Imaging Unmitigated ALMA Cubes

NAASC Memo #121

Authors: Amanda A. Kepley, Felipe Madsen, James Robnett, and K. Scott Rowe

Date: 15th August 2023

ABSTRACT

ALMA is capable of producing cubes that are extremely challenging to image using the ALMA Imaging Pipeline and CASA, its underlying data processing software. These cubes are currently mitigated, or reduced in size, in operations to successfully process and deliver this data to principle investigators (PIs). This memo investigates the performance of the ALMA Imaging Pipeline and CASA when fully imaging these large cubes. We find that the performance of the Imaging Pipeline does not always scale with increasing parallelization breadth. The performance of the CASA tclean task, however, largely scales with increasing parallelization breadth up to a breadth of 17. Increasing the parallelization breadth by running jobs across multiple nodes did not yield any improvements due to a high overhead associated with file locking. A detailed investigation of a representative stage of the Imaging Pipeline -- findcont -- revealed that the run time of this stage is dominated by the CASA tasks immoments, imstat, and ia.getprofile, not by the time required for tclean to create the dirty cube. We find that the performance of tclean scales with number of channels, number of pointings, and number of measurement sets (i.e. execution blocks), largely as expected for these cubes, although we did encounter failures in the minor cycle of tclean for our largest cubes. In most cases, the minor cycle run time dominated over the major cycle run time with 60-80% of the minor cycle time spent automatically masking the emission. Finally, we have found that in some cases the use of Non-Volatile Memory Express (NVMe) devices reduces the imaging time by 30-53%. These results will inform improvements to CASA and the ALMA Imaging Pipeline needed to handle the order of magnitude increase in number of channels per spectral window that will be produced by the ALMA Wideband Sensitivity Upgrade.

NAASC Memo 121
Imaging Unmitigated ALMA Cubes

Amanda A. Kepley,[1] Felipe Madsen,[2] James Robnett,[2] and K. Scott Rowe[2]

[1] *National Radio Astronomy Observatory, 520 Edgemont Road, Charlottesville, VA 22903*
[2] *National Radio Astronomy Observatory, P.O. Box O, 1003 Lopezville Road, Socorro, NM 87801-0387*

(Received May 12, 2023; Published August 15, 2023)

ABSTRACT

ALMA is capable of producing cubes that are extremely challenging to image using the ALMA Imaging Pipeline and CASA, its underlying data processing software. These cubes are currently mitigated, or reduced in size, in operations to successfully process and deliver this data to principle investigators (PIs). This memo investigates the performance of the ALMA Imaging Pipeline and CASA when fully imaging these large cubes. We find that the performance of the Imaging Pipeline does not always scale with increasing parallelization breadth. The performance of the CASA `tclean` task, however, largely scales with increasing parallelization breadth up to a breadth of 17. Increasing the parallelization breadth by running jobs across multiple nodes did not yield any improvements due to a high overhead associated with file locking. A detailed investigation of a representative stage of the Imaging Pipeline – `findcont` – revealed that the run time of this stage is dominated by the CASA tasks `immoments`, `imstat`, and `ia.getprofile`, not by the time required for `tclean` to create the dirty cube. We find that the performance of `tclean` scales with number of channels, number of pointings, and number of measurement sets (i.e. execution blocks), largely as expected for these cubes, although we did encounter failures in the minor cycle of `tclean` for our largest cubes. In most cases, the minor cycle run time dominated over the major cycle run time with 60-80% of the minor cycle time spent automatically masking the emission. Finally, we have found that in some cases the use of Non-Volatile Memory Express (NVMe) devices reduces the imaging time by 30-53%. These results will inform improvements to CASA and the ALMA Imaging Pipeline needed to handle the 1.5 order of magnitude increase in number of channels per spectral window that will be produced by the ALMA Wideband Sensitivity Upgrade.

1. INTRODUCTION

The largest cubes produced by ALMA are extremely challenging to image because of their size, which can be up to 13k by 13k pixels with 7680 channels (∼5 TB). Attempting to image cubes this large often leads to unpredictable failures and would lead to ALMA Imaging Pipeline run times on the order of months to years. To successfully process and deliver the data to principle investigators (PIs), the ALMA Imaging Pipeline (Hunter et al. 2023) mitigates, or reduces, the size of cubes produced down to a maximum of 60 GB by averaging two channels together, reducing the imaged area, and/or reducing the number of pixels per restoring beam. It also reduces the number of sources and spectral windows imaged to reduce the total product size below 500 GB. These empirically derived heuristics allow Common Astronomy Software Applications (CASA; THE CASA TEAM et al. 2022), the processing software underlying the ALMA Pipeline (hereafter referred to as the Pipeline), to be able to successfully image these cubes.

Mitigation affects a significant amount of the ALMA data processed today. In Cycle 7, 19% of 12-m member observation unit sets (MOUSes) were mitigated; the largest impacts were for baselines greater than 1 km (Kepley et al. 2023). Cube imaging will become even more challenging with the upcoming ALMA Wideband Sensitivity Upgrade (WSU; Carpenter et al. 2022)[1]. The goal of the WSU project is to increase the correlated bandwidth of ALMA by a factor of two to four, which will lead to an increase in the maximum number of channels per spectral window by approximately 1.5 orders of magnitude (Kepley et al. in prep).

The goal of this memo is to investigate the performance of the current ALMA Imaging Pipeline, and its underlying data processing software CASA, when imaging the largest cubes ALMA is currently able to produce to guide future CASA and ALMA Imaging Pipeline development for WSU. We begin by giving an overview of

[1] https://science.nrao.edu/facilities/alma/facilities/alma/science_sustainability/wideband-sensitivity-upgrade

the ALMA Imaging Pipeline and the key CASA tasks used in the Pipeline in Section 2. We describe our testing methodology in Section 3, including our test samples (§3.1) and testing setup (§3.2). Our results are detailed in Section 4. We start by exploring the performance of the Imaging Pipeline and the underlying `tclean` task used for image reconstruction as a function of parallelization breadth (§4.1). Then we look in detail at the performance of a representative stage of the Imaging Pipeline: `findcont` (§4.2). In Section 4.3, we investigate how the performance of `tclean` scales with three key data set parameters: number of channels, number of mosaic pointings, and number of execution blocks. Next we compare the relative performance of the major and minor cycles in the scaling tests in Section 4.4. Finally, we compare the performance using Non-Volatile Memory Express (NVMe) devices for data storage during processing compared to Lustre (§ 4.5). We summarize our findings and note some possible directions for future work in Section 5.

2. OVERVIEW OF THE ALMA IMAGING PIPELINE

The ALMA Pipeline is a layer of Python-based heuristics on top of CASA, a general package for processing interferometer data. It is split into two parts: a Calibration Pipeline and an Imaging Pipeline. Here we focus on the functionality of the Imaging Pipeline since this is the more time and processing intensive portion of the Pipeline.

As a general purpose package, CASA has a wide range of functionality, only a subset of which is used by the ALMA Imaging Pipeline. The primary CASA task used by the ALMA Imaging Pipeline is `tclean`, which is used to image and deconvolve the data. Within `tclean`, the `auto-multithresh` algorithm is used to automatically mask emission in all images (Kepley et al. 2020). The Imaging Pipeline also uses several CASA tasks and tools related to image analysis including `imstat` (calculate image statistics), `immoments` (calculate image moments), and `ia.getprofile` (calculate spectral profile within a mask).

Below we provide an overview of the key steps in the Imaging Pipeline. We refer to each step by its stage name in the current Pipeline for convenience. For detailed information, please consult the current ALMA Pipeline User's Guide (ALMA Pipeline Team 2022) and Hunter et al. (2023). There are several planned future improvements to the Pipeline that may alter the heuristics outlined below including the combination of data from different array configurations and the inclusion of automated self-calibration.

The Cycle 9 Imaging Pipeline (PL2022) stages are the following:

1. `imageprecheck`: chooses a robust value for imaging so that the resulting synthesized beam most closely matches the PI-requested resolution. To do this, it uses CASA synthesis imager tools to calculate the synthesized beam for robust values between 0 and 2. For mosaics, only the central pointings are used to estimate the synthesized beam to reduce processing time.

2. `checkproductsize`: This stage determines whether the imaging products will be mitigated and how to mitigate them. The detailed heuristic is given in Appendix A and the mitigation parameters used are given below.

3. `makeimlist (mfs)`: determines the imaging parameters to use for subsequent continuum images (imsize, etc).

4. `findcont`: identifies line-free regions in the data to use for continuum subtraction. To do this, a dirty image is created using `tclean` for all sources/spectral windows (spws). Then a spectrum is generated from a masked region constructed from the integrated intensity and peak intensity images. The initial mask is amended to yield improved regions if necessary. This stage makes extensive use of the CASA tasks `immoments`, `imstat`, and `ia.getprofile`.

5. `uvcontfit` and `uvcontsub`: performs the continuum subtraction of the data using regions identified using `findcont`. This is done in *uv*-space with no phase shifting to account for continuum emission at the edge of the field of view. Prior to PL2023, this stage used a combination of the CASA tasks `gaincal` and `applycal` to fit and subtract the spectrum, but will transition to the `uvcontsub` task for PL2023. The numerical results are identical to machine precision between the two methods.

6. `makeimages (mfs)`: generates cleaned continuum images for each source/spectral window (spw) using the imaging parameters derived in the earlier makeimlist stage and `tclean`.

7. `makeimages (cont)`: generates aggregate continuum images for each source including all spectral windows using `tclean`.

8. `makeimlist (cube)`: determines the cube imaging parameters to use for subsequent cube images.

Unless the cube is mitigated, all channels are imaged.

9. `makeimages(cube)`: generates continuum-subtracted line cubes for each source/spw combination using `tclean`. This stage includes heuristics using integrated intensity (moment 0) and peak intensity (moment 8) images to assess the quality of the continuum subtraction.

10. `makeimlist (cube_repBW)`: determines image parameters for the representative source and spectral window at the PI requested spectral resolution if the requested spectral resolution is greater than 4 times larger than the correlator channel width.

11. `makeimages (cube_repBW)`: generates a cube for the representative spectral window and source at the PI requested spectral resolution using `tclean`.

The current ALMA Pipeline mitigation limits are given below:

- `maxcubesize` – 40 GB. Cubes greater than this size will trigger the cube mitigation (averaging channels, reducing the imaged area, and/or reducing the number of pixels per beam).

- `maxcubelimit` – 60 GB. If the final mitigated cube is larger than this size, the Imaging Pipeline will stop with the error message "mitigation failed". This parameter also controls the total number of large cubes produced.

- `maxproductsize` – 500 GB. This is the total product size at which the number of science targets imaged is reduced. The cubes will be mitigated if necessary.

Depending on the ALMA Regional Center (ARC) that data is being processed at, the default mitigation limits may be lifted to avoid manual processing.

In operations, the Pipeline is generally run at the Joint ALMA Observatory (JAO). The Pipeline is launched via another piece of software called ADAPT, which has multiple subcomponents. CASA is configured to run in parallel using a wrapper script for the Message Passing Interface (MPI) executor that comes packaged with CASA. See Section 2.4 of THE CASA TEAM et al. (2022) for more details. Typical runs are configured to use 8 cores and 256 GB of memory. Data is processed per MOUS, which is a collection of executions of the same scheduling block. A MOUS has the same sources and spectral setup for every execution and the number of executions can range from 1 to many tens (more for Total Power and 7-m data).

3. TESTING METHODOLOGY

3.1. *Samples*

3.1.1. *Initial Sample Selection*

Our initial sample was chosen from the ALMA Pipeline benchmark, which is a set of MOUSes that were chosen to exercise a broad range of ALMA use cases to validate the ALMA Pipeline for use in operations. Figure 1 shows how the unmitigated properties of the selected data sets compare with those of 12-m data from Cycle 7 as a whole. The MOUSes that were selected for testing included some of the longer running benchmark projects and included examples of MOUSes with mosaics as well as those with multiple targets and execution blocks. Some of the selected MOUSes had their sizes mitigated in operations and we used the mitigated versions of these data sets for our testing. The detailed properties of the selected MOUSes in the initial sample are given in Table 1 and the mitigations that were used are given in Table 2.

3.1.2. *Extended Sample Selection*

Since data sets in the benchmark are used for Pipeline validation and testing, they are weighted towards MOUSes that have relatively short run times with the longest running cases taking two to three days. In operations, however, some Pipeline jobs can take up to a month to run. The longest running Pipeline jobs have had their run times limited by reducing the size of the imaging products through a process called mitigation. Without mitigation, the estimated Pipeline imaging run times could be as long as 1.7 years (Kepley et al. 2023). To test these cases, we decided to expand our sample beyond the ALMA Pipeline benchmark to include more MOUSes with extreme image sizes and larger numbers of channels.

