EVLA Memo #133
Parallelization of the off-line data processing operations using CASA

S. Bhatnagar, H. Ye and D. Schiebel

May 18, 2009

Abstract

This memo describes the work done towards parallelization of the most time consuming steps involved in calibration and imaging of large databases (∼100 GB or larger). The major data processing steps identified for parallelization were: simple initial flagging of the raw data, primary time and frequency calibration followed by image deconvolution (continuum and spectral line imaging) and SelfCal of multi-channel database. The initial effort was focused on parallelization using the Single Program Multiple Data (SPMD) paradigm where the data is distributed across a cluster of computers with local disks and the same (or similar) program is run on each node to process the piece of the data on the local disk.

A Python framework utilizing the CASA scripting interface for the parallelization of the CASA tasks was developed. CASA tasks `flagdata`, `gaincal`, `bandpass`, `applycal` and `clean` were parallelized using this framework. Using all the 16 nodes of the CASA development cluster, $10-20\times$ improvement in total run time compared to the single-node run time for end-to-end continuum and spectral line imaging was measured. Memory requirements for spectral line imaging are higher. The total available RAM when using all the 16 nodes of the cluster (16×8 GB) was sufficient to fit the run time memory buffers in the RAM, resulting in higher speed-up for spectral line case. For the parameters of the cluster hardware, this argues in favour of larger memory per CPU/node used for computing. It is therefore conceivable to design an affordable cluster where the run-time memory buffers required for the average data volumes and image sizes for NRAO telescopes will fit entirely in the cluster RAM.

1 Introduction

The major data processing steps identified for parallelization for processing large databases (of size ∼ 100GB or larger) were: simple initial flagging of the raw data, primary time and frequency calibration followed by image deconvolution (continuum and spectral line

imaging) and SelfCal of multi-channel database. Single-node run time analysis shows that the off-line processing using CASA applications was ~ 30× longer than the total time of observation (EVLA Memo 132). The total *effective* I/O was ~ 1 TB (for a ~ 100 GB database) and the post-processing time was limited by the disk I/O rate. Consequently this initial effort was focused on parallelization using the Single Program Multiple Data (SPMD) paradigm where the data is distributed across a cluster of computers with local disks and the same (or similar) program is run on each node to process the piece of the data on the local disk.

A Python framework utilizing the CASA scripting interface for the parallelization of the CASA tasks was developed. CASA tasks `flagdata`, `gaincal`, `bandpass`, `applycal` and `clean` were parallelized using this framework. Using all the 16 nodes of the CASA development cluster, the wall-clock run-time was measured to be 15 min. for flagging, ~ 20min. each for time and frequency (self-)calibration and 2 and 1.2 hours for spectral line and continuum imaging respectively. The total run time for end-to-end continuum and spectral line imaging was in the range ~ 2.5 and ~ 3.5 hr respectively, compared to ~ 22 and ~ 74 hr when using a single node for post processing. This corresponds to ~ 10 − 20× improvement in total run time for end-to-end continuum and spectral line imaging compared to the single-node run time.

Section 2 describes the parameters of the database used for this work. The results from the parallel execution of the various tasks on a 16-node cluster is given in Section 3. Section 4 describes the software framework used for the parallelization of the various CASA tasks used for data distribution and parallel post processing. Finally, Section 6 has discussion for further work and investigations of issues identified in this work.

2 Data Description

The parameters of the simulated data used for these tests are given in Table 2 (see EVLA Memo 132 for details). The simulated database corresponds to ~ 86 GB worth of observed data on two fields. One of the fields is a calibrater field (with a single point source at the phase center) and the other is the target field. Both fields were cyclically observed in all 32 spectral windows simultaneously, with a dwell time of 2 min and 30 min for the calibrater and target field respectively. The total disk space required during processing, including disk space for scratch columns, was ~ 266GB[1].

3 Wall-Clock time for parallel post-processing

The CASA Development cluster was used for timing parallelized CASA applications. Each node of this 16-node cluster has 2 Quad-core Intel Xeon E5420 processors running at 2.5 GHz, each with a cache size of 6144 KB (L2 cache of 1.2MB), 8 GB of RAM

[1] Work in CASA is currently in progress to eliminate the scratch columns where not necessary. However note that scratch data on the disk is required during processing in all packages, including CASA.

Table 1: Parameters used for data simulation

λ	6cm	the wavelength of observation
N_{ant}	27	Number of antennas
B_{max}	12.5 Km	Max. baseline length
N_{ch}	1024	Number of channels
N_{spw}	32	No. of Spectral Windows
N_{pol}	4	Number of polarizations
T	2h	Total length of observations
ΔT	1s	Integration time
N_{Fields}	2	No. of fields observed

per node and 1.8 Tera byte RAID-1 local disk. All the nodes are interconnected with a 1 Gigabit non-blocking Ethernet switch.

