CASA HPC

Intro, Overview, Future

Sandra Castro

Justo Gonzalez

Julian Taylor

ESO – Pipeline Systems Group

on behalf of the CASA team

ALMA Developer's Workshop

25-27 May 2016

Overview - Topics

- Why parallelise CASA?
- HPC project scope
- Parallelisation Concept
- Parallelisation Implementation
- Performance tests
- How to use CASA in parallel? - Justo
- Future

Why parallelise CASA?

- Many tasks require traversing the entire data set and are I/O limited.

  → flagdata, applycal, time averaging

- CASA must try to make the most efficient possible use of whatever resources are available. CASA has focused on 2 standard systems

  Workstation → multi-core system, local disk, single shared memory

  Cluster → many multi-core nodes, high performance network file system (Lustre), no shared memory access

CASA HPC Project Scope

- Define a parallelisation concept
- Implement a parallelization framework for task and tool levels. (Python and C++)
- Support the parallelisation of the Pipeline
- Improve performance on computing clusters and desktops.
- Provide documentation to users and developers

Parallelisation Concept

Trivial parallelisation

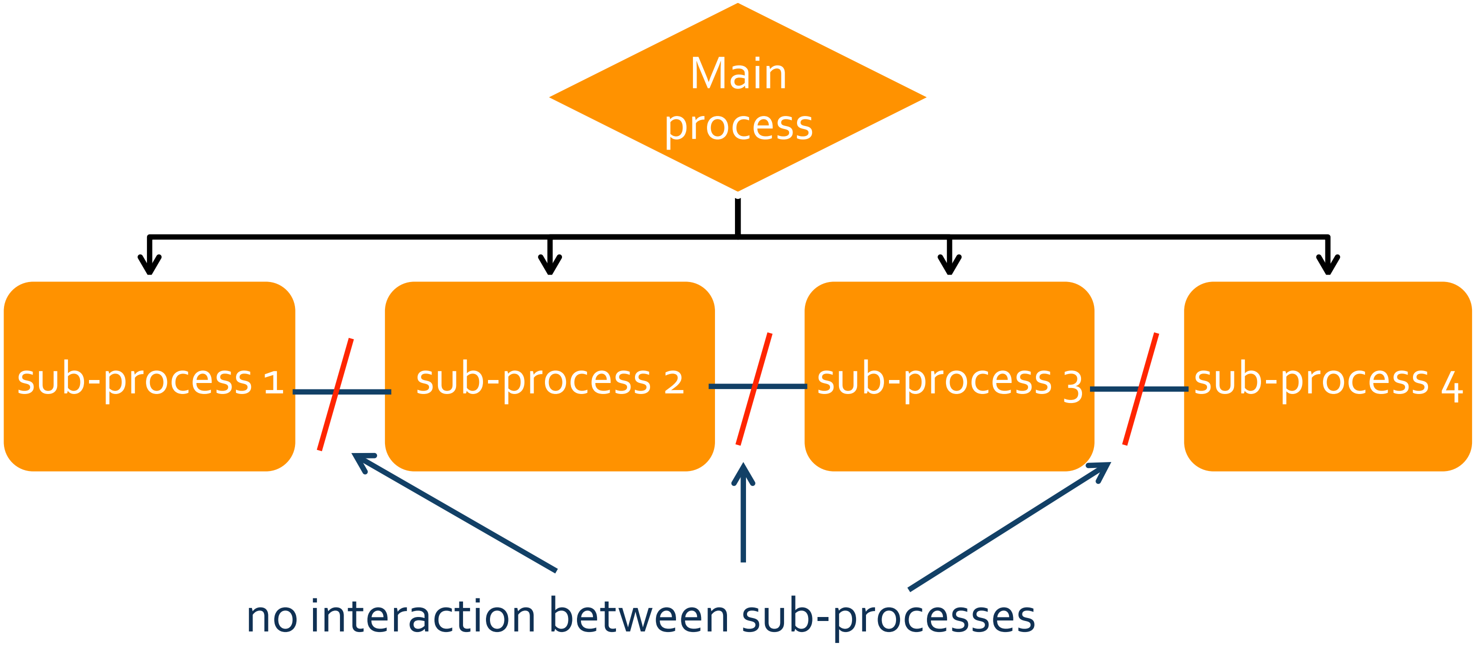
- Partition the MS into sub-MSs (spw, scan axes)
- Run a CASA instance on each sub-MS in parallel