To identify a sample of suitably challenging cubes, we identified all 2019 12-m MOUSes that had some QA2 pass data taken for them by the end of Cycle 7, excluding solar, total power projects, and solar system projects. Solar and total power MOUSes were excluded because they follow different data reduction pathways, while solar system MOUSes were excluded because the archive gives their fields of view (FOV) as the FOV over which the object was observed on the sky rather than the imaged FOV. We then downloaded high level information about these MOUSes from the ALMA Archive including FOV, angular resolution, array, spectral window information (frequency, bandwidth, spectral reso-

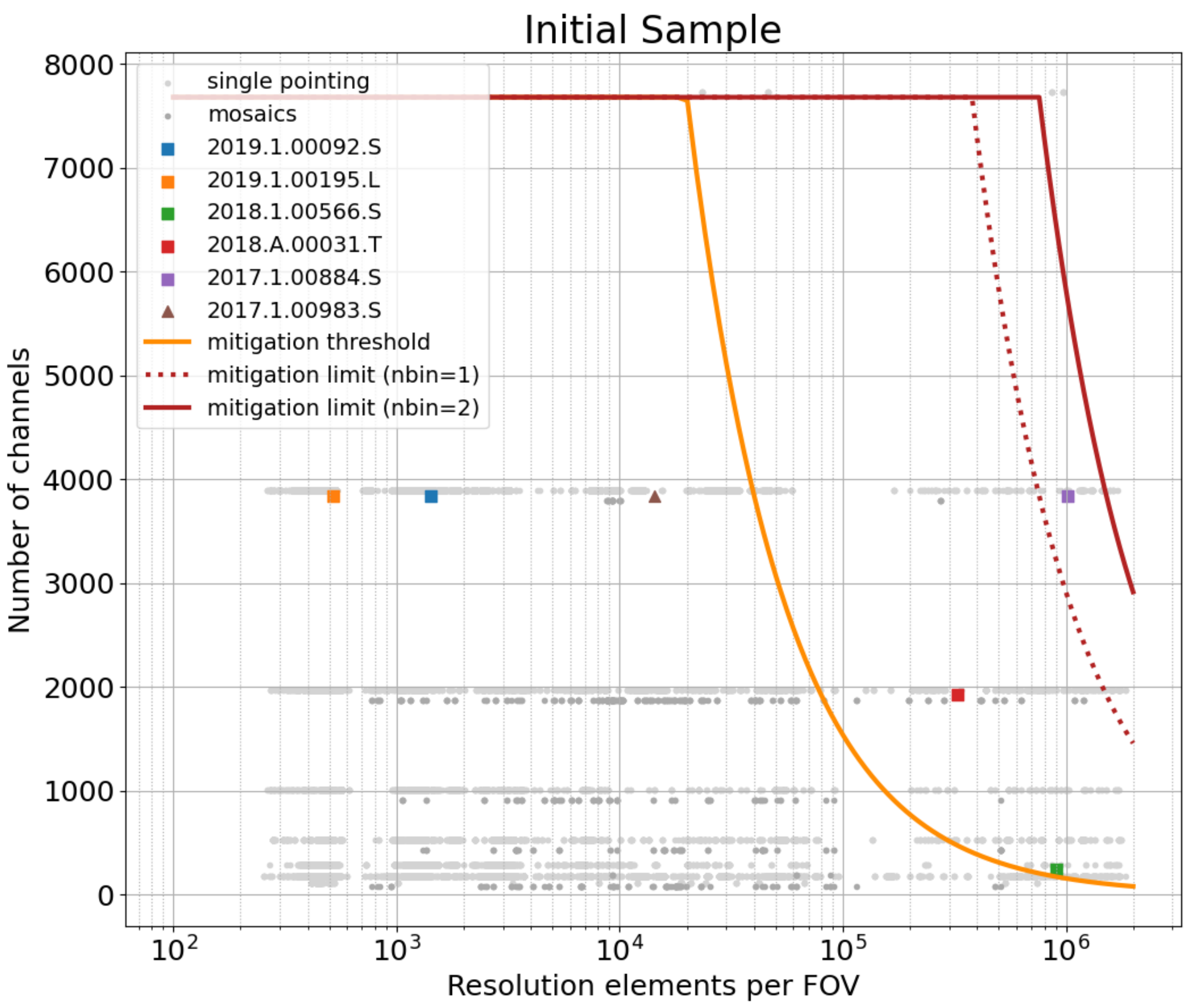


Figure 1. Number of channels as a function of resolution elements per field of view for all ALMA Cycle 7 12-m data. The orange line indicates the mitigation threshold, above which the Pipeline begins to mitigate cube sizes. The red dotted line indicates the mitigation limit, or maximum cube size permitted by the Pipeline, when channels cannot be binned because they have already been binned in the correlator. The red solid line indicates the mitigation limit when the channels can be binned by a factor of 2 because they have not been binned in the correlator. The data sets selected as part of our initial sample are indicated by colored symbols. Single fields are indicated by squares and mosaics by triangles.

lution, and polarization), and number of targets. This information is determined from the Scheduling Block properties and other public meta-data in the Archive rather than that derived by the Pipeline and thus represents what the observations would have achieved had they not been mitigated during processing. Since the spectral window information in the archive had been transformed to match International Virtual Observatory Alliance (IVOA) standards, we converted the spectral window parameters from wavelengths to frequencies and reverse engineered the resulting instrumental properties like number of channels based on information in the ALMA Technical Handbook (Cortes et al. 2022).

From the resulting database, we selected two sets of MOUSes: one set close to the mitigation threshold (project codes: 2019.1.00876.S, 2019.1.00915.S, and 2019.1.00263.S_X3477) and the rest close to or at the mitigation limits (project codes: 2019.1.00877.S, 2019.1.00263.S_X3465, 2019.1.01425.S, 2019.1.01463.S,

Table 1. Initial Sample

Proposal Code	MOUS	N_{ebs}	N_{ptngs}	N_{srcs}	Points per FOV	N_{pol}	Mitigated?	Mitigated imsize	Mitigated max N_{chan} [a]
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
2019.1.00092.S	`uid://A001/X1465/X3a33`	1	1	1	1,428	2	N	[450, 450]	3840
2019.1.00195.L	`uid://A001/X146c/Xd8`	1	1	28	517	2	N	[192, 192]	3840
2018.1.00566.S	`uid://A001/X133d/X2067`	6	1	1	901,816	2	Y	[6250, 6250]	240 (240)
2018.A.00031.T	`uid://A001/X13b3/Xdc`	1	1	1	327,800	2	Y	[1440, 1440]	1920 (1920)
2017.1.00884.S	`uid://A001/X1296/X7b1`	1	1	1	1,018,995	2	Y	[1600,1600]	1920 (3840)
2017.1.00983.S	`uid://A001/X12a3/X3be`	3	27	2	14,268	2	N	[750, 800]	3840

[a] Unmitigated maximum N_{chan} given in parenthesis if applicable.

Note—Column (1): Proposal Code. Column (2): MOUS. Column (3): Number of execution blocks (EBs). Column (4): Number of pointings (> 1 pointing indicates a mosaic). Column (5): Number of sources. Column (6): Number of resolution elements per field of view. This is a frequency-independent quantity proportional to the image size in pixels. Column (7): Number of polarizations. Column (8): whether or not the data was mitigated by the Pipeline in operations. Column (9): The mitigated image size in pixels. Column (10): The mitigated number of channels in a spectral window (maximum over all spectral windows).

Table 2. Pipeline Mitigations for Initial Sample

Proposal	MOUS	nbins	`hm_imsize`	`hm_cell`	field	spw
(1)	(2)	(3)	(4)	(5)	(6)	(7)
2018.1.00566.S	`uid://A001/X133d/X2067`	· · ·	0.25pb	· · ·	· · ·	21, 23, 25
2018.A.00031.T	`uid://A001/X13b3/Xdc`	31:2,25:2,33:1,27:2,35:1,29:2	0.7pb	3ppb	· · ·	· · ·
2017.1.00884.S	`uid://A001/X1296/X7b1`	25:2,27:2,23:1,19:1	0.7pb	3.25pb	· · ·	· · ·

Note—Column (1): Proposal Code. Column (2): MOUS. Column (3): number of channels binned per spectral window, represented as a spw:nbin in a comma separated list. A null value means that no channels were binned by the Pipeline in any spectral windows (although they may have been binned in the correlator). Column (4): fraction of the primary beam imaged. Column (5): The number of pixels per beam. A null value means that the default value (five pixels per beam) was used. Column (6): The fields included in imaging. A null value means all fields were imaged. Column (7): The spectral windows imaged. A null value means that all spectral windows were imaged.

2019.1.01074.S, and 2019.1.00592.S, 2019.1.00796.S).[2] Figure 2 provides an overview of the unmitigated properties of the selected 12-m MOUSes compared to 12-m data from Cycle 7 as a whole. We included mosaics and single fields since these use different gridding options in `tclean`. The extended sample also includes a single polarization case (2019.1.00592.S) which has the largest possible number of channels that the ALMA 64-Input Correlator (also known as the Baseline Correlator or BLC) can produce today: 7680. The properties for the extended test data sets are given in Table 3 and the mitigations used in operations to process the data are given in Table 4.

[2] We note that two of our data sets come from the same project (2019.1.00263.S), so we distinguish them by appending the last section of their MOUS identifier to the proposal code.

3.2. *Testing Setup*

Initial testing was done using a combination of nodes from the ALMA Pipeline Working Group (PLWG) at the North American ALMA Science Center (NAASC) as well as general processing (nmpost) nodes at Pete V. Domenici Science Operations Center (DSOC) and VLASS nodes at both DSOC and New Mexico Tech (NMT). Our tests on the extended sample used the PLWG nodes exclusively. All nodes had identical specifications – dual 12-core, 3 GHz processors (Intel Xeon Gold 6136) and either 512 GB or 768 GB of memory – and similar levels of achieved performance. Tests were run exclusively, meaning no other processing was running on the node during each test. The initial performance tests described in Section 4.1 used 250 GB for

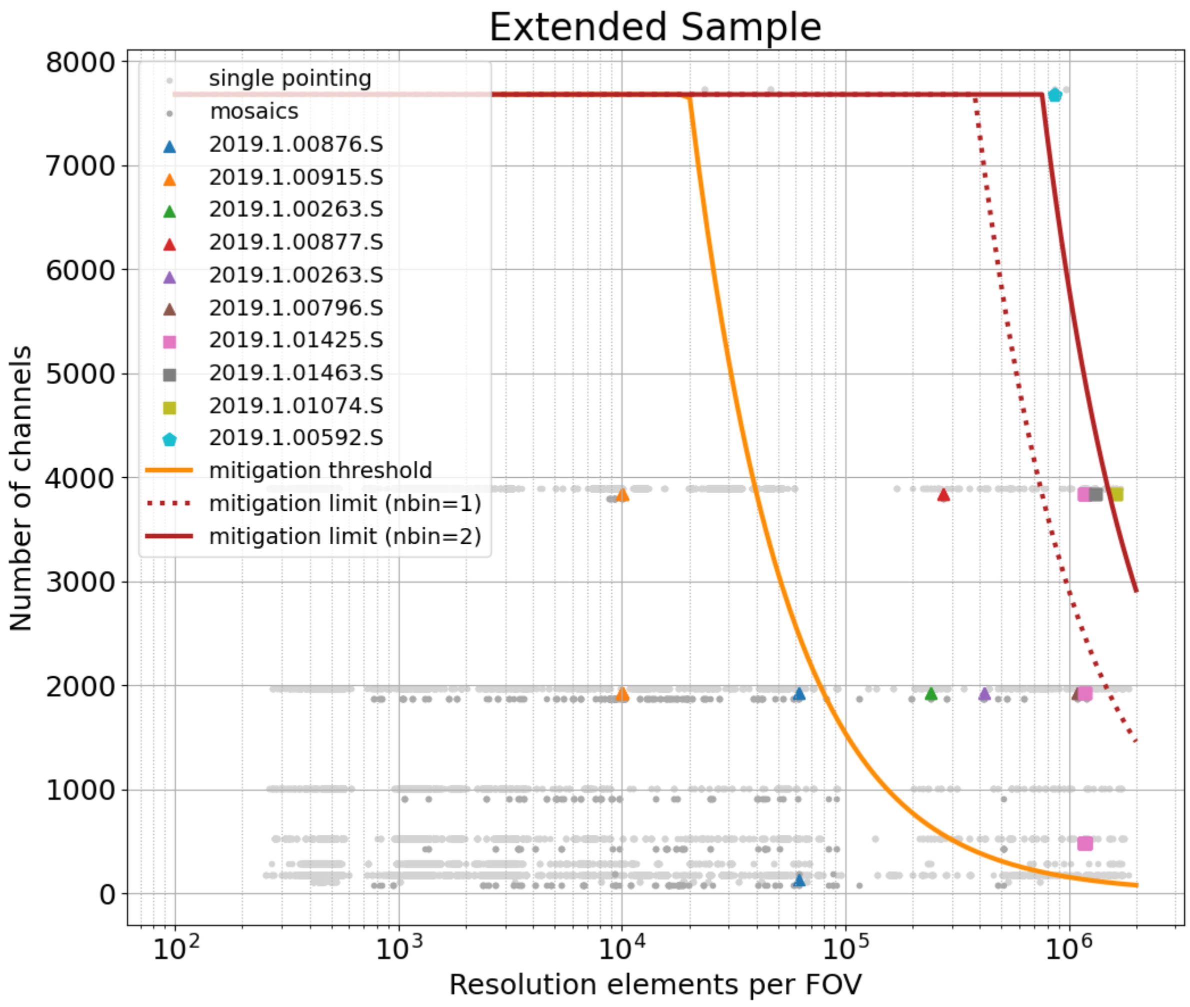


Figure 2. Number of channels as a function of resolution elements per field of view for all ALMA Cycle 7 12-m data. The orange line indicates mitigation threshold, above which the Pipeline begins to mitigate cubes sizes. The red dotted line indicates the mitigation limit, or maximum cube size permitted by the Pipeline, when channels cannot be binned (nbin=1) and the red solid line indicates the mitigation limit permitted by the Pipeline when channels can be binned (nbin=2). The data sets in our extended sample indicated by colored symbols. Single fields are indicated by squares and mosaics by triangles. The pentagon indicates the single polarization case.

parallelization breadths of 8 and 9, either 250 GB or 500 GB for a parallelization breadth of 17, and 500 GB for a parallelization breadth of 25. Our extended sample testing used either 500 GB or 745 GB of memory.

