***Title*:** Alma pipeline benchmarking and profiling report #2	***Author*:** Felipe Madsen, James Robnett, K. Scott Rowe	***Date*:** 10/30/2019
NRAO Doc. #:		***Version*:**

Alma pipeline benchmarking and profiling report #2

PREPARED BY	ORGANIZATION	DATE
Felipe Madsen, James Robnett, K. Scott Rowe	NRAO	10/30/2019

Change Record

VERSION	DATE	REASON

TABLE OF CONTENTS

1 INTRODUCTION

This is the second in a series of reports on the profiling and benchmarking of the ALMA pipelines, an ongoing work conducted by the Scientific Computing Group at NRAO. After the Phase 1 report that introduced the profiling framework and the methodology of this work, we conducted a series of tests to investigate the effects of resource selection and contention on run time. We focused primarily on imaging rather than calibration as the former was shown on the first report to be the more expensive pipeline.

The tests conducted in this work were indicated as next steps in the conclusions of the Phase 1 report, consisting of a more in-depth investigation of the ALMA pipelines. In this context we present and discuss results for tests with different parallelization configurations (varying number of MPI processes and OpenMP threads), local storage types (NVMe and RAID0 array) vs. Lustre, memory limits (what is the effect of constraining memory?), and job concurrency (how does it affect throughput?).

The tests reported and analyzed in this document were run on casa-release-5.4.0-68 for consistency with the results from the first report, so that phase 1 and phase 2 tests could be compared. We will run a series of tests with a more recent version of CASA on the next phase of this work.

Confirming the scientific correctness of pipeline products was outside the scope of the work done. Comparisons with baseline products were performed to ensure there were no macro scale issues, but the testing process should not be viewed as sensitive to subtle imaging errors.

2 UPDATED PROFILING FRAMEWORK

To support profiling in a parallel context, the profiling framework from Phase 1 was modified to gather discrete memory, number and duration of system calls, and file descriptor counts from each MPI process. I/O shape and counts per pipeline execution versus Lustre were recorded to characterize parallelization effects on I/O. We developed a CASA log analysis tool to measure major and minor cycle times on tclean() calls. We also developed scripts that transparently redirect PPR-based pipeline submissions from local clusters to Amazon Web Services (AWS) and return the complete environment to the local cluster. This allowed access to a broader spectrum of hardware and provided additional resources to support the compute load.

The per process data is collected by an external monitor that is called by the modified pipeline script, as represented in Figure 1. This external monitor uses ps and strace to collect data at fixed time intervals while keeping track of current CASA and pipeline tasks.

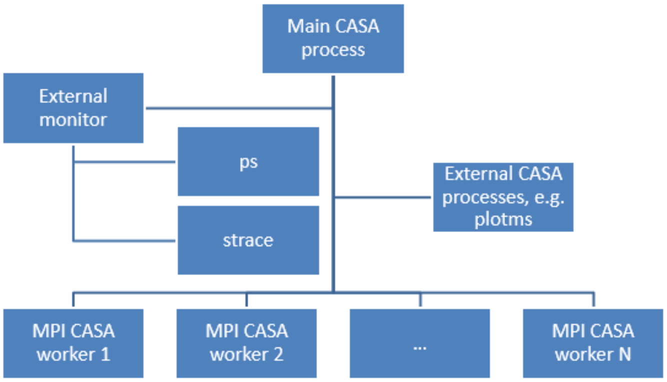

Figure 1: Representation of the process hierarchy on parallel pipeline runs.

The external monitor, represented in Figure 1, synchronously communicates with the main CASA process. Blocking on MPI subprocesses can impact reported CASA and pipeline task run times. To ensure the accuracy of reported run times, we record the monitor contribution to each task's run time so that it can be corrected during analysis.

3 DATA SELECTION AND PARAMETER SPACE COVERAGE

Based on the conclusions in the Phase 1 report and given the limited computing resources, we chose four datasets which exhibited the broadest range of input data size, memory footprint and run times in the imaging pipeline. To reduce the run time coverage gap shown in Figure 2 from the Phase 1 report, we reviewed four additional datasets from which one was selected.