The entire data base was spread across 16-nodes of the cluster as described in Section 4. A parallelized version of the CASA tasks were run on the cluster using the software framework, described in Section 4. The run time for data flagging, continuum and spectral line imaging and Self-Cal are given below. Summary of the run times is given in Table 2. Image size in RA and Dec for both kind of imaging was $1K \times 1K$ pixels. Spectral line imaging cube had 1024 channels along the frequency axis of the image cube. The Cotton-Schwab Clean algorithm using the `ft` algorithm for forward and reverse transforms was used for image deconvolution.

Table 2: Summary of the wall-clock run time for parallelized tasks.

Operation	16-node run time	1-node run time
Flagging:`quack` only	15^m	23^m
Flagging: clipping	15^m	66^m
G-Jones solver (SelfCal)	20^m	1^h54^m
B-Jones solver (SelfCal)	20^m	1^h53^m
Data correction	15^m	2^h23^m
Imaging: Spectral line	2^h	64^h
Imaging: Continuum	1.2^h	12^h
Total	$\sim 2.5^h - 3.5^h$	$\sim 74^h - 22^h$

3.1 Data Flagging

The CASA task `flagdata` was used in two modes: (1) `quack` mode, and (2) `manualflag` mode. Due to on-line software latencies, often the initial few integrations worth of data in each scan is bad. The `flagdata` task in the `quack` mode flags data corresponding to a user defined length of time starting from the beginning of each scan. The `manualflag` mode of `flagdata` is used to flag data with amplitude outside a user defined range. This typically removes strong RFI which is often sufficient - particularly at higher frequencies.

The `flagdata` in the `quack` mode and `manualflag` mode took $\sim$ 15 min each. The `quack`ing time[2] was set to 1 min. The run time in the `quack` mode has weak dependence on the `quack`ing time. `manualflag` run time is independent of the amplitude range for flagging.

3.2 Imaging: Spectral line imaging

Image deconvolution can be done entirely in the image domain (Högbom Clean) or by iteratively reconciling the model image with the data (major-minor cycle based Clean, e.g. CS-Clean). The former requires a single read and gridding of the data to compute the images and the PSF, while the latter requires reading, gridding and data prediction in each major-cycle. CS Clean is however almost always more accurate and is more commonly used. We therefore used the CS-Clean algorithm for spectral line and continuum imaging.

Ignoring a few details, spectral line imaging is an embarrassingly parallel application with no serial component for the image deconvolution iterations. Each node runs a full spectral line deconvolution algorithm to convergences using the piece of data on its local disk. The only serial part is in combining (if required) the image cubes from each node to make a single image cube. The fraction of time for this last operation is however much smaller than the total time required for image deconvolution using 16-nodes.

In the 16-node tests, each node has 2 spectral windows with 32 channels in each. A 64-channel spectral line cube imaging was done at each node and later combined to make the final image cube. This entire operation took $\sim$ 2 hours of wall-clock run time.

Spectral line detection experiments also involves the operation of removing continuum emission. A model corresponding to a continuum image is first subtracted from the data. Any residual spectral-baseline is removed by either fitting and subtracting a spectral-baseline fit from the spectrum in the image domain (using the `IMLIN` kind of algorithms) or in the visibility domain (using `UVLIN` kind of algorithms). While continuum subtraction was ignored in these tests, its runtime cost is expected to be comparable to a single major-cycle of the continuum imaging operation (in addition to the cost of computing the continuum image).

3.3 Imaging: Continuum imaging

The computation for the χ^2 is the same as for spectral line imaging and is an embarrassingly parallel operation. Computing the update direction for iterative deconvolution involves computing residual images using the pieces of the data available at the local disk at each node, and averaging the residual images from all the nodes. This involves communication of residual images from each node to one node and was done using the the 1-Gigabit non-blocking cluster interconnect. A image plane deconvolution is done on the final continuum residual image. This constitute the serial part of the continuum imaging

[2] Starting from the beginning of the scan, all data corresponding to `quack`ing time is flagged in the `quack` operation.

process.

The 16-node run time for continuum imaging with 5 major cycles was ~ 1.2 hours. The embarrassingly parallel operations were done using the CASA task `clean` while the serial operations were implemented in the parallelization framework. For the image sizes used in these tests, the fraction of time for data communication over the inter-connect network and the serial operations were much smaller compared to the parallel operation. This ratio will however be a function of the data and image sizes as well as the number of nodes.

3.4 Calibration: Solving for G- and B-Jones

Self-Calibration is defined as solving for multiplicative antenna based gains as a function of time and/or frequency using a model for the image derived from the image deconvolution step. While the algorithm used for this and primary calibration is the same, self-calibration test was done for the target field using the image model derived from the imaging step as described above.

The 16-node run time to solve for G- and B-Jones terms was ~ 20 minutes each. G-Jones terms (gains as a function of time) were solved on a time scale of 2 minutes while B-Jones (gains as a function of frequency) were solved on a time scale of 20 minutes. Run time for primary calibration was insignificant compared to the total post-processing time.

3.5 Calibration: Data correction

This step involves computation of the G- and B-Mueller matrices and correcting the measured data by multiplying it by these Mueller matrices. This involves one full read *and* write of the data. The computational cost of these steps are relatively small.

The 16-node run time for this operation was ~ 30 minutes.