Unless specified otherwise, we used Lustre, a distributed parallel file system, as the file system for our tests. Benchmarks of the NAASC Lustre filesystem, which is associated with the PLWG nodes, using a single, 500 GB file show sequential reads averaging 693 MB/s and sequential writes averaging 860 MB/s. The AOC Lustre filesystem, which is associated with the nmpost and VLASS nodes, shows slightly better performance with sequential reads averaging 743 MB/s and sequential writes averaging 892 MB/s. The large 500 GB file was used for these tests to prevent the operating system from caching the file. Thus these benchmarks represent the average minimum speed of each Lustre filesystem to read and write a single file. With smaller files, caching starts to become significant. For example using a single, 10 GB file on NAASC Lustre shows much faster sequential reads and writes averaging 2,428 MB/s and 1,167 MB/s, respectively. Finally, because each node has a 40 Gb/s connection to Lustre, the theoretical effec-

Table 3. Extended Sample

Proposal Code	MOUS	N_{ebs}	N_{ptngs}	N_{srcs}	Points per FOV	N_{pol}	Mitigated?	Unmitigated imsize	Unmitigated max N_{chan}
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
2019.1.00263.S	`uid://A001/X1465/X3477`	3	19	1	259,946	2	Y[a]	[1440, 3072]	1920
2019.1.00263.S	`uid://A001/X1465/X3465`	2	53	1	384,414	2	Y	[3200, 3000]	1920
2019.1.00592.S	`uid://A001/X1465/X2a84`	1	1	1	635,935	1	Y	[8640, 8640]	7680
2019.1.00796.S	`uid://A001/X1471/X317`	4	3	1	757,052	2	Y	[7776,7776]	1920
2019.1.00876.S	`uid://A001/X1465/X20d0`	2	7	1	55,089	2	N	[1728, 1728]	1920
2019.1.00877.S	`uid://A001/X1465/X20c1`	3	7	1	271,030	2	Y	[3600, 3840]	3840
2019.1.00915.S	`uid://A001/X1465/X2009`	1	27	5	7,965	2	N	[600, 640]	3840
2019.1.01074.S	`uid://A001/X1465/X1ac2`	2	1	1	1,772,044	2	Y	[12960, 12960]	3840
2019.1.01425.S	`uid://A001/X1465/Xd63`	3	1	7	868,489	2	Y	[12800, 12800]	3840
2019.1.01463.S	`uid://A001/X1465/Xc05`	6	1	1	1,100,957	2	Y	[11250, 11250]	3840

[a] This MOUS was mitigated in PL2021, but not in PL2020.

Note—Column (1): Proposal Code. Column (2): MOUS. Column (3): number of execution blocks (EBs). Column (4): Number of pointings (> 1 pointing indicates a mosaic). Column (5): Number of sources. Column (6): Number of resolution elements per field of view. This is a frequency-independent quantity proportional to the image size in pixels. Column (7): Number of polarizations. Column (8): whether or not the data was mitigated by the Pipeline in operations. Column (9): The unmitigated image size in pixels. Column (10): The unmitigatednumber of channels in a spectral window (maximum over all spectral windows).

Table 4. Pipeline Mitigations for Extended Sample

Project	MOUS	nbins	`hm_imsize`	`hm_cell`	field	spw
(1)	(2)	(3)	(4)	(5)	(6)	(7)
2019.1.00263.S	`uid://A0001/X1465/X3477`	···	···	···	···	25[a]
2019.1.00263.S	`uid://A001/X1465/X3465`	···	0.44pb	3ppb	···	···
2019.1.00592.S	`uid://A001/X1465/X2a84`	···	0.7pb	3ppb	···	25
2019.1.00796.S	`uid://A001/X1465/X317`	···	0.68pb	3ppb	···	25
2019.1.00877.S	`uid://A001/X1465/X20c1`	23:2,27:2,29:2,25:2	0.57pb	3ppb	···	23
2019.1.01074.S	`uid://A001/X1465/X1ac2`	23:2,21:2,27:2,25:2	0.7pb	3ppb	···	23
2019.1.01425.S	`uid://A001/X1465/Xd63`	37:2,27:2,33:2,35:1,29:2,25:2,31:2	0.7pb	3ppb	NGC1333IRAS2A	25,27,29,31,37
2019.1.01463.S	`uid://A001/X1465/Xc05`	29:2,27:2,31:2,25:2	0.7pb	3ppb	···	···

[a] This MOUS was mitigated in PL2021, but not in PL2020.

Note—Column (1): Proposal Code. Column (2): MOUS. Column (3): number of channels binned per spectral window, represented as a spw:nbin in a comma separated list. A null value means that no channels were binned by the Pipeline in any spectral windows (although they may have been binned in the correlator). Column (4): fraction of the primary beam imaged. Column (5): The number of pixels per beam. A null value means that the default value (five pixels per beam) was used. Column (6): The fields included in imaging. A null value means all fields were imaged. Column (7): The spectral windows imaged. A null value means that all spectral windows were imaged.

tive throughput, or digital bandwidth limit, of a single node is about 4,800 MB/s.

We also explored using NVMe devices as our file system for some of our tests. An NVMe is a solid state drive connected to a PCIe bus, providing a faster drive closer to the CPU than a traditional hard disk drive. Our initial NVMe testing used a combination of NVMes on the VLASS nodes at NMT and the DSOC, while our NVMe tests on the extended sample used a PLWG node (cvpost127). The VLASS nodes at NMT (nmpost{091..120}) have Western Digital Ultrastar DC SN640, 3.85 TB NVMe drives. Published specifications show sequential reads at 3,300 MB/s and sequential writes at 2,040 MB/s. Multi-process, multi-file benchmarks done on site confirm these values. Single-process benchmarks using a single 500 GB file found sequential reads averaging 859 MB/s and sequential writes averaging 1,369 MB/s. The VLASS nodes at DSOC (nmpost{061..090}) and cvpost127 have Samsung PM1735, 6.4 TB NVMe drives. Published specifications show sequential reads at 8,000 MB/s and sequential writes at 3,800 MB/s. Multi-process, multi-file benchmarks done on site confirmed sequential write values but found slightly lower sequential read values of 7,010 MB/s. Single-process benchmarks using a single 500 GB file found sequential reads averaging 1,694 MB/s and sequential writes averaging 2,068 MB/s.

Our initial tests used an early version of PL2021 *casa-6.2.1-4-pipeline-2021.2.0.69*, but we switched to the final, released PL2021 version *casa-6.2.1-7-pipeline-2021.2.0.128* for our tests with the extended sample. The differences between the early version and the final release were minor. A custom version of the Pipeline was used to perform so-called dry runs of `tclean` calls for unmitigated continuum subtracted data, issuing the `tclean` commands to generate unmitigated cubes. Finally, in some cases, gridding failed in our default version of CASA due to a known issue with file locking, which is documented in the CASA development ticket CAS-13609 on Jira. In that case, we attempted to use *casa-6.4.1-12*, where this problem was fixed, to image this data. However, these attempts also failed due to additional memory issues with `auto-multithresh` in the minor cycle.

Profiling information for Pipeline executions was obtained by using the Pipeline profiling framework[3] that has been used in previous ALMA Pipeline profiling work done by a subset of this group (F. Madsen, J. Robnett, and K. Rowe). The profiling framework consists of a suite of python tools and shell scripts to extract detailed performance information from Pipeline executions at runtime.

Timing information for CASA sessions that did not use the ALMA Pipeline was obtained by parsing CASA logs with *readImagingLog*[4]: a collection of python class and function definitions to parse CASA logs containing imaging tasks (tclean, sdintimaging). A script that is part of the profiling framework was also used to retrieve performance plots (memory_report, cpu_report and ib_bytes_in) from ganglia every 4 hours.

4. RESULTS

4.1. *Parallelization Breadth Tests*

4.1.1. *Single Node Parallelization Breadth Tests*

The initial phase of the testing focused on characterizing the performance of the ALMA Imaging Pipeline as a whole, and `tclean` in particular, with increasing parallelization breadth. To do this, we used the initial test sample described in Section 3.1.1 because it was possible to run these data sets through the ALMA Imaging Pipeline in a reasonable amount of time (much less than a month). We ran each MOUS in the initial sample changing parallelization breadth from 1 (serial) to 25-way and recorded the resulting run time. We also determined the run time of just the `tclean` calls within the Pipeline, since the majority of the computational work happens in these calls.

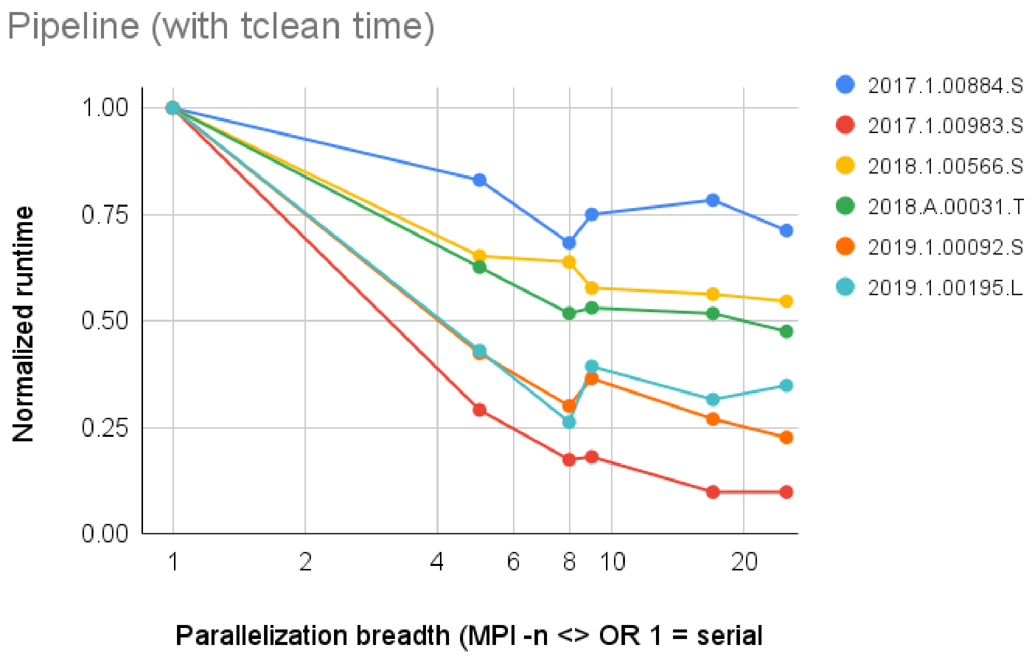
The results of these tests are shown in Figure 3. The run time of the Pipeline as a function of parallelization breadth is shown in the top panel and the run time of the `tclean` calls within the Pipeline as a function of parallelization breadth is shown in the bottom panel. For the Pipeline as a whole, the behavior of run time with increasing parallelization breadth varies widely between different MOUSes. Some MOUSes (e.g., 2017.1.00983.S) show the expected decrease in run time with increasing parallelization breadth, while others (e.g., 2017.1.00884.S) do not show significant improvement beyond 8-way parallelization. In contrast, for `tclean`, most MOUSes show improved performance with increasing parallelization breadths up to a breadth of 17 with the exception of 2019.1.00195.L. The latter data set has the smallest field of view by an order of magnitude compared to the rest of our data sets and thus is not able to benefit from increased parallelization breadth as much as data sets with larger field of view. From this, we conclude that while the behavior of `tclean` scales as expected with increasing parallelization breadth, the behavior of the ALMA Imaging Pipeline does not because the overall performance improvement for the Pipeline is limited by the serial operations such as `imstat`, `immoments`, etc.