→ partitioned data is called **Multi-MS or MMS**

→ partition task is the front-end to create a Multi-MS

also possible inside importasdm

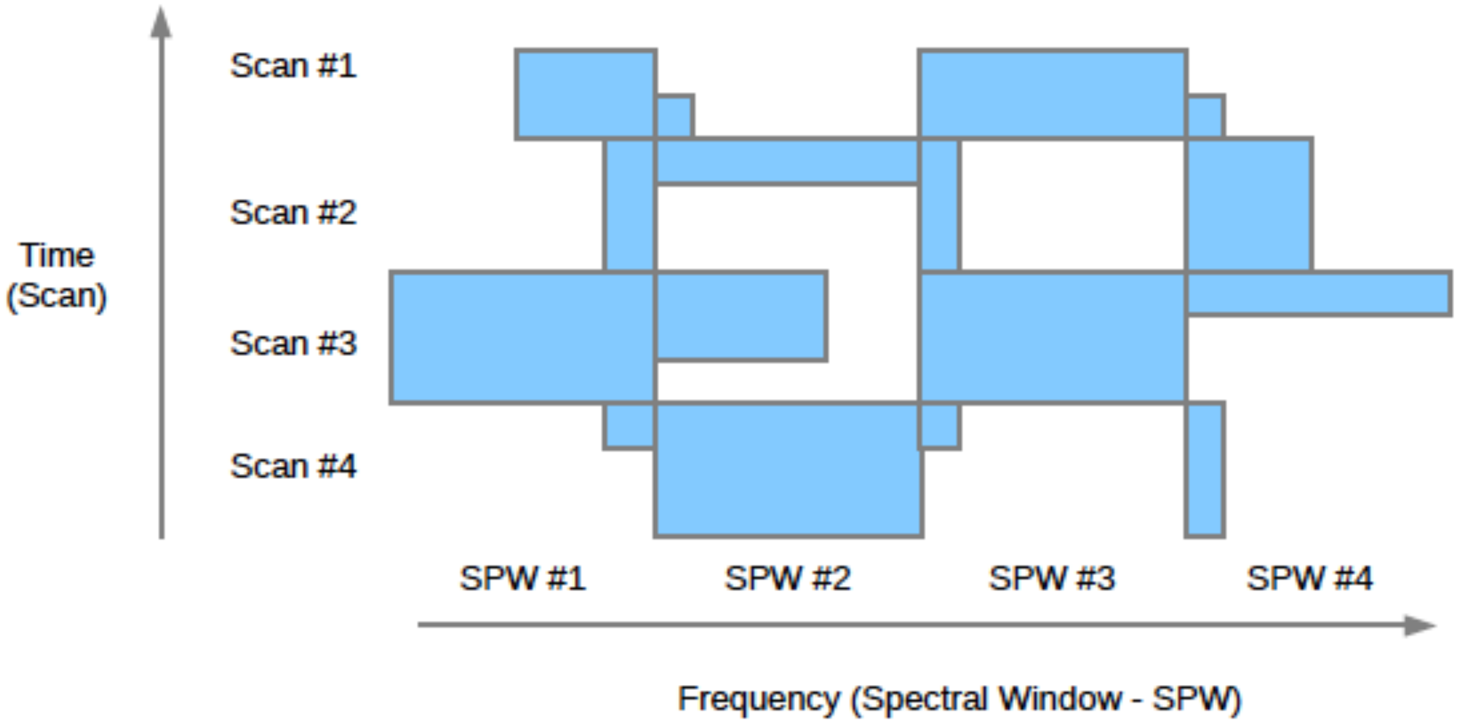
Trivial Parallelisation Principle



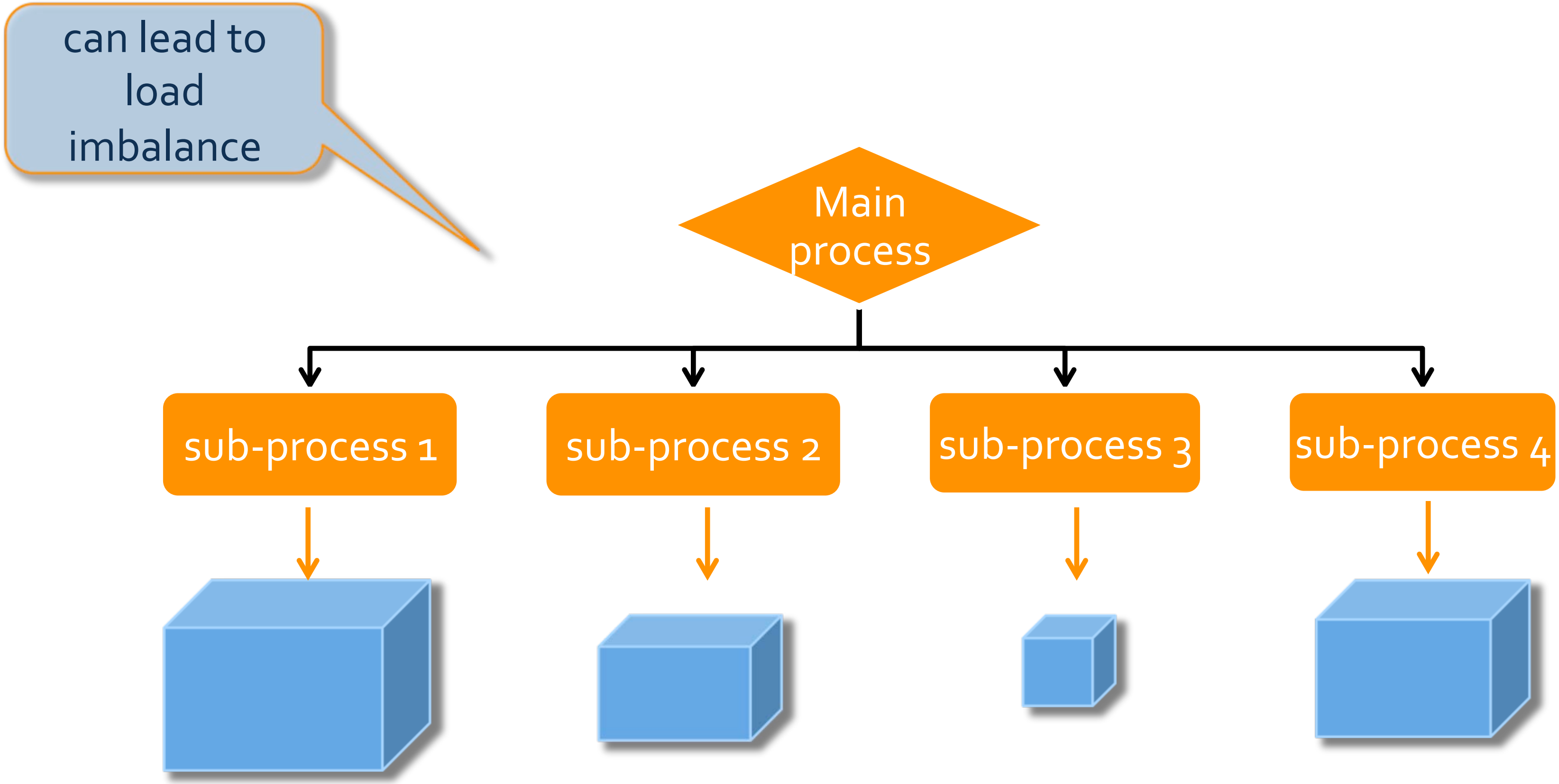
Theory: run-time parallel = $\frac{\text{run-time sequential}}{\text{\# sub-processes}}$