4 A `Python` based parallelization framework

The fundamental interface to the post-processing algorithms implemented in the CASA package is via the CASA toolkit (a Python binding for the underlying C++ code). A layer of Python framework on top of the toolkit provides the tasking interface. While more restrictive than the toolkit interface, traditional users use the tasking interface. Hence it was decided to build the parallelization framework using the tasking rather than the toolkit interface.

The tasking framework is implemented in the `IPython` script `casapy.py` which is loaded in an `IPython` interpreter via the program `casapy`. Since CASA tasks cannot be run as user process independent of the user interface, for remote execution it is required

that a copy of `casapy` be run on each cluster node, blocked on its input stream. A mechanism to send CASA tasking interface commands to these remote `casapy` processes and to receive the output of these commands is also required. The `IPython engine` and `IPython Controller` interfaces were used for this.

For parallel execution of the tasks, the following is assumed for the cluster configuration:

- password-less `ssh` access to all nodes
- NFS access to the local disks at each node. This was required for initial loading of the data[3] and for the relatively small data communication (compared to the total data volume) during processing (see section 6).
- same binary copy of the Python, `IPython` and the CASA packages is executable from each node of the cluster
- while not necessary, it is helpful to have some systematic convention for node names and local file system at each node (similar to the naming convention used for the CASA Development Cluster).

The Measurement Set (MS) is split along the frequency axis at spectral window boundaries using the `psplit()` function in the parallelization framework[4]. Given the number of nodes and base name of the directories on the local disks of each node, this uses the CASA task `split` to equally distribute the available channels and/or spectral windows across the nodes. The naming convention used for naming the individual pieces on the MS is implemented in the `mkname()` function which takes the node ID, node name, base directory, base file name and extension strings to construct the name of the local MS at each node. The same naming convention is then used later for the parallel execution of the tasks.

The `IPython` engine (`ipengine`) is a Python instance that takes Python commands over a network connection. This can handle incoming and outgoing Python commands (and even Python objects) over a network connection. Any number of instances of `ipengine` can be started. The `IPython` Controller (`ipcontroller`) provides an interface to communicate with multiple instances of `ipengine`. It internally maintains the connection with multiple `ipengine`s and manages the book-keeping required for the incoming and outgoing traffic per `ipengine`. It therefore provides a single-point blocking or non-blocking interface to a network of *ipengine*s using which commands can be sent to all or selected `ipengine`s, data received from `ipengine`s, etc. The `ipengine` and `ipcontroller` are user level independent processes which setup the communication between each other by exchanging the required connection information via a designated disk file (the `furl` file). The `MultiEngineCleint` Python class provides the Python scripting interfaces to the `ipcontroller` process. The `ipengine`s are started on the nodes using `ssh` and a

[3]For data-loading, a non-NFS based parallel data filler is possible.

[4]Data splitting at spectral window boundaries is convenient, particular for calibration. In principle, data can be split at channel boundaries rather than spectral window boundaries.

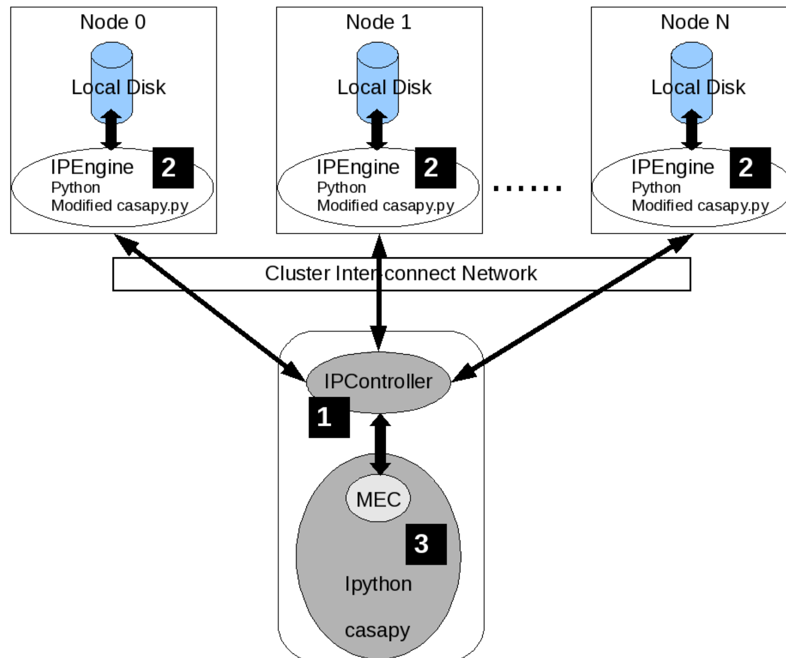

Figure 1: Figure showing the hardware configuration of the cluster and the software framework for running CASA applications on the cluster. The rectangular boxes represent computers running the Linux OS. The numbers in the black boxes represent the sequence in which various processes must be started: `IPController` on the user computer is started first, followed by the remote `IPEngines` on all the nodes of the cluster. Standard `casapy` is then started on the user computer and the `MultiEngineClient` (MEC) instantiated. MEC is then used to load the modified `casapy.py` in the remote `IPEngines`.

single `ipcontroller` is started on the user computer[5]. The CASA Parallelization framework is implemented in the `cluster.py` script (see Appendix B) which is loaded in the `casapy` process started on the same computer on which `ipcontroller` is run. The network of processes, the sequence in which they must be started and the relationship between them is shown in Fig. 1.