4.1.2. *Multi-node Parallelization Breadth Tests*

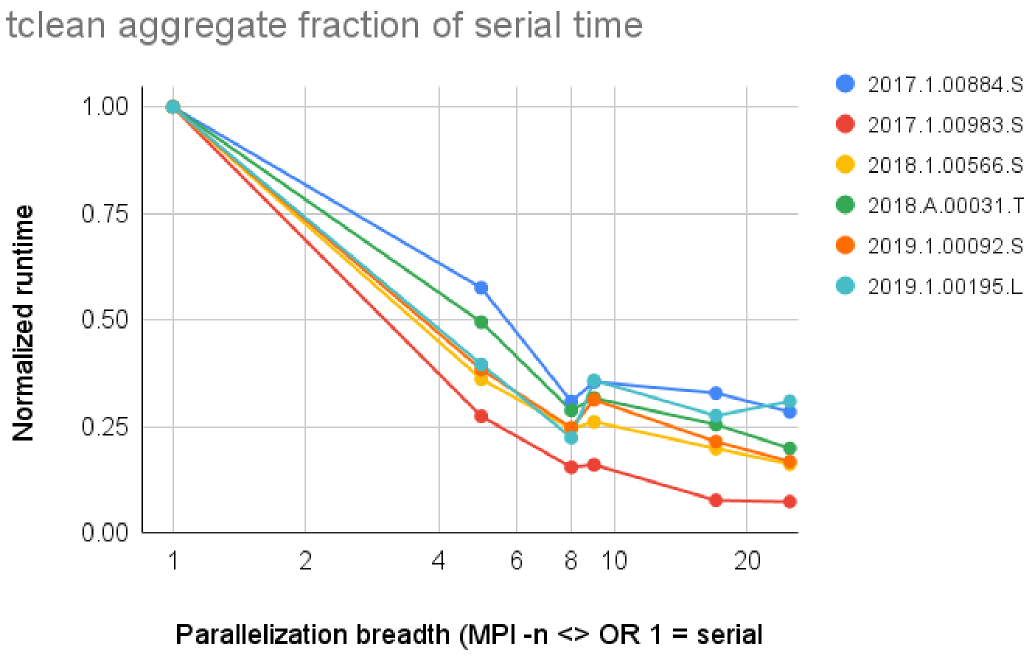
The initial testing described in Section 4.1.1 focused on increasing parallelization breadth up to the number of cores provided by a single node since, in ALMA operations, jobs are typically run on a single node. However, it is possible to run `tclean` in parallel across multiple nodes. To test how this performs in practice, we selected the three smallest data sets from our extended sample (2019.1.00876.S, 2019.1.00263.2_3477, and 2019.1.00915.S) and ran them parallelizing across 1, 2, and 3 nodes. Our tests were configured so that in each node we were using 16 cores and 500GB of RAM.

[3] Available upon request. Internal NRAO link: https://gitlab.nrao.edu/scg/pipeline_metrics

[4] Available upon request. Internal NRAO link: https://gitlab.nrao.edu/scg/readimaginglog



(a)



(b)

Figure 3. Run time of the ALMA Imaging Pipeline including tclean run time (panel a) and run time of the tclean calls within the ALMA Imaging Pipeline (panel b) as a function of increasing parallelization breadth.

Each test was run exclusively with no other jobs running on the node(s).

Figure 4 shows the Imaging Pipeline run time as a function of number of nodes used for three test data sets. Increasing the parallelization breadth to span two or more nodes does not appear to yield the expected improvements in performance. All three data sets had their run time increase when the parallelization breadth increases from 1 to 2 nodes. The run time for one of the data sets further increased from 2 to 3 nodes and the gains between 2-node parallelization and 3-node parallelization appear to be negligible for the two other data sets. These results are consistent with the results of `tclean` performance testing done by the CASA group while testing possible frameworks for a next generation CASA[5]. The lack of improvement in performance appears to be driven by write-lock and read-lock times. Table 5 shows the write-lock and read-lock times for our multi-node tests. The `tclean` run times are progressively more impacted by lock contention (time waiting on file read and/or write locks) as the parallelization breadth increases from 1 to 3 nodes.

4.2. *findcont Performance Testing*

Based on the initial investigations described in Section 4.1, we decided to test the more challenging cubes found in our extended sample described in Section 3.1.2. Running the full Imaging Pipeline for these data sets, however, would be prohibitively time consuming with estimated run times of up to several years. Running just the `findcont` stage, which identifies line-free regions in the data to use for continuum subtraction, though reduces the overall run time. This stage includes many CASA tasks used elsewhere in the Imaging Pipeline including `imstat`, `immoments`, `ia.getprofile`, and `tclean`, and thus can be used to assess the impact of these tasks on the overall Imaging Pipeline performance.

Our first result is that the size of the cubes tested here appears to trigger errors that are relatively rare for smaller cubes. During these tests, we noticed a high frequency of intermittent failures due to issues with file locking (CAS-13609). The symptom of this bug is a failed `tclean` execution with the error message "Task tclean raised an exception of class RuntimeError with the following message: Error in running Major Cycle : One or more of the cube section failed in de/gridding." and the same `tclean` command often successfully completes when re-run. This bug has been fixed in CASA 6.4.1-12 (PL2022 version), but was still present in CASA 6.2.10-7 (PL2021 version). It has been encountered in operations, but was seen relatively infrequently (20 problem report tickets in Cycle 8 out of all the MOUSes processed). Previous attempts by one of the authors (A. Kepley) to reproduce this bug by repeatedly re-running cases that failed in operations were also unsuccessful. We suspect that the larger cube sizes in our tests may have triggered this issue more consistently. For example, the project 2019.1.01074.S triggered this error repeatedly and we were unable to complete `findcont` for this data set and thus excluded it from our sample for subsequent investigations.

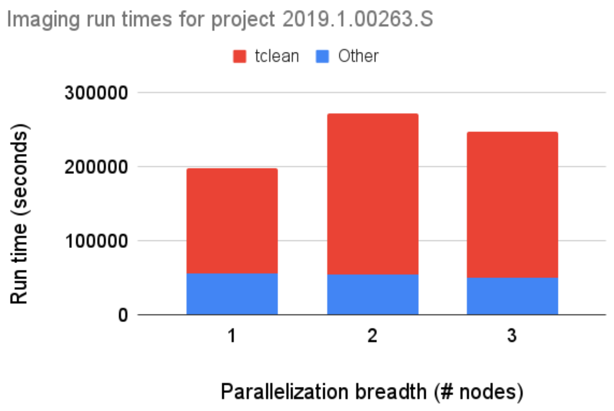
Figure 5 compares the `findcont` run time for the different MOUSes in our sample. Each data set is identified by its project code (and the last part of its MOUS UID if necessary). Three different runs were done for each data set: a mitigated run with 8 mpi (message passing interface) processes,[6] a mitigated run with 25 mpi processes, and an unmitigated run with 25 mpi processes. The total `findcont` run time was divided into the run time for the four dominant processes for `findcont`: `ia.getprofile`, `imstat`, `immoments`, and `tclean`. The task `ia.getprofile` is used to get the spectral profile for a masked region within a cube. The task `imstat` is used to calculate cube statistics and `immoments` is used to calculate moments of the cube (integrated intensity, peak intensity, etc). Finally, the `tclean` task is CASA's implementation of the well-known CLEAN algorithm. Other CASA/Pipeline tasks had a negligible contribution to the total `findcont` run times.

The first conclusion that can be drawn from Figure 5 is that the unmitigated `findcont` runs take 3 to 60 times longer than the mitigated runs for runs with the same memory and number of cores. The longest running `findcont` jobs in our sample have run times reaching 60-80 **days**! As we describe in Section 2, `findcont` is only one stage in the Imaging Pipeline, which also produces deconvolved mfs and cube images for all source/spw combinations. This result supports our initial hypothesis that running these data sets through the entire Imaging Pipeline would be prohibitively time consuming.

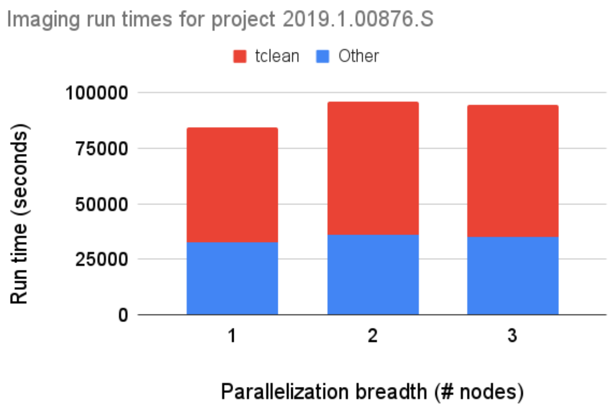
Because of the large range in `findcont` run times, Figure 6 separates the full sample shown in Figure 5 into three separate plots: panel (a) shows the unmitigated `findcont` run times greater than 10 days, panel (b) shows the unmitigated `findcont` run times greater than 1 day, and panel (c) shows the `findcont` run times

[5] https://cngi-prototype.readthedocs.io/en/latest/benchmarking.html

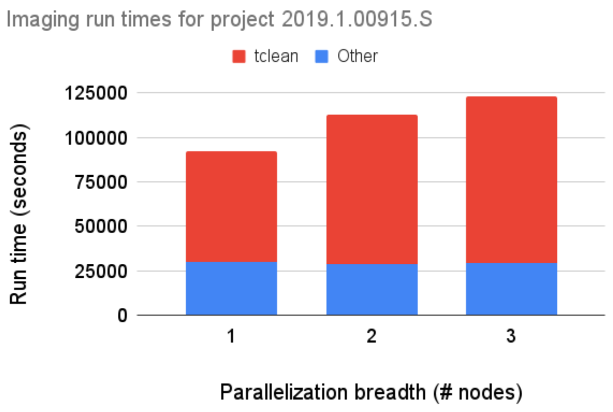
[6] The parallelization breadth is equal to the number of mpi processes.



(a)



(b)



(c)

Figure 4. ALMA Imaging Pipeline run time as a function of parallelization breadth across nodes for three projects in our extended sample: (a) 2019.1.00263.S_3477, (b) 2019.1.00876.S, and (c) 2019.1.00915.S. The run time due to `tclean` is indicated by the red portion of the bar.

Table 5. Lock Times for Multi-node Tests

	MPI		Cube tclean	Write-lock		Read-lock	
Project	Processes	Nodes	time (s)	time (s)	%	time (s)	%
2019.1.00263.S	17	1	117405	934	0.8	1839	1.6
2019.1.00263.S	32	2	196436	59580	30.3	26355	13.4
2019.1.00263.S	48	3	179508	87864	48.9	21673	12.1
2019.1.00876.S	17	1	37068	60	0.2	35	0.1
2019.1.00876.S	32	2	45511	6992	15.4	1346	3.0
2019.1.00876.S	48	3	44737	13679	30.6	3026	6.8
2019.1.00915.S	17	1	38960	218	0.6	131	0.3
2019.1.00915.S	32	2	56142	6663	11.9	857	1.5
2019.1.00915.S	48	3	60639	15867	26.2	2455	4.0