But…. the MS is an irregular grid

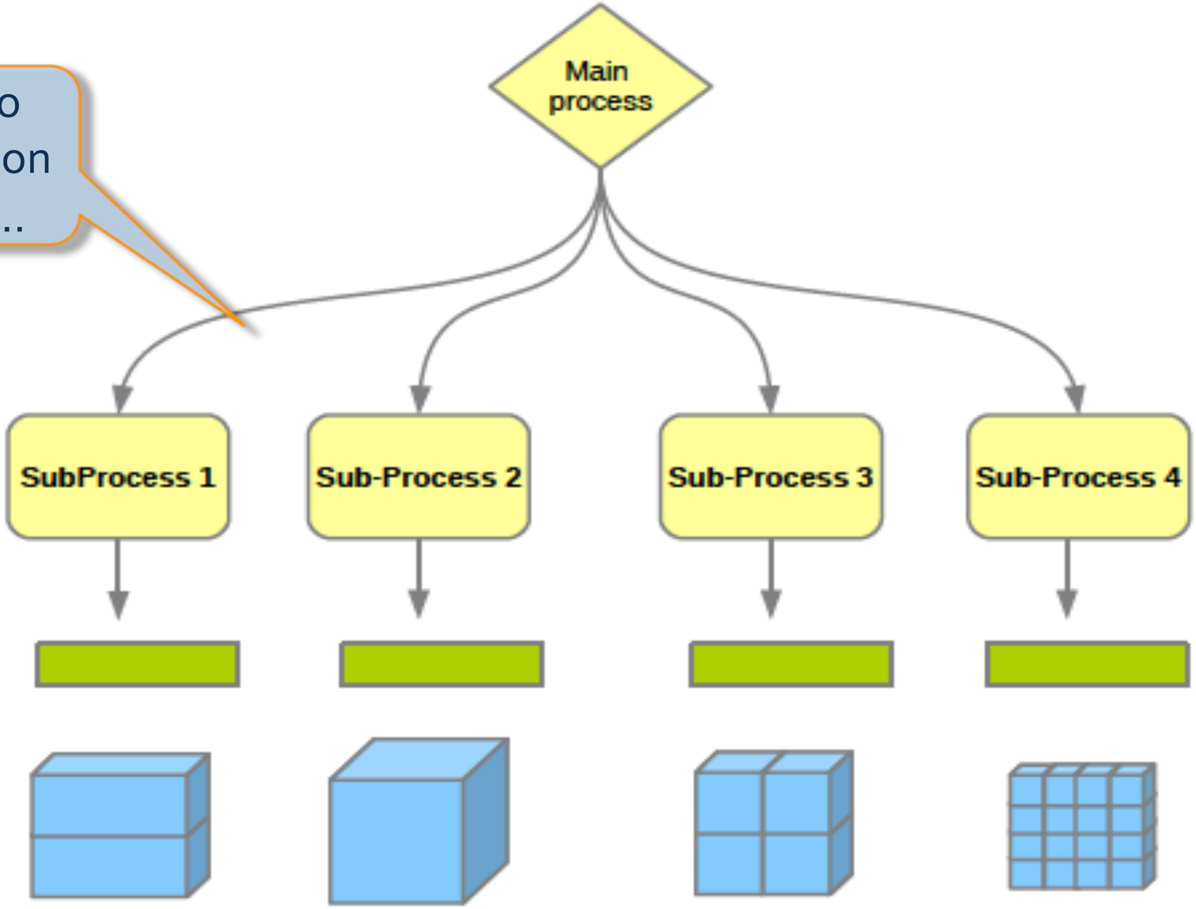
→ Problem for a simple partition per spw or scan