The CASA interface to the cluster (after the `ipengines` and `ipcontroller` have been started) is initialized by a call to the function pinit(). This returns a `MultiEngineClient` (MEC) Python object which is connected to all the remote `ipengines` and forms the primary interface to the cluster nodes. The `startcasapy()` function uses the MEC object to load a modified version of the `casapy.py` script[6]. If no exception is generated, all the nodes are now ready and CASA tasking interface commands can be sent to the nodes using the MEC object.

For the parallel execution of CASA tasks, first the value of the relevant task variables is set in the remote `casapy` processes. There are two classes of task variables - ones which have the same value at all nodes and the ones which potentially have different values

[5]This is currently done using a Linux shell script (see Appendix A). A more sophisticated Python interface for starting the remote `ipengines` is available (which also uses `ssh`).

[6]Modified to execute in `Python` rather than in `IPython`.

at each node. The first class of variables are set by specifying the variables and their values in a disk configuration file (this is typically the `<task>.last` file) and loading this file in each of the remote `ipengines`. The convention followed for setting the value of the second class of variables assumes that the values per node can be derived from a combination of the node name, the node ID (as reported by the MEC object) and some user defined values/strings. E.g., the name of the local input and output file names are formed by concatenating a user defined root directory name (`rootdir`), node ID, user defined work-directory (`workdir`), base name (`basename`) and extension (`ext`) strings. A set of functions in `cluster.py` take the user configuration file and the variables as input, to set up all the task variables at each node. These functions also have a list of parameters which can be used to over-write the values of the task variables set via the configuration file. The value of these parameters are set to default values such that they have no effect unless explicitly set by the caller of these functions.

Once the remote variables for the task have been setup, they can be examined using the command

```
MEC.execute("inp('<taskname>')")
(note that all the quotes are required)
```

Finally, the command

```
MEC.execute("go('<taskname>')")
```

can be used to start the execution of the remote tasks. This command will block till all the remote tasks finish execution.

5 Run-time monitoring

The framework and the parallelization of the CASA applications described here will typically be run either in the batch mode or in a post-processing pipeline. User monitoring of the run-time progress is therefore required. Typically, two levels of monitoring is done to determine progress and performance of the software: (1) monitoring the hardware parameters and its utilization (memory utilization, CPU utilization, etc.), and (2) monitoring the log messages from the software.

5.1 Node activity monitoring using `Ganglia`

For monitoring the cluster hardware parameters, the `Ganglia`[7] system of software was used. `Ganglia` is a widely used system and is scalable from small cluster of loosely coupled computers to large grids of clusters/computers. It is well supported, well documented and a mature software used at many super computing facilities.

[7] `http://ganglia.info`

A minimal system consisting of the `gmond` daemon and `gstat` program to query and display the in-memory real-time XML database of hardware parameters can be used to get a short-n-quick summary of the CPU usage of all the nodes as a function of time. See Fig. 2 for a screenshot of the display of the output of the `gstat` program.

```
Atlas
File  Edit  View  Terminal  Tabs  Help
casa-dev-01.aoc.nrao.edu     8 (    0/  202) [  1.64,  0.69,  0.27] [   0.0,   0.0,   1.1,  96.0,   2.8] OFF
casa-dev-02.aoc.nrao.edu     8 (    2/  167) [  2.56,  1.15,  0.46] [  12.2,   0.0,   0.4,  87.1,   0.4] OFF
casa-dev-03.aoc.nrao.edu     8 (    1/  164) [  3.76,  2.09,  0.80] [   2.3,   0.0,   1.7,  79.4,  16.6] OFF
casa-dev-04.aoc.nrao.edu     8 (    1/  164) [  4.80,  1.82,  0.66] [  11.9,   0.0,   3.0,  76.1,   9.0] OFF
casa-dev-05.aoc.nrao.edu     8 (    0/  169) [  6.20,  2.40,  0.88] [   2.0,   0.0,   1.4,  77.7,  18.9] OFF
casa-dev-06.aoc.nrao.edu     8 (    1/  175) [  4.32,  2.32,  0.91] [  12.2,   0.0,   2.4,  69.8,  15.7] OFF
casa-dev-07.aoc.nrao.edu     8 (    1/  164) [  1.99,  1.05,  0.40] [   1.6,   0.0,   3.7,  85.2,   9.5] OFF
casa-dev-08.aoc.nrao.edu     8 (    1/  166) [  1.44,  0.68,  0.25] [  12.2,   0.0,   0.3,  87.1,   0.4] OFF
casa-dev-09.aoc.nrao.edu     8 (    0/  173) [  9.16,  3.62,  1.33] [   0.3,   0.0,   1.8,  74.8,  23.1] OFF
casa-dev-10.aoc.nrao.edu     8 (    0/  165) [  2.02,  0.90,  0.33] [   0.0,   0.0,   0.1,  99.5,   0.5] OFF
casa-dev-11.aoc.nrao.edu     8 (    0/  167) [  2.83,  0.94,  0.33] [   0.1,   0.0,   0.3,  85.7,  14.0] OFF
casa-dev-12.aoc.nrao.edu     8 (    0/  166) [  2.13,  0.71,  0.25] [   3.1,   0.0,   0.6,  86.2,  10.2] OFF
casa-dev-13.aoc.nrao.edu     8 (    0/  163) [  4.50,  1.64,  0.58] [   3.0,   0.0,   1.8,  81.2,  14.0] OFF
casa-dev-14.aoc.nrao.edu     8 (    0/  166) [  2.59,  1.54,  0.61] [   0.1,   0.0,   1.2,  92.5,   6.2] OFF
casa-dev-15.aoc.nrao.edu     8 (    1/  174) [  3.44,  1.71,  0.81] [  12.2,   0.0,   0.3,  77.7,   9.8] OFF
casa-dev-16.aoc.nrao.edu     8 (    0/  164) [  0.23,  0.12,  0.04] [   0.3,   0.0,   0.6,  99.1,   0.1] OFF
```

