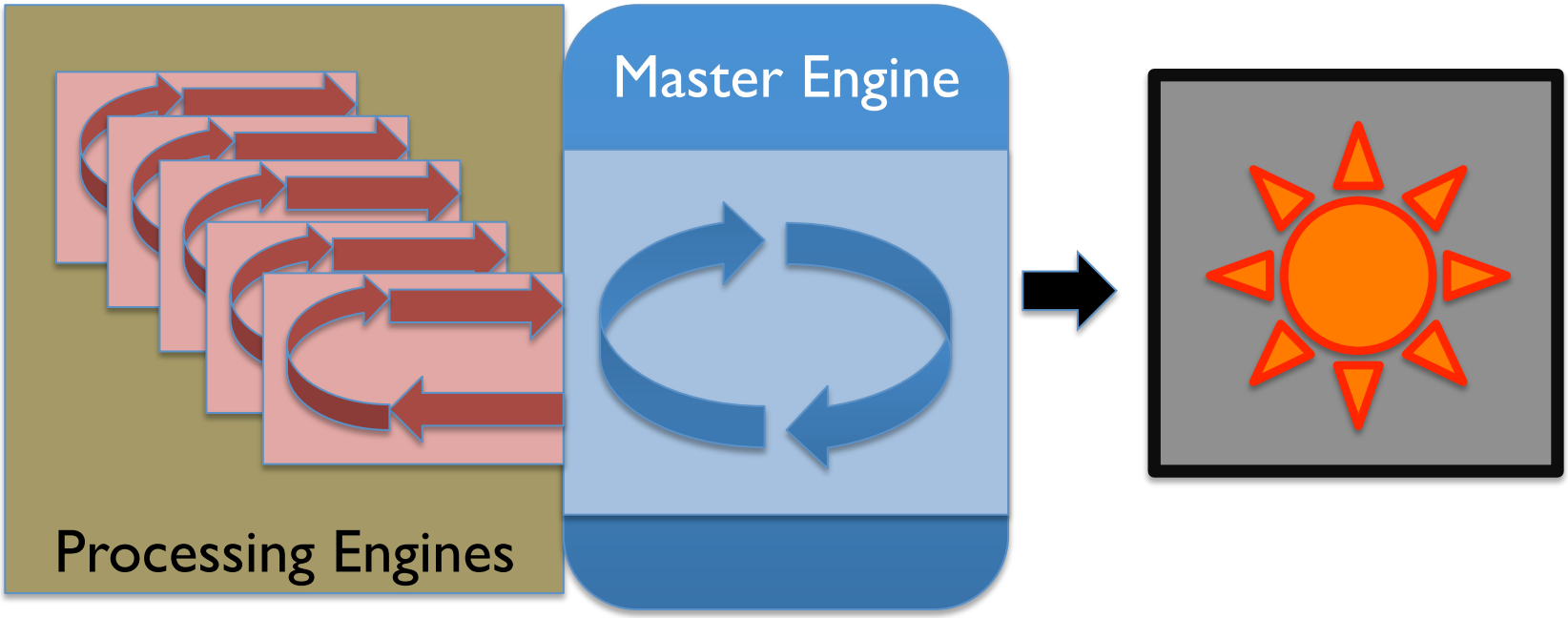
# Imaging




- Parallelization of imaging does not depend on how the data is partitioned in any way.
  - Partitioning is a simple way to distribute the data more efficiently within the Lustre file system.

# Continuum Imaging

- So far for continuum imaging we have only parallelized the major cycle.



# Running Continuum Imaging

- So far parallel clean is only available at the tool level.

```
>>> from parallel.pimager import pimager
>>> imager =pimager()
>>> imager.pcont(arguments)
```

- We plan to incorporate this into the task once a few more features have been enabled and after we have sufficient experience with users to be confident.

# Running Continuum Imaging

- Controlling the switch between major and minor cycles and how deeply to clean is fairly manual at the moment.

- The user specifies:
  - majorcycles integer number of CS major cycles to do
  - niter: maximum number of clean iterations to execute
  - threshold: string quantity of the residual peak at which to stop deconvolving
    - e.g. '0.1 mJy'
- Between each CS major cycles the minor cycle will fit niter/majorcycles components.
- If threshold is reached at any time the clean cycle will stop.