Multi-MS - load distribution



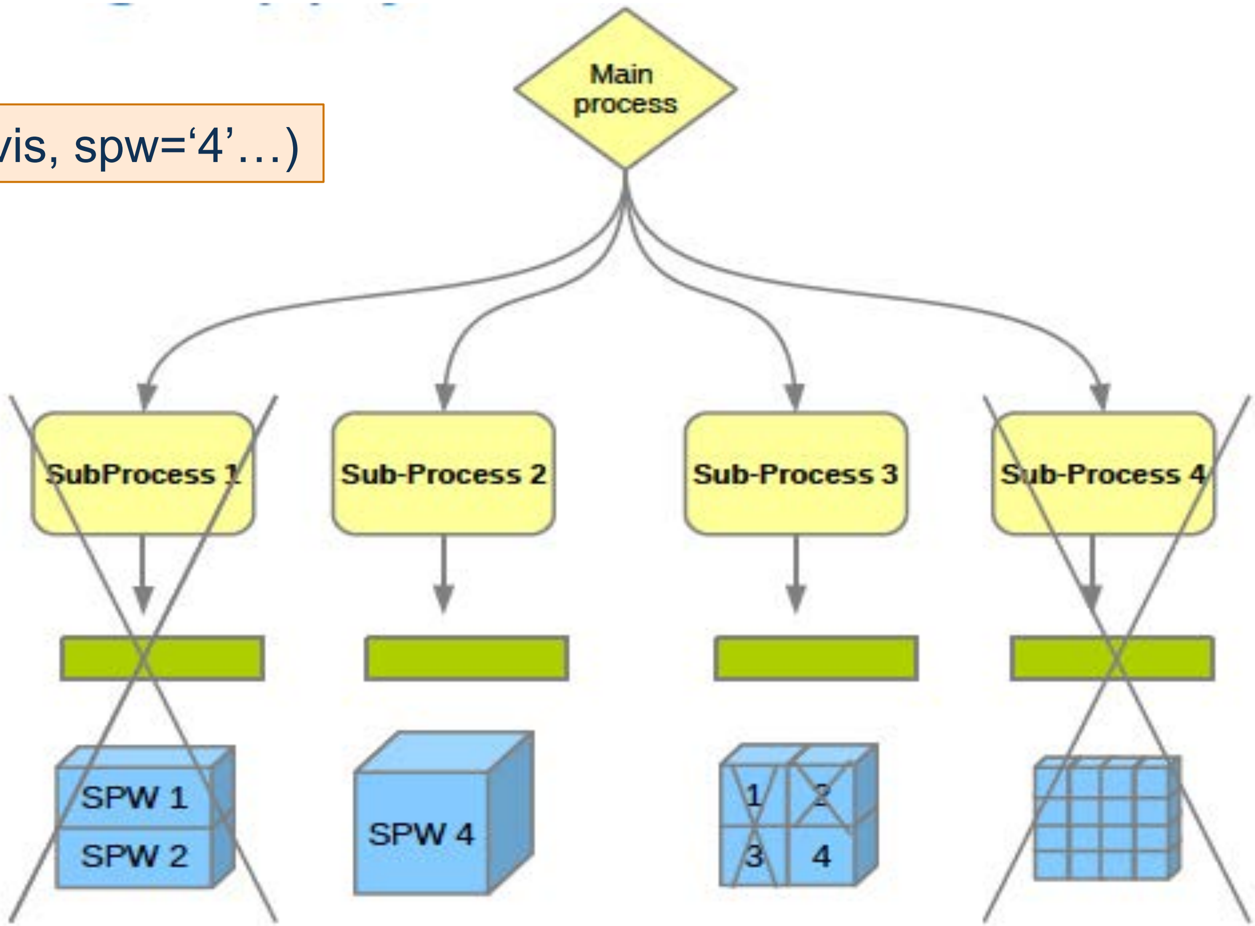
Partition - equal sized chunks



can lead to data selection problems...

Partition - data selection problem

applycal(vis, spw='4'…)



Partition - balanced mode - default

- Obtain the list of scan/DDI pairs
- Calculate the total number of visibilities per pair and sort the list in descending order.
- Each pair is allocated a separate Sub-MS following a global merit function
- **RESULTS in:**
  - each Sub-MS having roughly the same size in disk
  - the scan/spw content is spread in all Sub-MS
  - results in a better work-load for each parallel engine
  - tries to avoid idle engines when data selection is required

Implementation - framework

Parallelisation framework

mpi4casa → Gonzalez (2014)

- Uses the Message Passing Interface (MPI)
  - openMPI - MPI 3.0 standard
- Easy launching using custom **mpicasa** script
- Control of the number of processes at startup time
- Provide tools and documentation for developers
- Automated tests using Jenkins