Figure 2: Figure showing a screenshot of the display of the output of the `gstat` program which query the Ganglia `gmond` daemon. Each row contains the name of the node, number of CPUs, (active processes/Total number of processes), [1-,5-, 15-minute] average CPU load, and the fraction of the CPU time spent in User, Nice, System, Idle, I/O-wait states.

The full `Ganglia` system with a web-based interface is also installed for the CASA Development Cluster. Its use for monitoring the hardware, even by the "standard" users is highly recommended (see Fig. 3 for a screenshot).

5.2 Multiple `casapy.log` file monitoring using `multitail`

The framework described above for running parallel CASA tasks uses the SPMD paradigm for cluster computing. An independent `casapy` process is started per CPU/core used for parallel computing and I/O. Each of these processes write an independent log on the disk. With a largish number of such processes used in the parallelization framework, monitoring the application log files becomes an issue.

The `multitail`[8] program allows display of multiple ASCII log files in a manner similar to the use of the standard Unix command `''tail -f''`. The console screen is divided into a number of sections with a single log file displayed in each section (the topology of how the screen is divided is also user specified; see Fig. 4 for an example screenshot). Combination of this and use of small fonts allows monitoring of 10-20 log files simultaneously. However, since the number of CPUs/cores used can be much larger (e.g., each node typically has 2 CPUs with 4 cores each), the number of log files can be quite large and this solution does not scale well beyond 10–20 log files. A more complete survey for identifying a better technology is required.

[8] `http://www.vanheusden.com/multitail`

Figure 3: Figure showing a screenshot of the web interface to Ganglia. Top part of the screen shows an overview of the cluster resource usage. The grid of plots in the bottom part of the screen shows the CPU usage as a function of time for each of the 16 nodes.

6 Discussion

The goals of the work described in this memo were: (1) define and help in designing a development cluster, (2) identify the software technology and build a framework for deploying parallel applications for post-processing needs for the data volumes expected from NRAO telescopes in the next few years, (3) test the framework and time the parallel applications to measure the speedup in wall-clock run-time, and (4) based on the results, define the hardware parameters for the "final" cluster to be used by users for data reduction.

The parameters of the cluster bought for development and the wall-clock run time is described in Section 3. The software framework used for parallelization is described in Section 4.