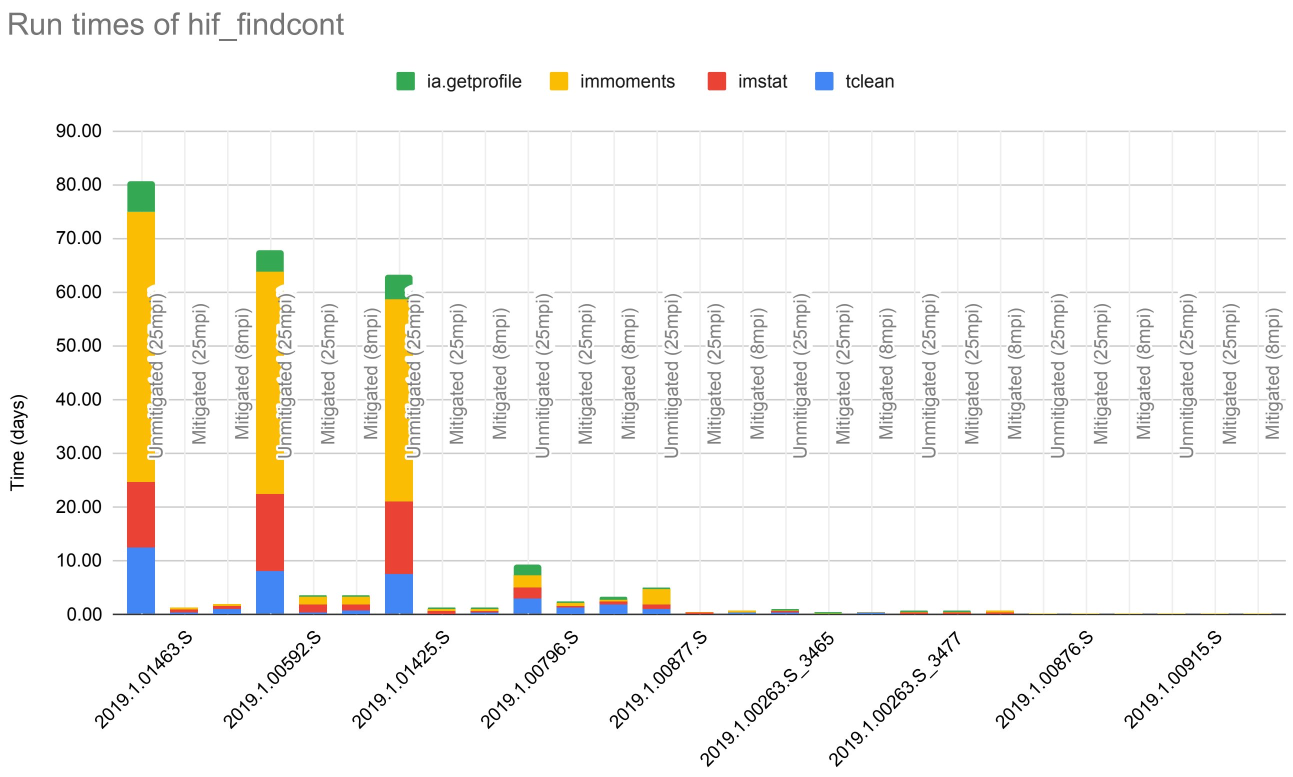
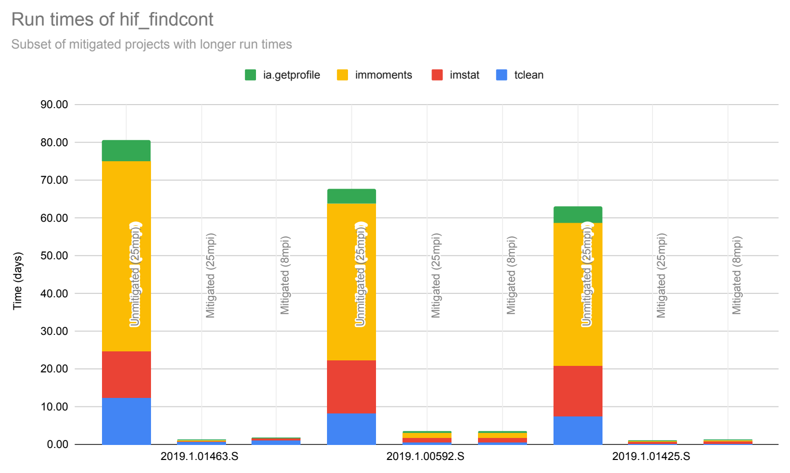
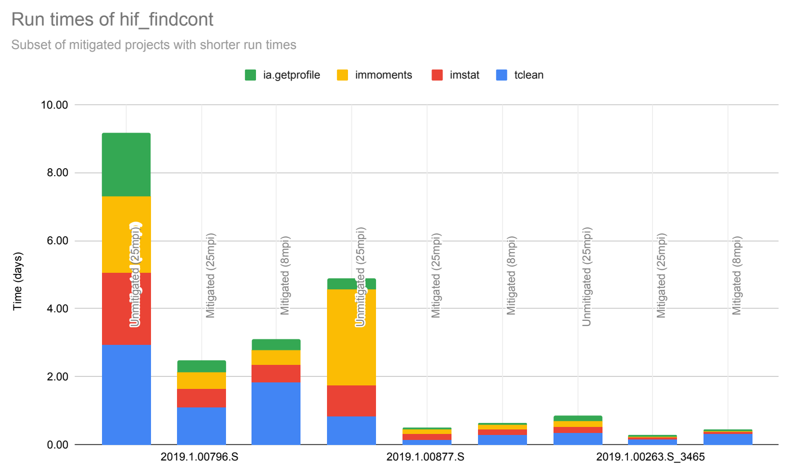


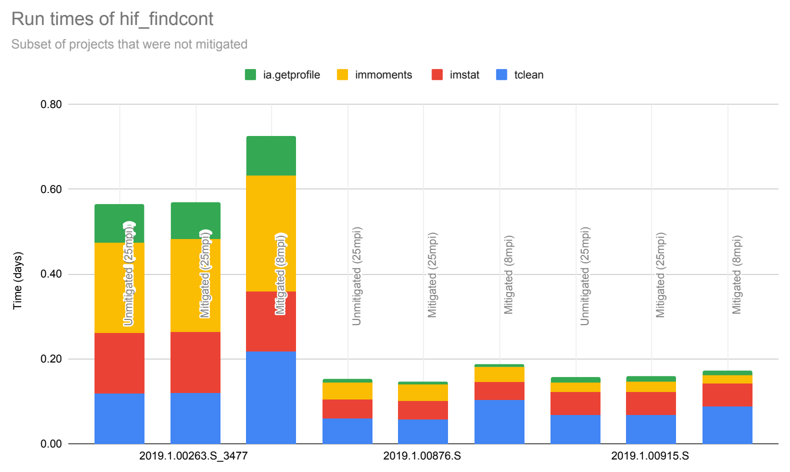
Figure 5. Run times for `findcont` stage for all MOUS in our sample. Each MOUS has three tests associated with: unmitigated with 25 mpi processes and 500 GB RAM, mitigated with 25 mpi and 500 GB RAM, and mitigated with 8 mpi processes and 250 GB RAM. The bars are broken down by time spent in four tasks: `ia.getprofile` (green, top), `immoments` (yellow, second from top), `imstat` (red, third from top), and `tclean` (blue, bottom.)



(a)



(b)



(c)

Figure 6. Run times for the `findcont` stage for the longest running MOUSes (a), the medium long MOUSes (b), and the MOUSes that were not originally mitigated (c). Each MOUS has three tests associated with it: unmitigated with 25 mpi processes and 500 GB RAM, mitigated with 25 mpi processes and 500 GB RAM, and mitigated with 8 mpi processes and 250 GB RAM. The bars are broken down by time spent in four tasks: `ia.getprofile` (green, top), `immoments` (yellow, second from top), `imstat` (red, third from top), and `tclean` (blue, bottom).

for projects that were not originally mitigated in operations. From these plots, we see that the run time of `findcont` is not dominated by the creation of a dirty cube by `tclean` as one might expect, but by the analysis tasks `ia.getprofile`, `immoments`, and `imstat`. For the unmitigated runs, these tasks typically take on order of 60-90% of the total `findcont` run time. A comparison of the mitigated run times between the 8 mpi and 25 mpi cases show that the run time of these tasks does not decrease as the parallelization breadth increases. This trend is in agreement with these tasks not making use of mpi parallelization, and in contrast to the `tclean` run time, which does decrease as expected when the parallelization breadth increases.

A subsequent investigation of the `immoments` task performance for a 425GB cube (dimensions: 7776, 7776, 1914) found that it only uses 3.6 GB of memory at a time in mpicasa[7]. Thus, `immoments` is effectively operating in serial and only processing less than 1% of the cube at one time. In serial CASA, it is possible to use multi-threading to parallelize tasks like `immoments`. Multi-threading, however, is turned off in mpicasa by setting the environment variable OMP_NUM_THREADS=1 because it interacts poorly with the mpi processing used in `tclean`. The `immoments` task does not appear to use multi-threading in serial mode. However, the `imstat` task does use multi-threading when calculating statistics over an entire cube in serial mode.

The continuum channel selection produced by `findcont` differed between the mitigated and unmitigated runs. However, since the goal of this section was to evaluate the computational performance of `findcont`, we did not investigate whether these differences lead to scientifically significant differences in the resulting cubes.

4.3. *Scaling results*

In this section, we take a closer look at the scaling performance of `tclean` with respect to three key quantities: number of channels, number of mosaic pointings, and the number of input measurement sets. The first two quantities impact the gridding and deconvolution in `tclean`. The latter is interesting because each time a scheduling block is observed on ALMA it creates a separate measurement set (often referred to as an Execution Block or EB). The `tclean` task then takes a list of measurement sets that it has to grid together rather than combining them into a single ms file. This process has the potential to cause a difference in performance.

[7] Private communication from R. Loomis on CASR-678

Since running the full Imaging Pipeline was not feasible for our data sets, we used a special purpose build of the Pipeline that generates Pipeline-like tclean commands for the continuum subtracted data set, but does not execute them. We note that not doing continuum subtraction would potentially result in different deconvolution performance due to broadband emission in the cube. The resulting `tclean` commands did not include suitable values for niter and threshold because those heuristics require dirty images and the `tclean` commands were generated prior to the creation of a dirty image. To set a reasonable value for these parameters, we estimated the values for niter and threshold based on the dirty image from `findcont` and the ALMA Pipeline heuristics using the procedure specified in Appendix B. To determine the imsize for the number of pointings scaling tests, we split off the relevant pointings and used the Pipeline to determine what the imsize would be for the split measurement set.

4.3.1. *Scaling with Number of Channels*

We expect that the scaling for `tclean` as a whole as well as the major and minor cycles should increase linearly as a function of number of channels. The major cycle grids the data and this operation depends linearly on the number of channels, while the minor cycle deconvolves the resulting cube which again depends linearly on the number of channels.

To explore how `tclean` run time scales with numbers of channels, we selected four different data sets: two mosaics (2019.1.00915.S and 2019.1.00263_X3465) and two single pointings (2019.1.01463.S and 2019.1.00592.S). We selected mosaics and single pointings because they use two different gridders within `tclean`. For each data set, we started with imaging all the channels and then decreasing the number of channels imaged by a factor of 2 for each subsequent `tclean` run. We included the brightest line emission region found by `tclean` in all cubes, since that will dominate the deconvolution.

Figure 7 shows the relative average total `tclean` run time, the average major cycle run time, and the average minor cycle run time as a function of number of channels. A linear trend (not a fit) is shown as an orange line. For 2019.1.01463.S, the tests for 3840 and 1920 channels repeatedly ran out of memory in the `auto-multithresh` portion of the minor cycle and thus are not included in this plot. This issue has been reported in CAS-13898.

On the whole, the `tclean` run time and the run times for the major and minor cycles scale linearly with the number of channels. The exception is the MOUS with the largest number of channels, 2019.1.00952.S, where the `tclean` run time is twice that expected. The difference is greater for the major cycle (factor of 2.3), but still present in the minor cycle (factor of 1.7). This run used a considerable amount of swap and that may have affected the performance. It may be related to the large amount of swapping seen in auto-multithresh (CAS-13898), but the time frames when swapping occurred were not always related to `auto-multithresh`. Thus we have reported it separately in CAS-14142.

4.3.2. *Scaling with Number of Pointings*

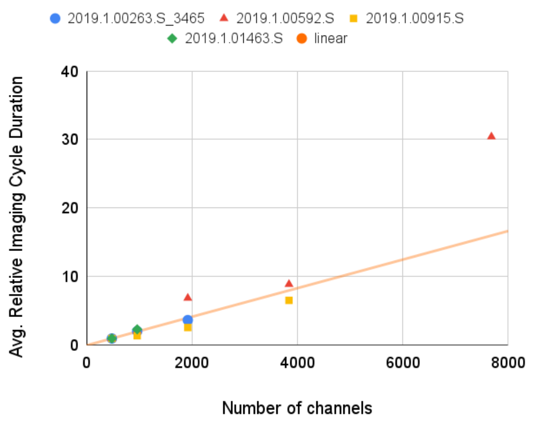
Another important aspect of `tclean` performance to investigate is the scaling with number of mosaic pointings imaged. There are two effects at play here. First, increasing the number of mosaic pointings will linearly increase the number of visibilities. Second, it will also increase the size of the gridding kernel quadratically. Thus we expect the performance of the gridding as a function of number of pointings to be quadratic. Since the deconvolution step depends on the number of pixels, we expect that to scale linearly with the number of pixels.

Figure 8 shows the scaling of the average imaging cycle duration, average major cycle duration and average minor cycle duration for two test data sets (2019.1.00263.S_34465 and 2019.1.00263S_3477). The average major cycle duration displays a quadratic dependence, as expected. The average minor cycle duration scales linearly with the number of pointings. Figure 9 shows that the image size scales linearly with the number of pointings, so this is again as expected. The trend for the average total imaging cycle duration is then just a combination of the quadratic major cycle trend and the linear minor cycle trend.

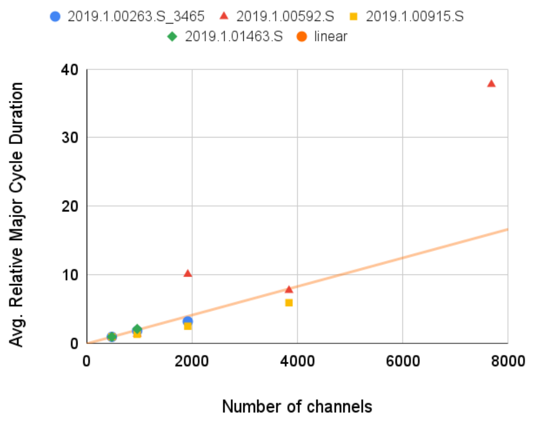
4.3.3. *Scaling with Number of Execution Blocks*

In ALMA operations, the scheduling block specifying the observation requested by the PI is executed one or more times producing one measurement set for each execution, which are referred to as execution blocks (EBs). The execution blocks are input into `tclean` as a list and combined within `tclean` to produce a single cube containing all the visibility data. Thus we expect that the run time of the major cycle in `tclean`, which does the gridding, to increase linearly with the number of execution blocks since a linear increase in the number of execution blocks should linearly increase the number of visibilities[8]. The run time of the minor cycle in `tclean`, which operates in image cube space should not be affected by an increase in the number of execution blocks.