Implementation - Tiers

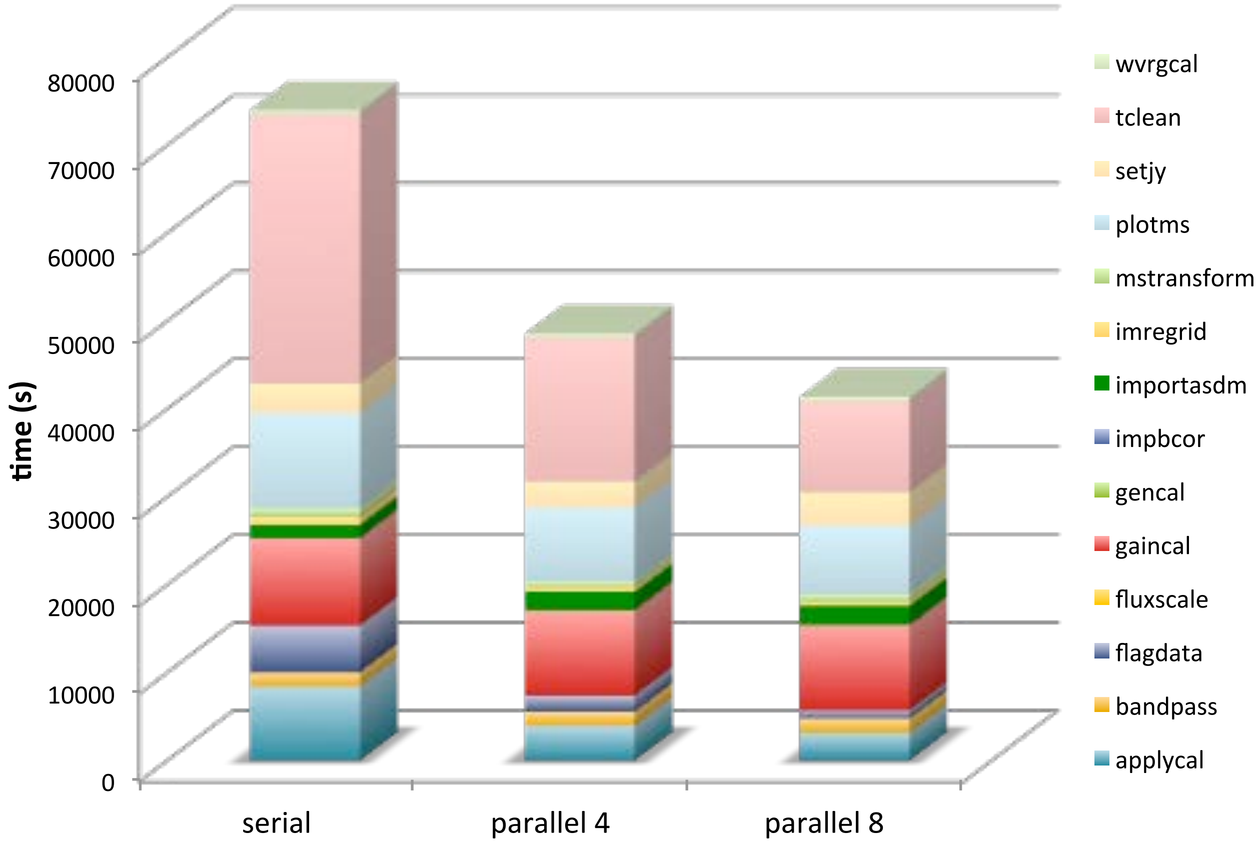
- Tier-1 Parallelisation
  - Internal parallelisation within tasks
    - → partition, split, flagdata, applycal, setjy, mstransform,
    - → will work in parallel, on each Sub-MS separately
- Tier-0 Parallelisation
  - Parallel execution of not internally parallelised tasks
    - → plotms, gaincal → see Multi-MS as a monolithic MS
- Tier-2 Parallelisation (future)
  - Parallel execution of internally parallelised tasks
    - → running several flagdata calls in parallel, each on an MMS

Support for the pipelines

IF and SD pipelines

- Tier-0 for plotms calls
- compression of online flags application
- spw-field breakdown in flagdata summary
- Tier-0 for baseline fitting
- baseline axis in partition
- I/O improvements

ALMA pipeline - performance



3 EBs ~ 115GB total

not all steps are parallelised

80+ calls to plotms and gaincal

flagdata is ~6x faster

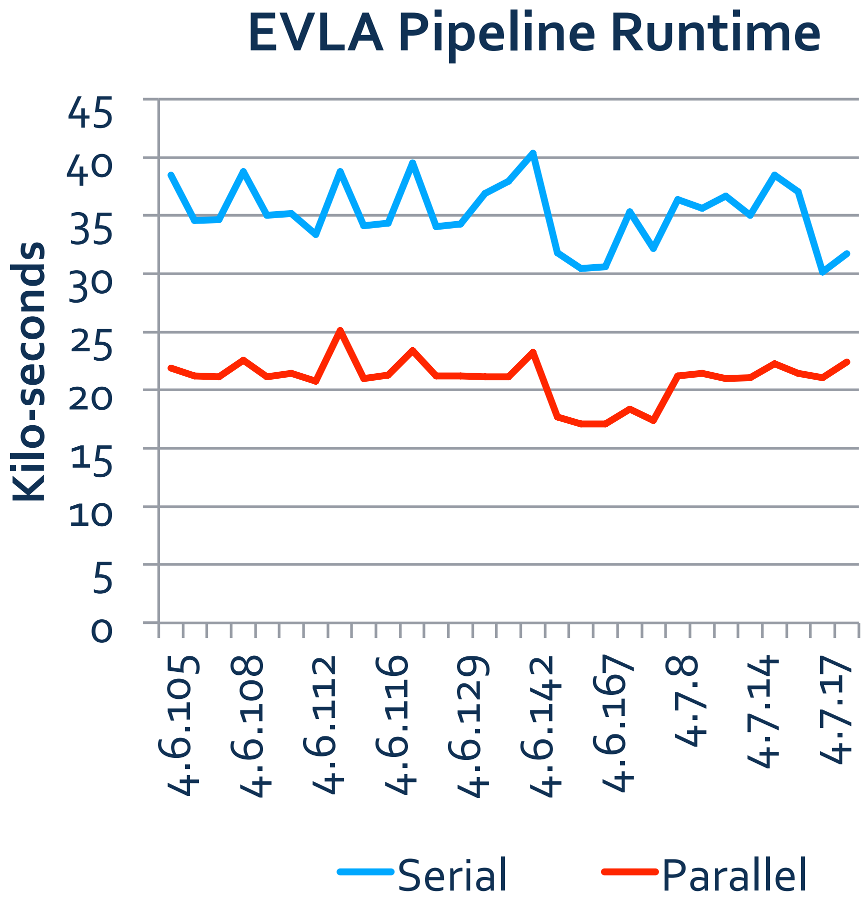
applycal and tclean are ~ 3x faster

tclean parallel is still under dev/test

EVLA pipeline - initial result

- VLA pipeline run in serial and parallel modes
  - 25 GB SDM
  - Parallel system using 8 cores