The image deconvolution step is the most expensive in terms of the actual run time. Using 16-nodes, a speed up of $10-20\times$ compared to single-node time was measured using a data volume of ~ 86 GB. The image size used in these tests was $1K \times 1K$ pixels on the sky with 1024 pixels along the frequency axis for spectral line imaging. The CASA imaging software requires floating point buffers of size equivalent to $4-5\times$ the

```
Atlas
File Edit View Terminal Tabs Help
2009-05-18 00:08:38 INFO    clean::ClarkCleanModel::solve    Initial maximum residual: 0.0482283
2009-05-18 00:08:38 INFO    clean::ClarkCleanModel::solve    Iteration: 279, Maximum residual=0.00784143
2009-05-18 00:08:38 INFO    clean::ClarkCleanModel::solve    Iteration: 284, Maximum residual=0.00708038
2009-05-18 00:08:39 INFO    clean::MFCleanImageSkyModel::solve  Finished Clark clean inner cycle
2009-05-18 00:08:39 INFO    clean::MFCleanImageSkyModel::solve  Clean used 284 iterations to approach a threshold of 0.00728543
2009-05-18 00:08:47 INFO    clean::MFCleanImageSkyModel::solve  258.457 Jy is the sum of clean components of model 0
2009-05-18 00:08:48 INFO    clean::MFCleanImageSkyModel::solve  279.675 Jy is the sum of clean components of model 0
2009-05-18 00:08:48 INFO    clean::MFCleanImageSkyModel::solve  *** Starting major cycle 3
2009-05-18 00:08:48 INFO    clean::MFCleanImageSkyModel::solve  Making residual images for all fields
00] /home/casa-dev-02/sanjay/PTEST/casapy--8.log *Press F1/<CTRL>+<h> for help*   115694 - May 17 18:15:38 2009
2009-05-18 00:06:43 INFO    clean::ClarkCleanModel::solve    Initial maximum residual: 0.0547069
2009-05-18 00:06:43 INFO    clean::ClarkCleanModel::solve    Iteration: 268, Maximum residual=0.00887983
2009-05-18 00:06:43 INFO    clean::ClarkCleanModel::solve    Iteration: 272, Maximum residual=0.00811873
2009-05-18 00:06:43 INFO    clean::MFCleanImageSkyModel::solve  Finished Clark clean inner cycle
2009-05-18 00:06:43 INFO    clean::MFCleanImageSkyModel::solve  Clean used 272 iterations to approach a threshold of 0.00829858
2009-05-18 00:06:48 INFO    clean::MFCleanImageSkyModel::solve  255.569 Jy is the sum of clean components of model 0
2009-05-18 00:06:50 INFO    clean::MFCleanImageSkyModel::solve  279.033 Jy is the sum of clean components of model 0
2009-05-18 00:06:50 INFO    clean::MFCleanImageSkyModel::solve  *** Starting major cycle 3
2009-05-18 00:06:50 INFO    clean::MFCleanImageSkyModel::solve  Making residual images for all fields
01] /home/casa-dev-03/sanjay/PTEST/casapy--1.log *Press F1/<CTRL>+<h> for help*   118652 - May 17 18:15:38 2009
2009-05-18 00:09:07 INFO    clean::ClarkCleanModel::solve    Initial maximum residual: 0.0496863
2009-05-18 00:09:07 INFO    clean::ClarkCleanModel::solve    Iteration: 282, Maximum residual=0.00803546
2009-05-18 00:09:07 INFO    clean::ClarkCleanModel::solve    Iteration: 287, Maximum residual=0.00725512
2009-05-18 00:09:08 INFO    clean::MFCleanImageSkyModel::solve  Finished Clark clean inner cycle
2009-05-18 00:09:08 INFO    clean::MFCleanImageSkyModel::solve  Clean used 287 iterations to approach a threshold of 0.00728907
2009-05-18 00:09:17 INFO    clean::MFCleanImageSkyModel::solve  258.111 Jy is the sum of clean components of model 0
2009-05-18 00:09:18 INFO    clean::MFCleanImageSkyModel::solve  279.642 Jy is the sum of clean components of model 0
2009-05-18 00:09:18 INFO    clean::MFCleanImageSkyModel::solve  *** Starting major cycle 3
2009-05-18 00:09:18 INFO    clean::MFCleanImageSkyModel::solve  Making residual images for all fields
02] /home/casa-dev-04/sanjay/PTEST/casapy--2.log *Press F1/<CTRL>+<h> for help*   115935 - May 17 18:15:38 2009
2009-05-18 00:13:14 INFO    clean::ClarkCleanModel::solve    Initial maximum residual: 0.0458692
2009-05-18 00:13:14 INFO    clean::ClarkCleanModel::solve    Iteration: 287, Maximum residual=0.00695866
2009-05-18 00:13:14 INFO    clean::ClarkCleanModel::solve    Iteration: 290, Maximum residual=0.006296
2009-05-18 00:13:15 INFO    clean::MFCleanImageSkyModel::solve  Finished Clark clean inner cycle
2009-05-18 00:13:15 INFO    clean::MFCleanImageSkyModel::solve  Clean used 290 iterations to approach a threshold of 0.00650773
2009-05-18 00:13:24 INFO    clean::MFCleanImageSkyModel::solve  260.116 Jy is the sum of clean components of model 0
2009-05-18 00:13:26 INFO    clean::MFCleanImageSkyModel::solve  280.189 Jy is the sum of clean components of model 0
2009-05-18 00:13:26 INFO    clean::MFCleanImageSkyModel::solve  *** Starting major cycle 3
2009-05-18 00:13:26 INFO    clean::MFCleanImageSkyModel::solve  Making residual images for all fields
03] /home/casa-dev-05/sanjay/PTEST/casapy--3.log *Press F1/<CTRL>+<h> for help*   118842 - May 17 18:15:38 2009
2009-05-18 00:09:51 INFO    clean::ClarkCleanModel::solve    Initial maximum residual: 0.0435646
2009-05-18 00:09:51 INFO    clean::ClarkCleanModel::solve    Iteration: 290, Maximum residual=0.00649474
2009-05-18 00:09:51 INFO    clean::ClarkCleanModel::solve    Iteration: 293, Maximum residual=0.00598444
2009-05-18 00:09:51 INFO    clean::MFCleanImageSkyModel::solve  Finished Clark clean inner cycle
2009-05-18 00:09:51 INFO    clean::MFCleanImageSkyModel::solve  Clean used 293 iterations to approach a threshold of 0.0062316
2009-05-18 00:10:01 INFO    clean::MFCleanImageSkyModel::solve  261.588 Jy is the sum of clean components of model 0
2009-05-18 00:10:03 INFO    clean::MFCleanImageSkyModel::solve  280.463 Jy is the sum of clean components of model 0
2009-05-18 00:10:03 INFO    clean::MFCleanImageSkyModel::solve  *** Starting major cycle 3
2009-05-18 00:10:03 INFO    clean::MFCleanImageSkyModel::solve  Making residual images for all fields
04] /home/casa-dev-06/sanjay/PTEST/casapy--4.log *Press F1/<CTRL>+<h> for help*   112396 - May 17 18:15:38 2009
2009-05-18 00:07:41 INFO    clean::ClarkCleanModel::solve    Iteration: 281, Maximum residual=0.00793771
2009-05-18 00:07:42 INFO    clean::ClarkCleanModel::solve    Iteration: 286, Maximum residual=0.00716609
2009-05-18 00:07:42 INFO    clean::MFCleanImageSkyModel::solve  Finished Clark clean inner cycle
2009-05-18 00:07:42 INFO    clean::MFCleanImageSkyModel::solve  Clean used 286 iterations to approach a threshold of 0.00731094
2009-05-18 00:07:56 INFO    clean::MFCleanImageSkyModel::solve  257.702 Jy is the sum of clean components of model 0
2009-05-18 00:07:58 INFO    clean::MFCleanImageSkyModel::solve  279.692 Jy is the sum of clean components of model 0
2009-05-18 00:07:58 INFO    clean::MFCleanImageSkyModel::solve  *** Starting major cycle 3
2009-05-18 00:07:58 INFO    clean::MFCleanImageSkyModel::solve  Making residual images for all fields
05] /home/casa-dev-07/sanjay/PTEST/casapy--9.log *Press F1/<CTRL>+<h> for help*   116199 - May 17 18:15:38 2009
```