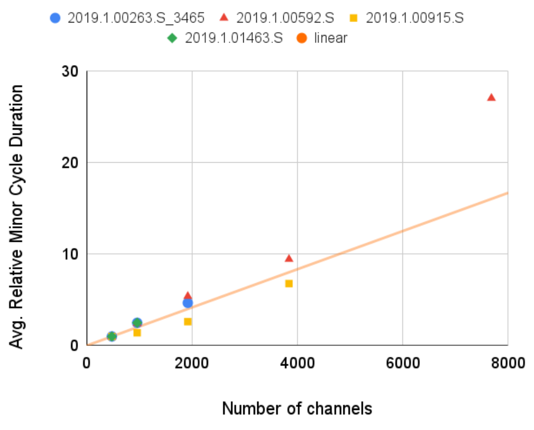
[8] Each execution block will have a similar number of visibilities since they are all generated from a single scheduling block.



(a)

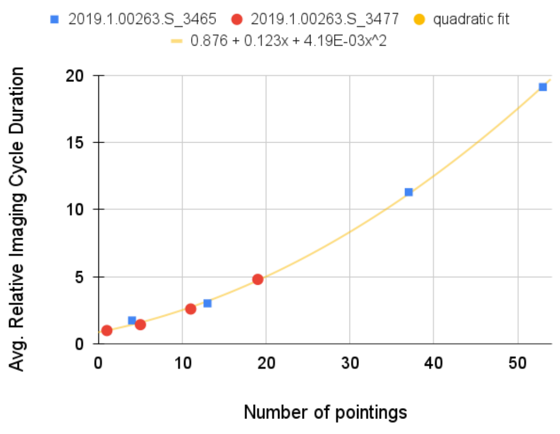


(b)

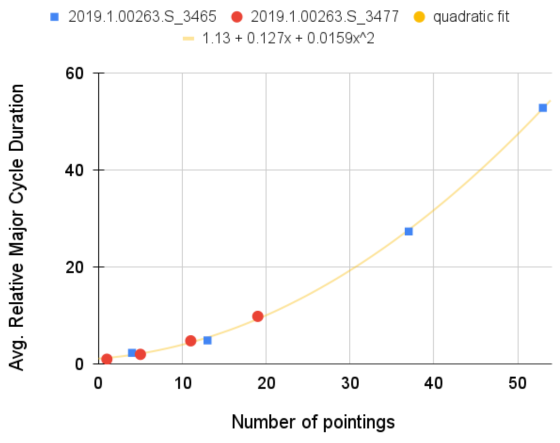


(c)

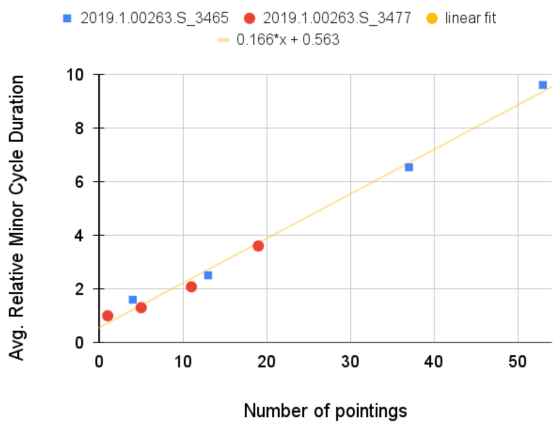
Figure 7. The scaling of `tclean` run time as a function of number of channels for the average relative total imaging cycle duration (panel a), average relative major cycle duration (panel b), and average relative minor cycle duration (panel c). The data sets are indicated using different symbols and a linear trend (not a fit) is shown as an orange line. For 2019.1.01463.S, the tests for 3840 and 1920 channels failed repeated due to running out of memory in the `auto-multithresh` step in the minor cycle and thus are not shown in this plot.



(a)



(b)



(c)

Figure 8. The scaling of the `tclean` run time as a function of number of pointings for the average relative total imaging cycle duration (panel a), average relative major cycle duration (panel b), and average relative minor cycle duration (panel c). The data sets are indicated using different symbols and a quadratic fit to the data is shown as an orange line.

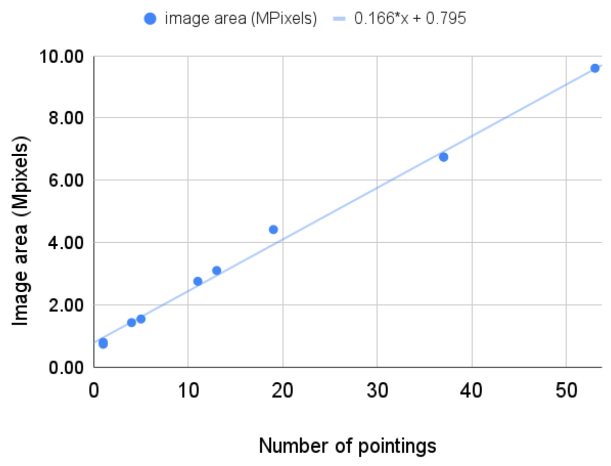


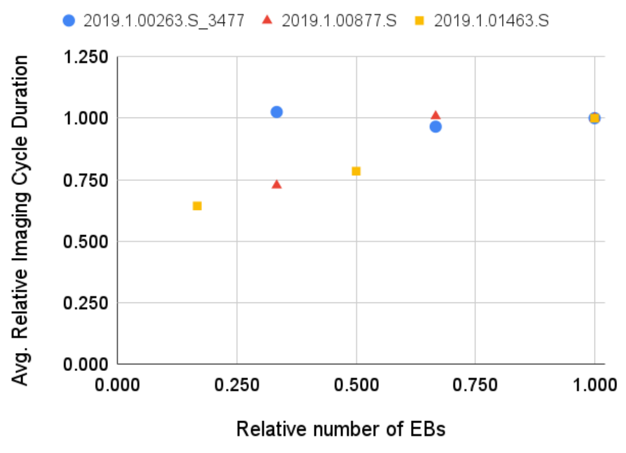
Figure 9. The imaging area as a function of number of pointings for the scaling tests with number of pointings. A linear trend is shown by the blue line.

To test this scaling, we selected data sets from our extended sample described in Section 3.1.2 that had multiple executions. We then ran these data sets with all execution blocks as well as a reduced number of execution blocks. For the test with reduced numbers of execution blocks, we scaled the threshold value used for each test by scaling by the square root of the ratio between the total number of execution blocks and the current number of execution blocks.

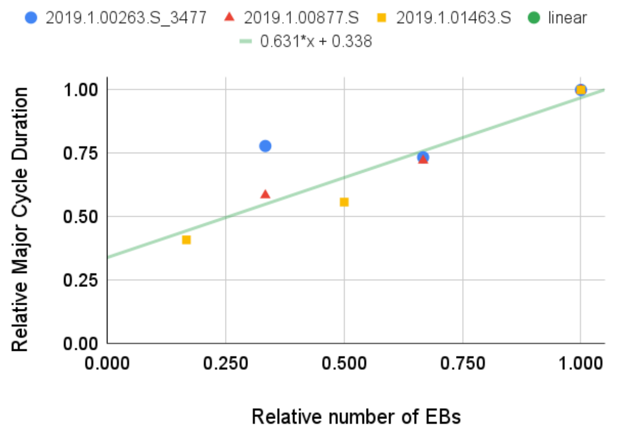
Figure 10 shows relative run times of our test data sets as a function of the relative number of execution blocks. As anticipated, changing the number of execution blocks does not have a significant effect on the run time of the minor cycle. The major cycle does display a linear trend, but the slope is 0.6 which is less than the expected slope of 1. There also appears to be a constant offset, suggesting some fixed overhead for each major cycle, independent of the number of execution blocks.

4.4. *Relative performance of the major and minor cycles*

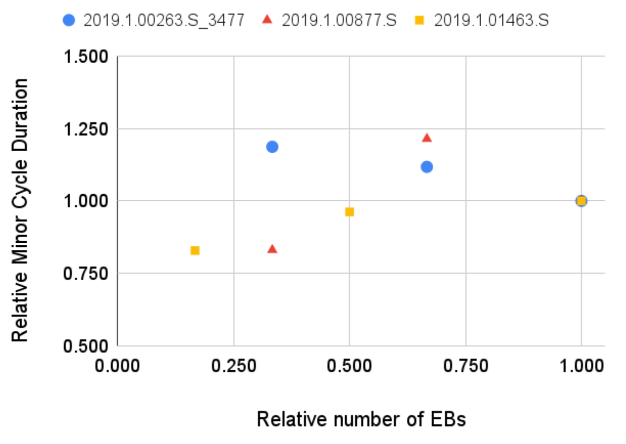
One commonly used assumption in theoretical calculations of imaging performance is that the major cycle time, which does the gridding of the visibilities, dominates over the minor cycle, which does the deconvolution. In this section, we compare the run times of the major and minor cycle to test whether this assumption holds true for the unmitigated ALMA cubes. Figure 11 shows the ratio of the average major and average minor cycle as a function of number of channels, number of pointings, and number of execution blocks. In most cases, we find that the minor cycle time dominates over the major cycle time with one exception:



(a)

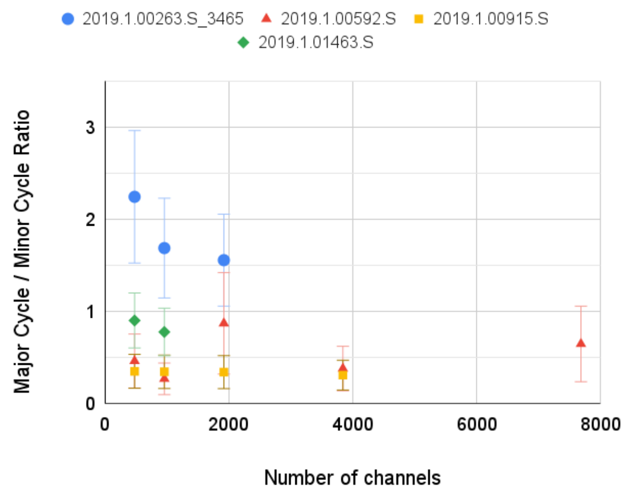


(b)

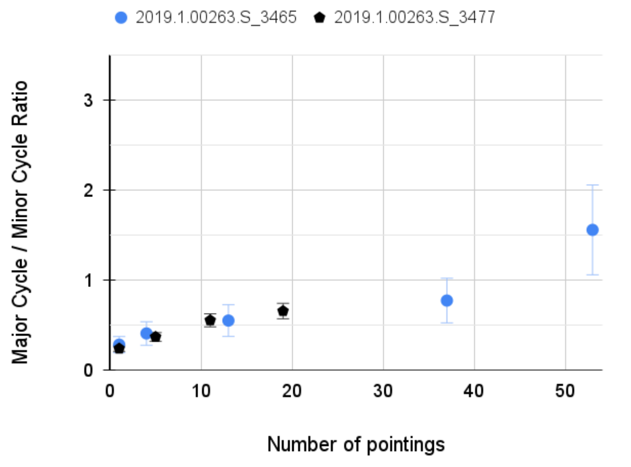


(c)

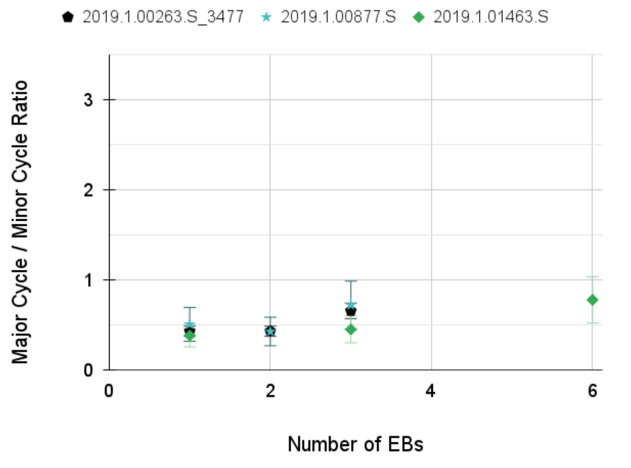
Figure 10. The relative duration as a function of the relative number of execution blocks for the entire timing cycle (panel a), the major cycle (panel b), and the minor cycle (panel c). Panel (b) includes trend line for the minor cycle points.