- When lustre space is available (~ July) will begin parallel testing on wide sample of data sets



Parallelization of imaging

- Parallel implementation of continuum and cube clean are fully integrated in tclean.
- tclean makes use of the MPI framework → similar to calibration tasks
- run time cost of imaging comes from two sources

  data I/O;

  re-sampling the data onto a grid (gridding and de-gridding)

- the run time reduces from many days to a few hours using tens of processes

source Bhatnagar et al. 2015

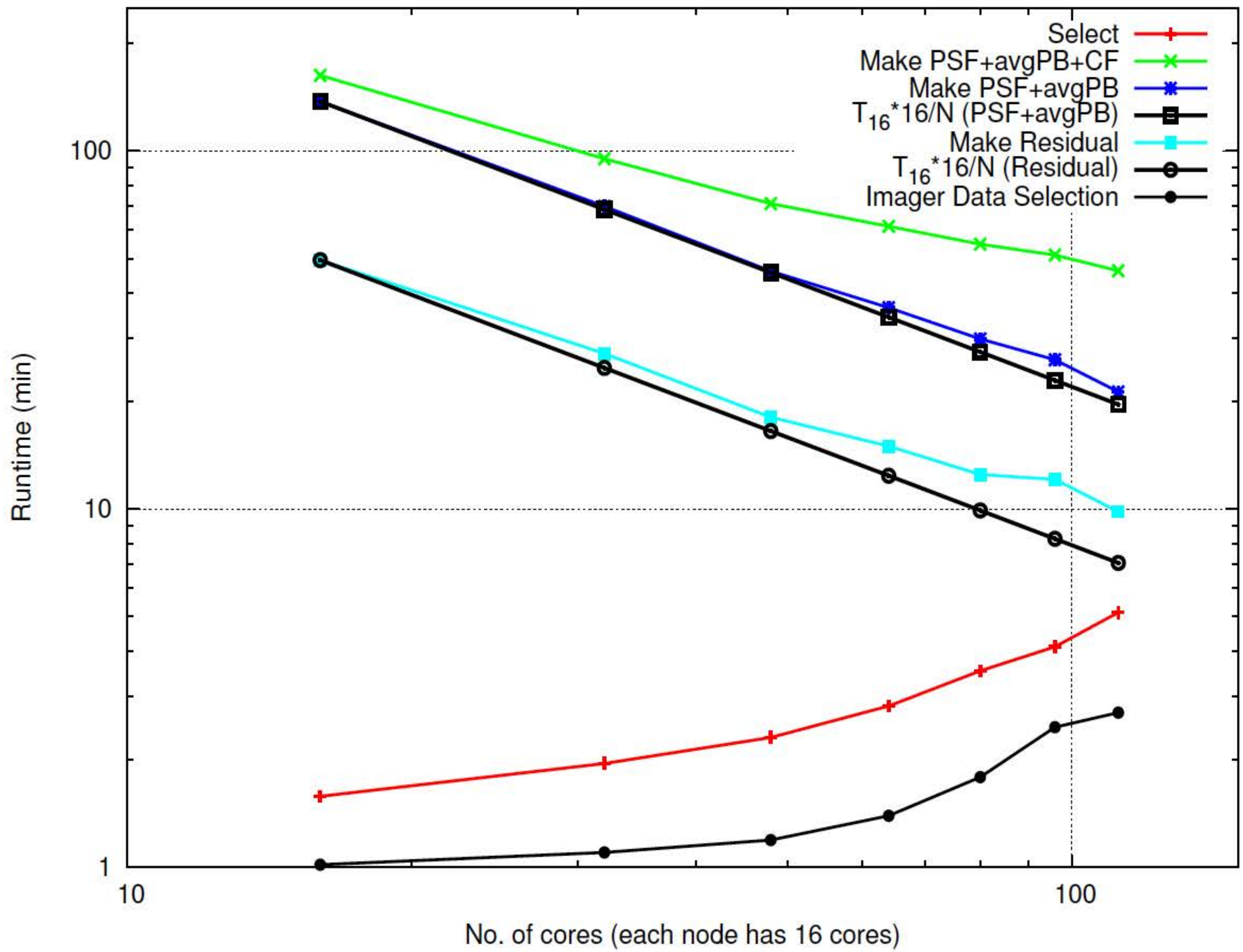


Figure 5: Run time performance for WB A-Projection for 2-term MT-MFS imaging. The curves labled "$T_{16} * 16/N$" show the theoretically expected scaling with number of cores. The curve labled "Imager Data Selection" is the time it takes for the C++ code to finish the data selection at each core.

Performance Considerations

- Ideally, use a shared high-performance file system for multi-node use and a strong I/O system.
- The type of processing done in the analysis.
- The size of the ASDM in order to decide if it is worth processing it in parallel or not.
- The size of the image and algorithm used will affect the memory consumption of tclean.

  channel chunks → under development for parallel case

Future development

- tclean parallelization (IF), imaging steps (SD)
- Tier-2 parallelisation or sub-clusters to process multiple EBs
- MPI at C++ level to support gaincal/bandpass in selfcal mode
- Resource identification/management. CASA must be able to identify the available resources in the system and use them efficiently at run time.