# Running Continuum Imaging

- *contclean*: Boolean specifying if we are continuing a clean or starting new
  - If false the imagename.model is deleted otherwise clean will continue from previous run

By setting *contclean* = True the clean process can be resumed to clean deeper or with a new mask.

# Running Continuum Imaging

The online help has definitions for the arguments many are the same as normal imaging:

- msname: input measurement set
- imagename: output image
- imsize: list of 2 numbers [nx,ny] defining image size in x and y
- pixsize: list of 2 quantities ['sizex', 'sizey'] defining the pixel size
  - e.g ['1arcsec', '1arcsec']
- phasecenter: an integer (field index) or a direction
  - e.g 'J2000 19h30m00 -30d00m00'
- field: field selection string
- spw: spw selection string

# Running Continuum Imaging

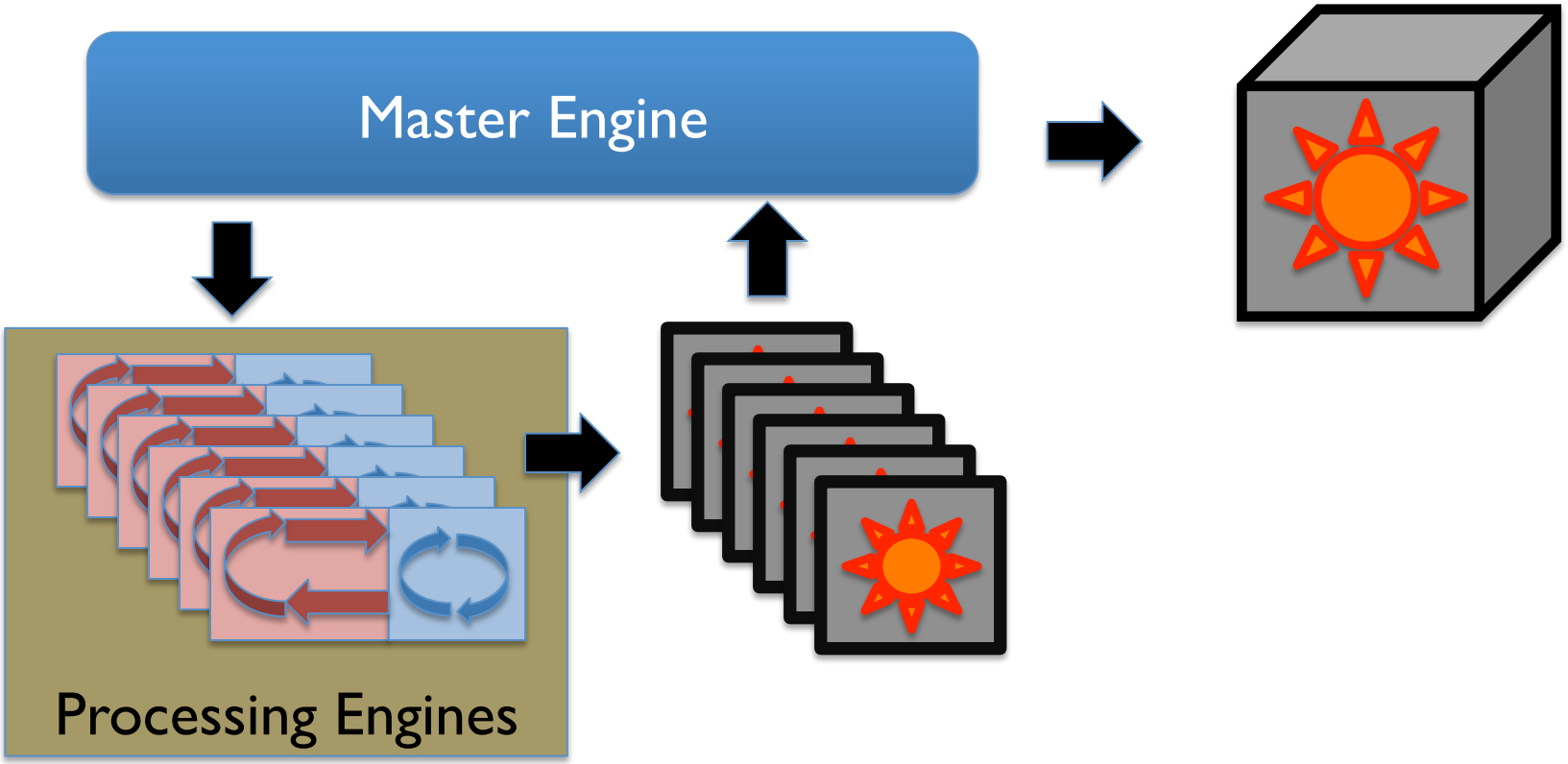
- stokes: string specifing imaging stokes
  - e.g 'I', 'IV
- ftmachine: the ftmachine to use (see Sanjay's lecture)
  - ft, wproject, mosaic NOTE: wproject and mosaic are still being tested
- facets: integer the number of facets to split image into.
- alg: string specifying the algorithm
  - Current possibilities are 'clark', 'hogbom', 'msclean'
- weight: string
  - e.g 'natural', 'briggs' or 'radial'
- robust: float valid for 'briggs'
- scales: scales to use when using alg='msclean'