(a)

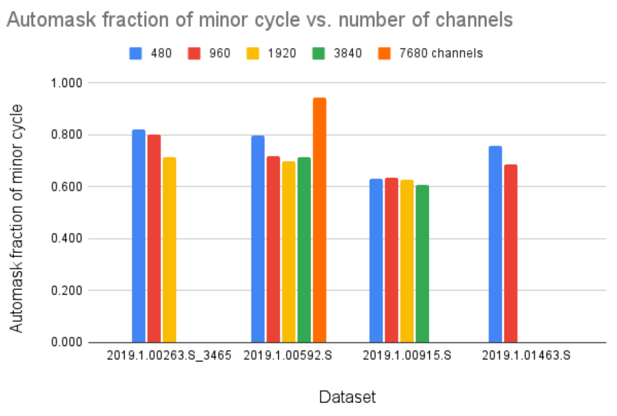


(b)

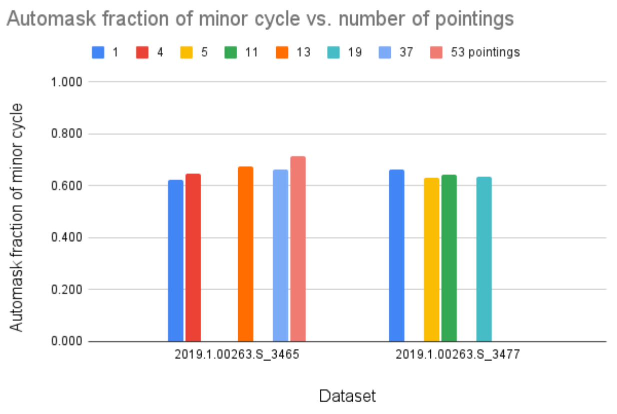


(c)

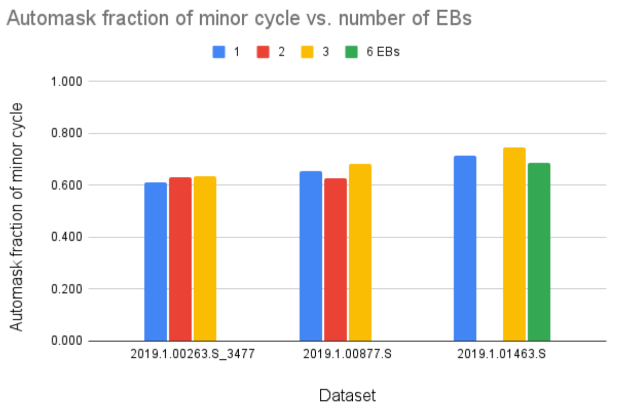
Figure 11. The ratio of the average major cycle run time to the average minor cycle run time as a function of number of channels (a), number of pointings (b), and number of executions (c). The error bars indicate the error in the ratio over all cycles for each data set.



(a)



(b)



(c)

Figure 12. The fraction of the minor cycle taken up by the automated masking done by `auto-multithresh` as a function of number of channels (a), number of pointings (b), and number of execution blocks (c).

2019.1.00263.S_3465. This project is a mosaic with the largest number of pointings in our sample, although it does not have the largest image size for a mosaic in our sample. As the number of pointings increases for this project, the ratio of the major cycle to the minor cycle increases to finally reach a value above 1. We note that the other mosaic test case shows a similar trend with number of pointings, but does not exceed 1. As the number of channels increases for 2019.1.00263.S_3465, the ratio of the major to minor cycle decreases, but is never below 1.

The underlying reason why the minor cycle dominates over the major cycle for these tests is due to the `auto-multithresh` algorithm (Kepley et al. 2020) used by the ALMA Imaging Pipeline to automatically mask emission during the minor cycle to enable deeper cleans. This algorithm identifies significant emission peaks in the residual image and cascades the mask down to lower signal to noise level. The mask is recalculated every minor cycle for every channel, unless a mask is not found in the first minor cycle for a channel or the mask changes by less than 10%. The latter behavior can be varied by changing an input parameter. Figure 12 shows the fraction of the minor cycle time spent on `auto-multithresh` for the tested data sets. We find that `auto-multithresh` is at least 60% of the minor cycle run time and can take up to 90% of the minor cycle run time. There is some indication that the `auto-multithresh` fraction decreases with increasing number of channels, although we do have one extreme case with 7680 channels where `auto-multithresh` took 90% of the minor cycle compared to 70-80% for smaller number of channels. This case (2019.1.00592.S) is a large single field mosaic with the anomalously large imaging cycle duration noted in Section 4.3.1. There does not appear to be a strong variation in `auto-multithresh` time with number of pointings or number of execution blocks.

4.5. *NVMe experiments*

In operations, ALMA data processing uses the Lustre file system for data storage. Lustre is a shared, distributed file system comprised of dozens of arrays of drives called OSTs. At NRAO, Lustre is configured to write one file per OST and each OST has a write-cache memory. This configuration means that our Lustre file systems are able to write dozens of files in parallel. However, most modern high throughput computing clusters compartmentalize their data by transferring the relevant data to a drive local to the processing node rather than having all data accessible to the processing node all the time (as is the case with Lustre). Many of these clusters use NVMe drives for data storage for processing. We performed the same single, 500GB file benchmark tests on a Samsung PM1735 NVMe drive and saw sequential reads averaging 1,694MB/s and sequential writes averaging 2,068MB/s. This is 2.4 times faster than our NAASC Lustre benchmark tests. Therefore, reading and writing a large file can be much faster when done using NVMe instead of Lustre. In addition, since only a small number of jobs can be run on a node because of the large memory requirements per job, the number of jobs trying to access the file system is reduced for NVMe compared to Lustre. However, individual NVMes are limited in size. The 6.4 TB NVMes used in our testing represent the mid-range of the storage volumes available on the commercial market with top-end drives having 15TB of storage at the time of purchase in early 2022.

In this section, we explore whether using NVMe drives instead of Lustre as data storage could improve the run time performance of `tclean`. Initial tests of NVMe performance were done using three data sets from our initial sample (see Section 3.1.1) and the NVMe devices associated with the VLASS nodes at both NMT and DSOC (see Section 3.2 for details). The aggregate `tclean` duration for two different parallelization breadths (8 and 9) are shown in Figure 13. One data set (2017.1.00983.S) showed a 30% decrease in run time when using the NVMe instead of Lustre, but the performance was similar between Lustre and NVMe for the two other data sets (2018.1.00566.S and 2019.1.00092.S).

For our second round of NVMe tests, we used the unmitigated data sets taken from the extended sample (see Section 3.1.2) and the NVMe device on the PLWG node cvpost127 (see Section 3.2). We were restricted to choosing data sets that would fit on the NVMe device ($\lesssim$6.4 TB). Out of these data sets, we selected two data sets for testing that trigger different gridder modes: one single pointing (2019.1.00592.S) and one mosaic (2019.1.00263.S_3465). We then ran the same `tclean` commands as in our previous testing using both the NVMe and Lustre as our data storage. Figure 14 shows the average imaging cycle duration, the average major cycle duration, and the average minor cycle duration for two different data sets. The single pointing data set saw a 53% reduction in run time, but the mosaic data set showed comparable run times between Lustre and NVMe.

To explore the lack of improvement in the mosaic data, we imaged the mosaic data with 1, 4, 13, 37, and 53 pointings and looked at the average imaging cycle duration, the average major cycle time, and the average minor cycle time. The results are shown in Figure 15.

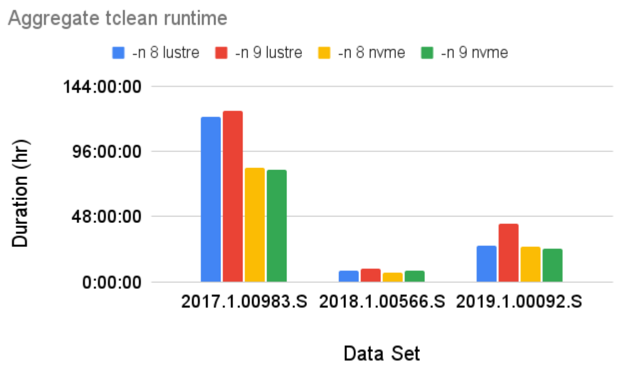


Figure 13. Aggregate `tclean` run time for Lustre and NVMe for two different parallelization breadths (8 and 9). Shorter bars are better.

For the minor cycle, the NVMe durations are always shorter than Lustre. For the major cycle, the NVMe run times are shorter than the Lustre run times for our tests including up to 13 pointings. Beyond 13 pointings, the NVMe average major cycle run time is longer than the Lustre average major cycle run time. This increase in the major cycle run times for large numbers of pointings for the NVMe devices is reflected in the average total imaging cycle duration.

In CASA 6.2.1, the mosaic gridder was found to be doing unnecessary IO (CAS-13991) which may have contributed to the relative slowness of the NVMe compared to Lustre for 2019.1.00263.S_3465. We re-ran our tests using a CASA build incorporating a fix for the issues seen in CAS-13991 and found that while both the NVMe and Lustre run times improved, the ratio of the run times remained the same. Thus we do not think that the unnecessary IO for mosaics identified in CAS-13991 is contributing to the results described above.

5. SUMMARY AND FUTURE WORK

In this memo, we have investigated the performance of the ALMA Imaging Pipeline and its underlying data processing software, CASA, for the largest cubes that ALMA can produce. These cubes are currently mitigated, or reduced in size, when imaged in operations. They have their fields of view, number of pixels per beam, and/or channels binned by 2. Characterizing the performance of the Pipeline and CASA when fully imaging these challenging cubes, however, is a key step towards being able to process cubes produced by the ALMA WSU. This upgrade will increase the maximum number of channels per spectral window by approximately 1.5 orders of magnitude, further increasing the number of large cubes produced by ALMA.

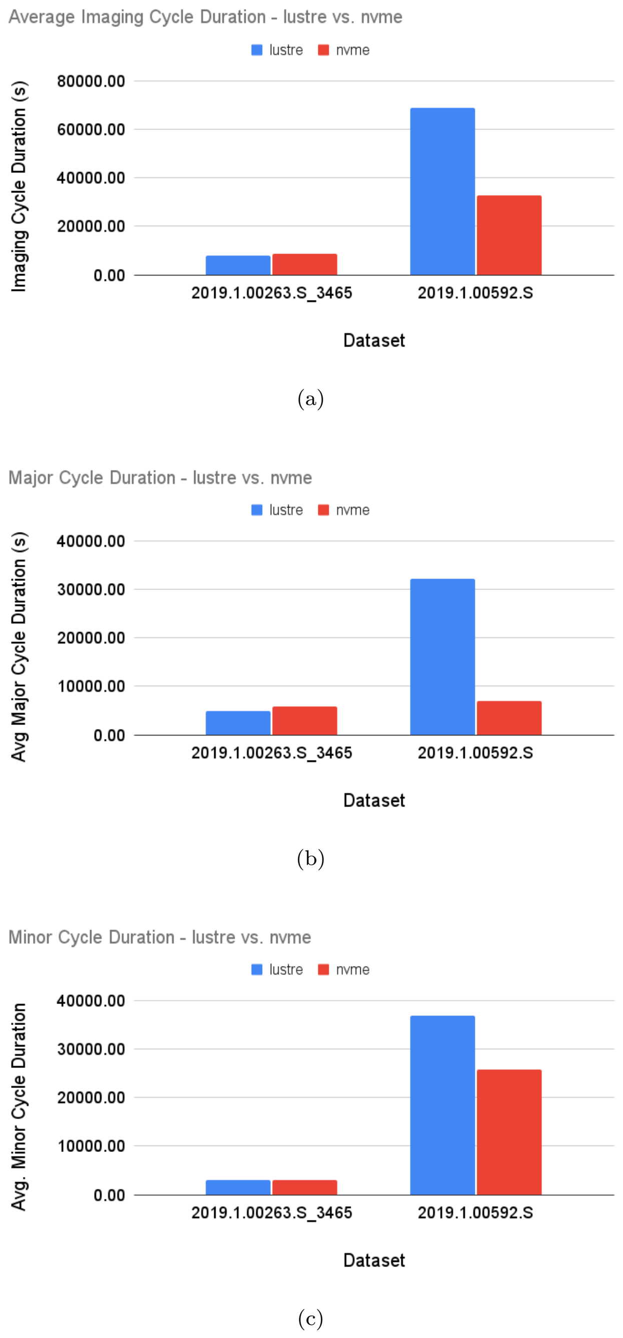


Figure 14. Average total imaging cycle duration (panel a), average major cycle duration (panel b), and average minor cycle duration (panel c) for two test data sets using both Lustre and NVMe for data storage backends. Shorter bars are better.