HPC Documentation

- Users Documentation
  - CASA cookbook 4.5+
    - Chapter 10 – Parallel Processing in CASA
    - Chapter 4 – Synthesis Calibration (mstransform)
  - Example script on how to run in parallel
    - alma-m100-analysis-hpc-regression.py

- Developers Documentation
  - CASA MPI Framework
  - Multi-MS Structure
  - Guide to running tests with Multi-MSs

Thank you

QUESTIONS?

CASA Parallelization Tutorial

Justo Gonzalez

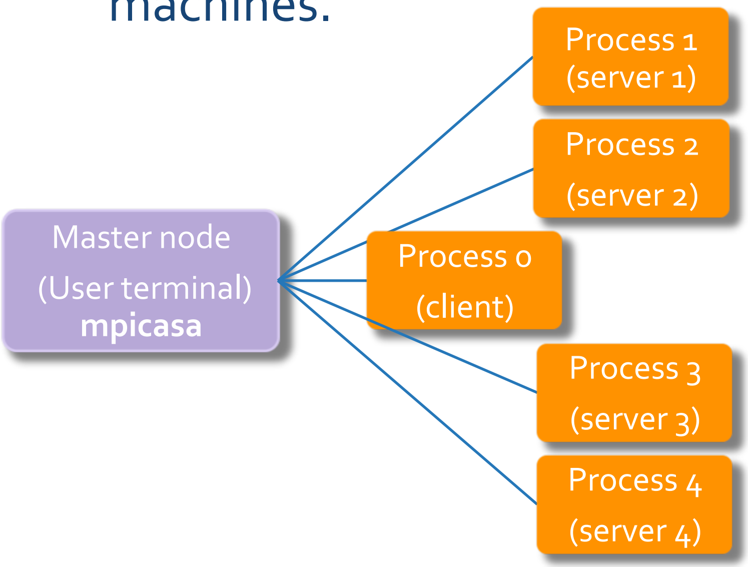
- Message Passing Interface (MPI)
- How to run CASA in parallel mode (mpicasa)
- Parallelization Interface (mpi4casa)
- Default CASA parallelization
  - Calibration
  - Imaging

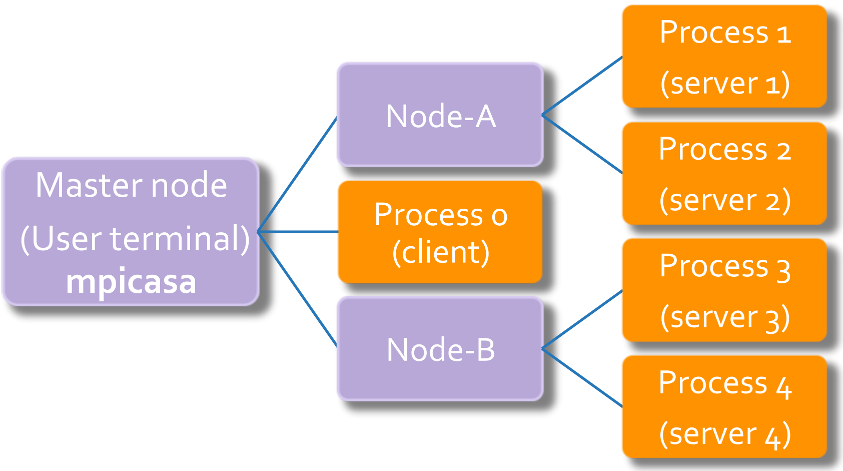
Message Passing Interface (MPI)

- MPI is a standard process communication interface
  - Good portability to several platforms / OSs
- Supported and elaborated by governmental programs (NSF, ARPA by USA and Espirit by EU)
  - Proper maintenance, good long-term choice

How to run CASA in parallel mode (mpicasa)

- To run CASA in parallel it is necessary to use a script included in the CASA distribution called mpicasa.
- mpicasa handles environment settings and spawns the required number of processes on the local host machine and/or on remote machines.





How to run CASA in parallel mode (mpicasa)

- Deploy processes only on local host
  - **mpicasa -n <number_of_processes> path_to_casa/casa <casa_options>**
    - number_of_processes: Number of processes to deploy

      (**number of Servers + 1 (client)**)
    - casa_options: CASA options such as: –nogui, –log2term, etc.

```
mpicasa –n 5 casa –nogui –log2term –c “myscript.py”
```

    - Batch mode: -c <script_name>

```
mpicasa –n 5 casa
```

    - Interactive mode: An xterm window pop-ups, necessary to log in with X11 forwarding

How to run CASA in parallel mode (mpicasa)

- Deploy processes only on local host
  - **mpicasa -n <number_of_processes> path_to_casa/casa <casa_options>**
    - number_of_processes: Number of processes to deploy (**number of servers + 1 (client)**)
    - casa_options: CASA options such as: –nogui, –log2term, etc.
    - Batch mode: -c <script_name>

```
mpicasa -n 5 casa -nogui -log2term -c “myscript.py”
```

    - Interactive mode: An xterm window pop-ups, necessary to log in with X11 forwarding

```
mpicasa -n 5 casa
```