# Running Continuum Imaging

- These parameters are either not yet fully implemented, have been deprecated, or are available for developer use.
  - hostnames: deprecated use '' (empty string)
  - numcpuperhost: deprecated use any integer
- visinmem: Boolean load visibility in memory for major cycles set to False
- These parameters are to support the work ongoing work on A-Projection, simply use the default values.
  - painc, cfcache, pblimit, dopbcorr, applyoffsets epjtablename

# Spectral Line Imaging

- Each plane of the output cube can be done independently.



# Running Parallel Spectral Imaging

```
>>> from parallel.pimager import pimager
>>> imager =pimager()
>>> imager.pcube(arguments)
```

- Like parallel continum clean there is no task (yet) for parallel spectral line imaging.
- Most arguments are the same as for the Continuum case
- The *maskimage* parameter is an input image mask. If it is not the same shape as the final cube a best guess interpolation is done.
- Interactive clean is not supported in this mode, suggestions are welcome.

# Running Spectral Line Imaging

- A new parameter has been introduced in the *pcube* method: *chanchunk*
  - *chanchunk* determines how many output channels are handled simultaneously by each of the processing engines.
  - The trade off here is between the amount of I/O and the memory footprint.

Serial Reduction: 351 min

*Once we have understood the heuristics we expect to handle this parameter in the Task.*

| | Number of Engines | | |
|---|---|---|---|
| **Chan Chunk** | 6 | 11 | 12 |
| 1 | 123 min | 98 min | 96 min |
| 4 | 73 min | 51 min | |
| 8 | 59 min | 43 min | |
| 16 | 57min | | |

# Resource Specification

- The computing hardware resources available to CASA are specified in a cluster specification file.
  - By default CASA assumes that it should use all of the cores available on the current system.
- Specification file looks like:

```
# <hostname>, <number of engines>, <working dir>
casa-dev-06, 4, /lustre/jkern/parallelTest/cont.large
casa-dev-07, 8, /lustre/jkern/parallelTest/cont.large
```

- You must be able to ssh without typing a password to all of the systems listed in your specification file.
  - If you do not have a specification file you must be able to ssh back to local host
  - See the NRAO goldbook for instructions on setting up ssh.

# Resource Specification

- In order to instruct CASA to use the resources in your cluster configuration file:

```
>>> from simple_cluster import simple_cluster
>>> simple_cluster().init_cluster(<clusterFile>)
```

- After these lines all tasks or tools which can make use of the parallelization will use these resources.

# Advanced Usage

- For experts there are some helper functions to assist in making use of the parallelization.
- JobData is a python class which encapsulates work to be done on a single engine:

```
>>> from simple_cluster import JobData
>>> myJob = JobData('flagdata', <Argument Dictionary>)
```

- JobQueueManager is responsible for executing a set of jobs on the cluster

```
>>> from simple_cluster import JobQueueManager
>>> queue = JobQueueManager()
>>> queue.addJob(myJob)
>>> queue.executeQueue()
```

  - This will return when all jobs have either succeeded or broken. Return state can be found from the queue object.
  - Check the online help for more details on using these classes.