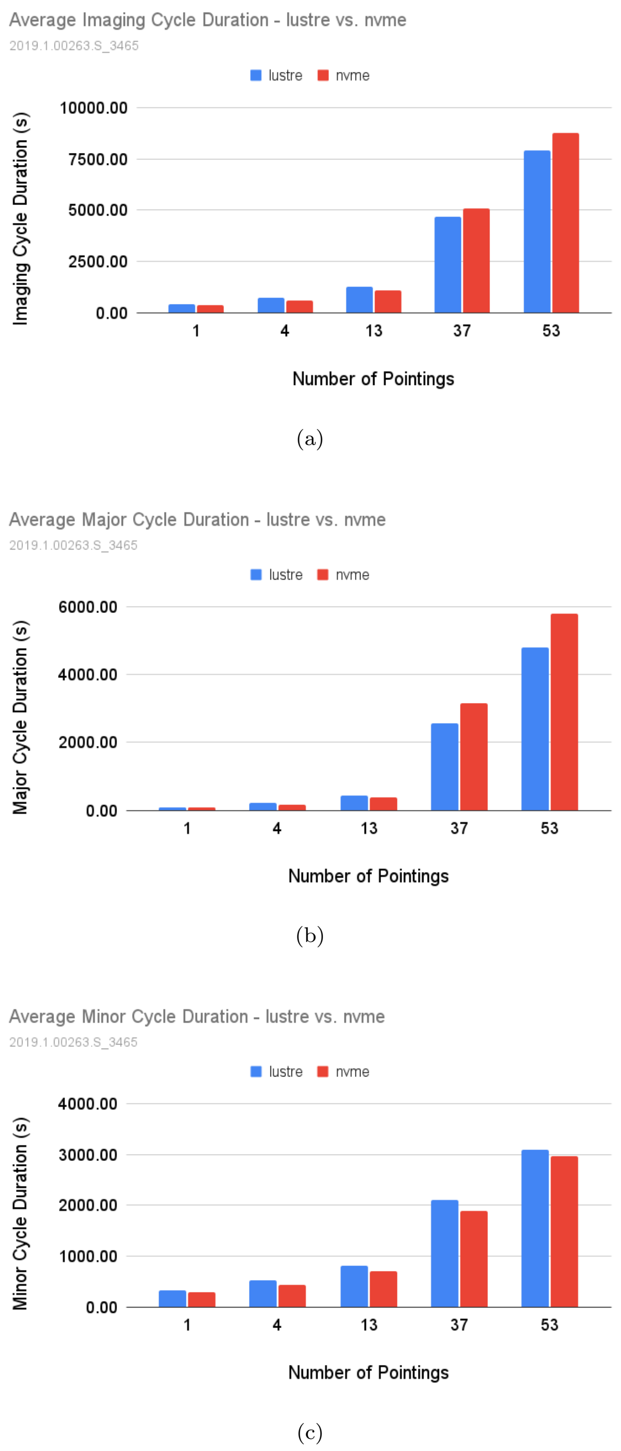


Figure 15. Average total imaging cycle duration (panel a), average major cycle duration (panel b), and average minor cycle duration (panel c) for 2019.1.00263.S_3465 as a function of number of pointings imaged for both Lustre and NVMe. Shorter bars are better.

We have split our testing into four main parts. First, we explored the parallelization breadth for the ALMA Imaging Pipeline and `tclean` on a single node and across multiple nodes. For parallelization breadths up to that provided one node, we found that the Imaging Pipeline performance does not scale with parallelization breadth for all projects, but the performance of tclean largely does up to a breadth of 17. Neither the Imaging Pipeline performance nor the tclean performance scales with parallelization breadths that span two or more nodes.

Second, we investigated the performance of the findcont stage of the Pipeline, which identifies line-free regions within a spectra to use for continuum subtraction. Since running the full Imaging Pipeline would have been too time-consuming for our largest cubes, this stage was selected for detailed testing because it uses CASA tasks that are used frequently in other portions of the Imaging Pipeline including imstat, immoments, ia.getprofile, and tclean. We find that running the findcont step on unmitigated data takes 3 to 60 times longer than running it on unmitigated data, with the largest cube taking up to 80 days. The findcont run time is dominated by the tasks immoments, imstat, and ia.getprofile, which take 60-90% of the time, rather than the creation of the dirty cube by tclean.

Next we transitioned to a detailed investigation of the scaling of tclean performance with number of channels, number of pointings, and number of input measurement sets (i.e., execution blocks). The first two axes were chosen because they represent fundamental properties of the data. The number of execution blocks was chosen because it reflects the total number of visibilities used to create the cube as well as a key operational ALMA use case for tclean. We find that the scaling with number of channels and number of pointing for tclean is as expected. The performance scales linearly with number of channels and quadaratically with number of pointings. For one of our largest cubes, however, we repeatedly ran out of memory during the automated mask creation portion of the minor cycle (auto-multithresh). This issue has been reported in the ticket CAS-13898. The performance for number of execution blocks appears to have a constant overhead and scales sub-linearly with the number of execution blocks.

For each of the scaling tests, we compared the run times of the major and minor cycles within tclean to investigate whether the major cycle dominates over the minor cycle as typically assumed. We found that the opposite is true: minor cycle run time dominates over the major cycle run time for most of our unmitigated imaging runs. The most time-consuming portion of the minor cycle is the automatic masking of emission done by the auto-multithresh algorithm with the masking taking between 60 to 80% of the minor cycle.

Lastly, we investigated whether using NVMe instead of Lustre for our data storage would improve the performance of tclean. A local NVMe drive can read and write 2 to 3 times faster than a Lustre OST, but is limited in size to only a few TB. We found that some of our test cases experienced 30-53% reductions in overall run time, but that other showed comparable run times between NVMe and Lustre. For the mosaic gridder, the NVMe major cycle run time slowed down compared to the Lustre major cycle run time as the number of pointings imaged increased. For the minor cycle, the NVMe test run times were always faster than the Lustre test run times.

These results suggest several avenues for further investigation for imaging current unmitigated ALMA cubes and future imaging of the WSU cubes. The first and most obvious are related to current and upcoming bug fixes in CASA. We should re-run our number of channel scaling tests for 2019.1.01463.S once a fix for the memory issues that caused them to fail is in place (CAS-13898). In addition, a detailed investigation of NVMe performance for mosaics would be useful to understand why one of our mosaics showed little to no run time improvement when run on an NVMe compared to Lustre.

Some more forward looking investigations would be to investigate whether it would be possible to parallelize imstat, immoments, and ia.getprofile within mpicasa. Parallelizing these tasks would significantly decrease the run time of the findcont stage for large cubes. CASA does have an interface to manipulate the number of threads within mpicasa. However, care would have to be taken that the number of OMP threads is set to 1 during any tclean executions. These tests could be done on both the Lustre file system and using an NVMe device as the data storage.

Finally, we could experiment with increasing the computing throughput of current CASA by running the individual steps within the larger monolithic tclean task with appropriately partitioned data. There are prototypes that could be used for these experiments (e.g., htclean developed by F. Madsen)[9]. Another, simpler, version of this approach is already being used by some groups to image large cubes. These groups use the current CASA tasks to align the frequency axes of individual measurement sets (cvel2 or mstransform), combine the measurement sets into one measurement set (concat), split off each channel (split or mstransform) and image it using tclean, and then recombine the channels into a final cube containing all channels. These experiments, however, may be more appropriate for the ngCASA framework.

[9] https://open-confluence.nrao.edu/display/SCG/htclean+-+a+distributed+approach+to+running+tclean+on+high+throughput+computing+environments

We would like to thank the following people: Dirk Muders for creating a special build of the ALMA Pipeline that allowed us to extract the `tclean` commands without having to run a full clean, Ryan Loomis for doing performance testing of the `immoments` task, and Crystal Brogan for providing the initial test data set sample. We would also like to thank the ALMA Pipeline Working Group for allowing us use of their cluster nodes for testing.

REFERENCES

ALMA Pipeline Team. 2022, ALMA Science Pipeline User's Guide for Release 2022.2, CASA 6.4.1 Interferometric and Single-Dish Processing, Tech. rep. http://www.almascience.org.

Carpenter, J., Brogan, C., Iono, D., & Mroczkowski, T. 2022, ALMA Memo Series, 621. https://library.nrao.edu/public/memos/alma/main/memo621.pdf

Cortes, P. C., Remijan, A., Hales, A., et al. 2022, ALMA Cycle 9 Technical Handbook. www.almascience.org.

Hunter, T. R., Indebetouw, R., Brogan, C. L., et al. 2023, Publications of the Astronomical Society of the Pacific, 135, 074501, doi: 10.1088/1538-3873/ace216

Kepley, A. A., Lipnicky, A., Rao Venkata, U., & Indebetouw, R. 2023, ALMA Memo Series, 263. https://library.nrao.edu/public/memos/alma/main/memo623.pdf

Kepley, A. A., Tsutsumi, T., Brogan, C. L., et al. 2020, Publications of the Astronomical Society of the Pacific, 132, 024505, doi: 10.1088/1538-3873/ab5e14

THE CASA TEAM, Bean, B., Bhatnagar, S., et al. 2022, PASP, 114501, doi: 10.1088/1538-3873/ac9642

APPENDIX

A. ALMA PIPELINE MITIGATION HEURISTIC

Below we have copied the mitigation heuristic used in the ALMA Pipeline as of PL2022 (Cycle 9 Pipeline) from the ALMA Pipeline User's Guide (ALMA Pipeline Team 2022).

Step 1: If cubesize > maxcubesize, for each spw that exceeds maxcubesize:

a. If (nchan == 3840) or (nchan in (1920, 960, 480) AND online channel averaging was NOT already performed), then set nbin=2.

b. If still too large, then calculate the Gaussian primary beam (PB) response level at which the largest cube size of all targets is equal to the maximum allowed cube size. The cube sizes are initially calculated at primary beam power level PB=0.2. For an image of width d, the response level at the edge will be PB=exp(-d^2*ln(2)/FWHM^2), the image size $d^2 \propto$ -ln(PB), and the required power level to create an image of size = maxcubesize is:

$$\mathrm{PB\_mitigation} = \exp(\ln(0.2) * \mathrm{maxcubesize} / \mathrm{current\_cubesize})$$

   i. Then account for imsize padding: PB_mitigation = 1.02 * PB_mitigation

   ii. Then limit the size reduction to PB=0.7: PB_mitigation = min(PB_mitigation, 0.7)

   iii. Then round to 2 significant digits: PB_mitigation = round(PB_mitigation, 2)

   NOTE: this mitigation cannot be applied to mosaics, only single fields, and the same mitigated FoV is used for all science target image products.

c. If still too large, change the pixels per beam (cell size) from 5 to 3.25 (if robust=+2) or 3.0 otherwise.

d. If still too large, *stop with error*, the largest size cube(s) cannot be mitigated.

Step 2: If productsize > maxproductsize

a. If the number of science targets (single fields or mosaics) is greater than 1, reduce the number of targets to be imaged until productsize < maxproductsize. The representative target is always retained.

b. If productsize still too large, repeat steps 1a, 1b, and 1c, recalculating productsize each time.

c. If productsize is still large, *stop with error*, the productsize cannot be mitigated.

Step 3: For projects with large cubes that can be mitigated, restrict the number of large cubes that will be cleaned:

a. If there are cubes with sizes greater than 0.5 * maxcubelimit, limit the number of large cubes to be cleaned to 1. The spw encompassing the representative frequency shall always be among the cubes retained.

Step 4: For projects that have many science targets, limit the number to be imaged to 30, the representative target is always retained in the list. *This statement is not consistent which what is actually done in the Pipeline, which is to issue a warning if there are more than 30 sources and the aggregate number of channels across all spectral windows is greater than 960.*

B. ESTIMATING VALUES FOR THE THRESHOLD AND NUMBER OF ITERATIONS

The ALMA Pipeline heuristics use the theoretical RMS and the dynamic range of the dirty image to set the threshold. We estimated the theoretical RMS by determining the median absolute deviation in the signal-free `findcont` region and converting to an RMS by multiplying by 1.4826. The dirty dynamic range was calculated by taking the peak within the signal `findcont` region and dividing by the theoretical RMS. The dynamic range corrected threshold was then calculated using the same heuristics as calculated for the 12-m Array[10]:

```
def dr_correction(threshold, dirty_dr, residual_max, tlimit):
    n_dr_max = 2.5
    if dirty_dr > 150.:
        maxSciEDR = 150.0
        new_threshold = max(n_dr_max * threshold, residual_max/maxSciEDR * tlimit)
    else:
        if dirty_dr > 100.0:
            n_dr = 2.5
        elif 50. < dirty_dr <= 100.:
            n_dr = 2.0
        elif 20. < dirty_dr <= 50.:
            n_dr = 1.5
        elif dirty_dr <= 20.:
            n_dr = 1.0
        new_threshold = threshold * n_dr

    return new_threshold
```

In the above residual_max is the peak of the dirty cube and tlimit is set to 2.0 (the Pipeline default). The number of iterations was calculated using

```
def niter_correction(cell, imsize, residual_max, threshold, mask_frac_rad=0.45):

    kappa = 5
    loop_gain=0.1

    r_mask = mask_frac_rad * max(imsize[0],imsize[1]) *  cell
    beam = 5.0 * cell

    niter = int(kappa/loop_gain * (r_mask/beam)**2 * residual_max/threshold)

    # avoid tclean overflow issue
    niter = min(niter,2**31-1)

    return niter
```

We compared the results of the above calculations with the results from several Pipeline runs and in general found them to be in good agreement.

[10] All our test data are 12-m.