- Deploy processes on remote machines
  - **mpicasa -hostfile <hostfile> path_to_casa/casa <casa_options>**
    - <hostfile>: Text file with one line per node, and the number of processes to be deployed.

```
# This is an example hostfile
node-A.example.com slots=2
node-B.example.com slots=2
```

CASA Parallelization interface (mpi4casa)

- MPI firstly introduced in CASA at the python level with the mpi4casa package (developed by Lisandro Dalcin, CIMEC)
  - Supports all MPI operations
  - Allows to communicate python objects
  - Low overhead, comparable with C (15 microseconds)
- CASA HPC group developed a layer on top of it using a client-server model, where:
  - Client is the master process, driving user interaction, and dispatching user commands to the servers
  - Servers are all the other process, running in the background, waiting for commands sent from the client side

CASA Parallelization interface (mpi4casa)

- Initialization
  - Import MPICommandClient form mpi4casa module

```
from mpi4casa.MPICommandClient import MPICommandClient
```

  - Create an instance of MPICommandClient

```
client = MPICommandClient()
```

  - Set logging policy

```
client.set_log_mode('redirect')
```

    - Redirect: Logging from all servers is redirected to the main log file
    - Separated: Logging from each server is sent to a separated log file
  - Initialize command handling services

```
client.start_services()
```

CASA Parallelization interface (mpi4casa)

- Syntax to send a command request

```
ret = client.push_command_request(command,block,target_server,parameters)
```

  - **command**: String containing the Python/CASA command to be executed. The command parameters can be included within the command in itself also as strings.
  - **block**: Boolean to control whether command request is executed in blocking mode (True) or in non-blocking mode (False). Default is False (non-blocking).
  - **target_server**: List of integers corresponding to the server ids to handle the command
    - target_server=None: The command will be executed by the first available server
    - target_server=2: The command will be executed by the server n#2 as soon as it is available
    - target_server=[0,1]: The command will be executed by the servers n #2 and #3
  - **parameters** (Optional): Alternatively the command parameters can be specified in a separated dictionary using their native types instead of strings.
  - **ret** (Return Variable):
    - In non-clocking mode: Integer (command id) to retrieve the command response at a later stage.
    - In blocking mode: List of dictionaries, containing the response parameters.

CASA Parallelization interface (mpi4casa)

- Syntax to receive a command result

```
ret = client.get_command_response(command_request_id_list,block)
```

  - **command_request_id_list**: List of Ids (integers) corresponding to the commands whose result is to be retrieved.
  - **block**: Boolean to control whether to block until all command results have been received
  - **ret** (Return Variable): List of dictionaries, containing the response parameters. The dictionary elements are as follows:
    - 'successful' (Boolean): indicates whether command execution was successful or failed
    - 'traceback' (String): In case of failure contains the traceback of the exception thrown
    - 'ret': Contains the result of the command in case of successful execution

CASA Parallelization interface (mpi4casa)

- Example 1
  - Run wvrgcal in 2 different measurement sets (for instance each one corresponding to an Execution Block):

```
# Example of full command including parameters
cmd1 = “wvrgcal(vis=‘X54.ms',caltable=‘cal-wvr_X54’,spw=[1,3,5,7])”
cmdId1 = client.push_command_request(cmd1,block=False)

# Example of command with separated parameter list
cmd2 = “wvrgcal()”
params2={vis=‘X54.ms',caltable=‘cal-wvr_X54’,spw=[1,3,5,7]}
cmdId2 = client.push_command_request(cmd2,block=False,parameters=params2)

# Retrieve results
resultList = client.get_command_response([cmdId1, cmdId2],block=True)
```

  - **target_server**: Is not specified because these are monolithic state-less commands, therefore any server can process them

CASA Parallelization interface (mpi4casa)

- Example 2
  - Use the CASA ms tool to get the data from 2 EBs and apply a custom median filter:

```
# Open MSs
client.push_command_request(“tb.open(‘x54.ms’)”,target_server=1)
client.push_command_request(“tb.open(‘x220.ms’)”,target_server=2)

# Apply median filter
client.push_command_request(“data=ms.getcell(‘DATA’,1)”,target_server=[1,2])
client.push_command_request(“from scipy import signal”,target_server=[1,2])
client.push_command_request(“filt_data=signal.medfilt(data)”,target_server=[1,2])

# Put filter data back in the MSs
client.push_command_request(“tb.putcell(‘DATA’,1,filt_data)”,target_server=[1,2])

# Close MSs
client.push_command_request(“tb.close(),target_server=[1,2],block=True)
```

  - **target_server**: Specified as each command depends on the state generated by previous ones
  - **block**: Block only on the last commands as all the others will be executed using a FIFO queue

Default CASA parallelization

- Calibration
  - If a Measurement Set is partitioned, and CASA runs in parallel mode, the following tasks trigger automatically internal parallelization:
    - flagdata, applycal, setjy, uvcontsub,
    - mstransform, split, hanningsmooth, cvel2, clearcal, delmod
  - To partition a Measurement Set there are two options (both run in parallel)
    - importasdm with option createmms=True

```
importasdm(asdm='uid_X54',vis='X54.ms',createmms=True, numsubms='auto')
```

    - partition (allows to specify desired data column)

```
partition(‘X54.ms,outputvis=‘X54.mms’,separationaxis='auto’)
```

- Imaging
  - If CASA runs in parallel mode tclean resorts to parallelization if parallel=True
  - It can work with normal MSs and parted MS, so there is no need to part the data