Figure 4: A screenshot showing use of the `multitail` program to monitor a number of log files in a single window.

image size. For continuum imaging, 8 GB of RAM was sufficient. Since 8 GB RAM was not sufficient for spectral line imaging, a number of disk scratch files were generated, resulting in significantly higher effective disk I/O. As a result, all else kept constant, the run time for spectral line imaging on a single node is $\sim 6\times$ longer than for continuum imaging.

While there was no I/O for accessing the image buffers during imaging, the run-time for continuum imaging was dominated by the I/O required to access the data from the disk (with 5 major cycles the total effective data I/O is equal to $\sim$ 500 GB). On the other hand, while the data I/O was same for spectral line imaging, since the image buffers are accessed in tight loops for iterative deconvolution, the run time here was limited even more severely by the disk I/O required for accessing these temporary buffers from the disk.

A crucial parameter for the cluster design therefore appears to be matching the available memory on the hardware to the memory requirements of the software. On the hardware, the relevant measure for memory is the available memory per CPU/core used for computing. The foremost software memory requirement comes from the required memory buffers which are a function of the image cube size. For spectral line imaging, this can be large. From a cluster design point of view, a cluster with sufficient *total* memory available across the cluster to hold these buffers should be the top priority. The memory requirement for image buffers can be estimated using the following equation

$$\texttt{Image buffer size} = 4 \times N_x \times N_y \times \frac{N_{Ch}}{N_{Nodes}} [1 + 4 \times N_{CoresPerNode}] \texttt{ bytes} \qquad (1)$$

For $N_x = Ny = N_{Ch} = 1024, N_{CoresPerNode} = 1$, even with the current cluster parameters (8 GB RAM per node), this limit is crossed using 4 nodes and one CPU per node. Using only 4 nodes, the run time is limited by the data I/O time. However using 16 nodes for imaging a $\sim$ 86 GB database, there is enough RAM to also hold all of the visibility data in the RAM distributed across the cluster. The run time advantages of this are significant - we indeed measure a super-linear speed up with number of nodes for spectral line imaging. Increasing the RAM per node to 32 GB can hold 512 GB worth of buffers (of course shared between the imaging buffers and the data buffers). It is therefore conceivable to design and afford a cluster with enough total RAM and computing power to process larger databases or larger images in extreme cases and possibly both for average data rates and image sizes.

6.1 Data splitting

For this work, the data base was split along the frequency axis at spectral window boundaries for distribution across the cluster. Assuming averaging the frequency channels available per node provides enough signal-to-noise ratio (SNR) for calibration, this splitting of the data is appropriate for both calibration and imaging. This, we think, will be the case for a significant number of observations, at least initially. A different splitting of the data will be required for imaging and calibration when a polynomial solver is required (e.g. for when a low order polynomial or G-Spline is used to model B-Jones across the band). The data base must be split across the frequency channel for imaging and across the time axis for polynomial-based calibration. That said, it is possible to parallelize the inner loops of the polynomial solver as well, which will mitigate the need for different splitting of the data (this requires keeping two copies of the data at the nodes with the associated increase in the required book-keeping complexity).

6.2 Serial computing and data communication

Spectral line imaging requires communication of the final image cubes from all nodes to a single node (see section 3.2). Continuum imaging requires communication of residual images from all nodes to a single node, a serial minor cycle followed by communication

of the updated model image to all nodes (see section 3.3). Note that the communication of spectral images in spectral line imaging is a one-time operation done, if at all required, at the end of the deconvolution iterations. Communication of images in the continuum imaging case is required as part of the major cycle iterations.

All these operations (data communication and serial computing) scale poorly with image size. $N_x \times N_y \times \frac{N_{channels}}{N_{nodes}}$ floating point pixels in the spectral line case and $2 \times N_x \times N_y \times N_{MajorCycles}$ floating point pixels in the continuum case are communicated across the cluster inter-connect network where N_x, N_y, and $N_{channels}$ are the number of pixels along the three axis of the image cubes and N_{nodes} is the number of compute nodes in the cluster. The number of bytes corresponding to this data communication is however much smaller than the data volume and the time for this data communication is relatively small fraction of the total run time. The serial computing operation is the computation of the final residual image. Assuming that the residual image fits in the RAM, this operations scales linearly with image size (or the search size in the image).

We are currently using a 1 Gigabit inter-connect. For large images, the data communication time might demand that a 10 Gigabit inter-connect be used. For the problem sizes we need to address, a much higher bandwidth interconnect might not be required, but this needs more careful assessment.

6.3 Error handling

The `MultiEngineClient` does provide some services to receive and handle exceptions from the `ipengine` clients connected to it. No error or exception handling was implemented in the framework described above. This is however crucial for robustness and for production line software and needs more work and possibly more exploration of the optimal approach for dealing with errors/exceptions in a distributed computing environment.

6.4 Cluster setup

As mentioned before, the current mechanism for initializing the software framework uses home-grown shell scripts (see section A). Use of a more sophisticated and possibly easier to use system using the `IPCluster` class is recommended. This also allows a more complete specification for the cluster resources by the end user or for configuring a post-processing pipeline (number of nodes, number of cores per node, names of the nodes, etc.).

7 Appendix

A The startclients.sh Bash Script

The following Bash script was used to start the ipcontroller on the host machine and ipengines using ssh on the list of machine named casa-dev-[01-15]. Since the ipengines ultimately run casapy which writes the log files in the local directory, the current directory is changed to a directory on the local disks at the nodes before starting the ipengines. Standard out and standard error output streams of the ipengines are redirected to the the local /tmp directory.

```
ipcontroller >| /tmp/ipcontroller.log &
echo "Starting local ipcontroller..."
sleep 10;

for c in $1
do
  ssh -f casa-dev-$c "cd /home/casa-dev-$c/sanjay/PTEST; \
     ipengine >| /tmp/engine.stdout 2>|/tmp/engine.stderr"
  echo "Start on $c"
done
```

B The cluster.py Python Script

The various functions in the cluster.py script are briefly described below.

- pinit(message="Hello CASA Cluster"):

  Function to setup the communication between the ipcontroller and the interactive IPython CASA interface. This function returns the MultiEnigneClient object which is the primary scripting interface for communication with the cluster nodes.

- startcasapy(rmec,root=PARALLEL_CASAPY_ROOT_DIR):

  Function to load the modified casapy.py script at the cluster nodes. This function *must* be called after successful execution of the pinit() function. The rmec variable should be set to the object returned by the pinit() function.

- mkname(node,rootdir,workdir,basename,ext,regex=false):

  Function to construct the directory and file names using node ID, and other user defined strings. The file/directory naming convention used throughout the parallelization framework is implemented in this function.

- `rcmd(var,op,value)`:

  A helper function to construct `Python` commands strings to apply the operator `op` on variable `var` using `value` as the RHS of the operator. This is typically used to construct strings like `var=value`.

- `setUpFlagger(...)`, `setUpGaincal(...)`, `setUpBandpass(...)`, `setUpApplycal(...)`, `setUpClean(...)`

  Functions to set up the variables of the `flagdata`, `gaincal`, `bandpass`, `applycal`, and the `clean` tasks. Apart from task specific arguments, all these functions take the following parameters

  - `rmec`

    The `MultiEngineClient` object returned from a successful call to `pinit()`.

  - `nodes`

    A list of node IDs to be used for parallel computing and I/O.

  - `configfile=PARALLEL_CONFIGFILES_DIR+"<taskname>.last"`

    A task configuration file to set the task variables with a common value across the nodes. `PARALLEL_CONFIGFILES_DIR` is a global framework variable pointing to the directory containing the configuration files.

  - `rootdir=PARALLEL_BASE_ROOT_DIR`

    Path to the basename of the work-directory. `PARALLEL_BASE_ROOT_DIR` is a global framework variable.

  - `workdir=PARALLEL_WORK_DIR`

    Path to the name of the work-directory. `PARALLE_WORK_DIR` is a global framework variable.

- `pspectralline(...)`

  A function for parallel spectral line image deconvolution. This uses the `setUpClean()` and calls `MEC.execute('go('clean')')` command.

- `pcontinuum(...)`

  A function for parallel continuum image deconvolution. This uses the `setUpClean()` function, and implements the deconvolution minor cycle as well.