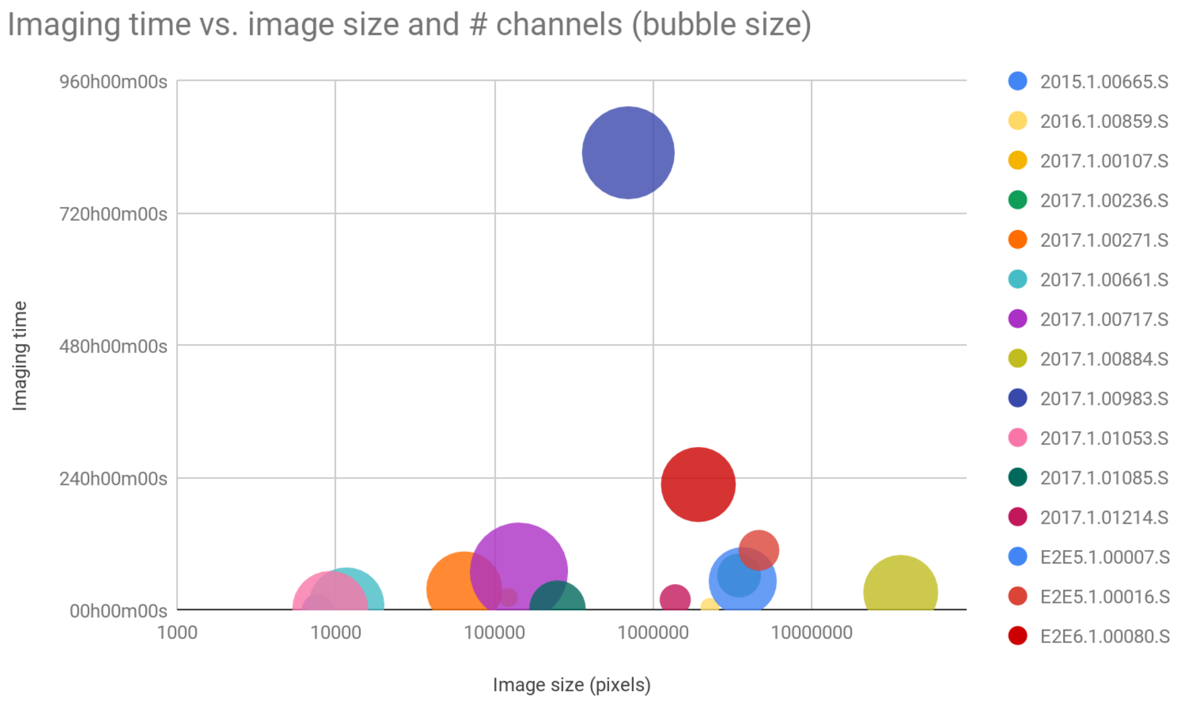

Figure 2: Serial imaging run times vs. image size and number of channels. Note the large gap in timing between ~ 240 and 800 hours. The red circle near 240 hours run time was selected to reduce this timing gap, corresponding to dataset E2E6.1.00080.S.

The following datasets were selected:

- 2017.1.00717.S
- 2017.1.00884.S
- 2017.1.00983.S
- 2017.1.01214.S
- E2E6.1.00080.S

4 METHODS AND MEASUREMENTS

All tests were executed on the AOC cluster and Amazon Web Services (AWS). Initially, tests were performed on the AOC cluster to establish a baseline to which the AWS tests could be compared. This allowed us to detect and account for potential systematic effects within AWS.

The following tests were performed on the AOC cluster:

- Serial benchmarks for all datasets
- Parallelization breadth (number of MPI processes)
- Storage type
- Concurrency

Nodes on the AOC cluster were selected as follows, according to test requirements and properties.

- nmpost001-050 for parallelization breadth (to be consistent with phase 1) and storage
- nmpost051-060 for concurrency (and also parallelization breadth)

The following tests were performed on AWS:

- Parallelization breadth (number of MPI processes)
- Memory limit
- Timing vs CPU type
- Number of OpenMP threads

The tests were executed with a script that takes a locally submitted PPR based pipeline and automates all the steps required to run on AWS (e.g. start an appropriate instance and image, copy data, run the pipeline, copy data back). One or more AWS instances were selected for each specific test, matching test requirements and instance properties, which are presented in Section 5.

A total of 122 pipeline runs were considered in this work. From the raw profiling data, an auxiliary database was built, consisting of detailed timing data for the complete pipeline, as well as the imaging tasks hif_findcont() and hif_makeimages() (total and tclean() only run times). The lines in that database are identified by project name, number of MPI processes, number of OpenMP threads, memory limit, storage type and CPU, so that data can be queried and extracted for analysis of each individual test case.

5 RESULTS

The profiling and benchmark results are presented in the subsections below, broken out by type of test: parallelization breadth, storage type, memory limit, concurrency, OpenMP and timing vs. CPU. The subsections are independent, each consisting of a description of the test, expected results, data analysis, results and conclusions.

5.1 Parallelization breadth (MPI)

To demonstrate the effects of parallelization, run time tests were performed on both calibration and imaging pipelines. The full range of parallelization breadths (2-, 4-, 8-, 16-way) of all five datasets was performed on the AOC cluster, while tests with 8- and 16-way parallelization of all datasets and an additional 24-way parallelization of the 2017.1.00983.S dataset were performed on AWS with the imaging pipeline. We also ran the calibration pipeline serially and with 8-way parallelization for all five datasets on

AWS. We used instance type m5.12xlarge (48 vCPUs, Intel Xeon Platinum 8175, and 192 GB RAM) to run all the AWS tests reported in this section.

Tests of the calibration pipeline were run only on AWS. Since the pipeline is not set to create multi-MS, the only pipeline task expected to run faster with higher parallelization is hif_makeimages() which calls tclean(), which is not a dominant contributor to run times in the calibration pipeline (see Phase 1 report). For each dataset, there was one serial run and one with 8 MPI processes. We considered the run times of the serial runs as baselines, and computed relative run times for each pipeline task on the 8-way parallelization run with respect to the baseline serial run. The results are shown in Figure 3.

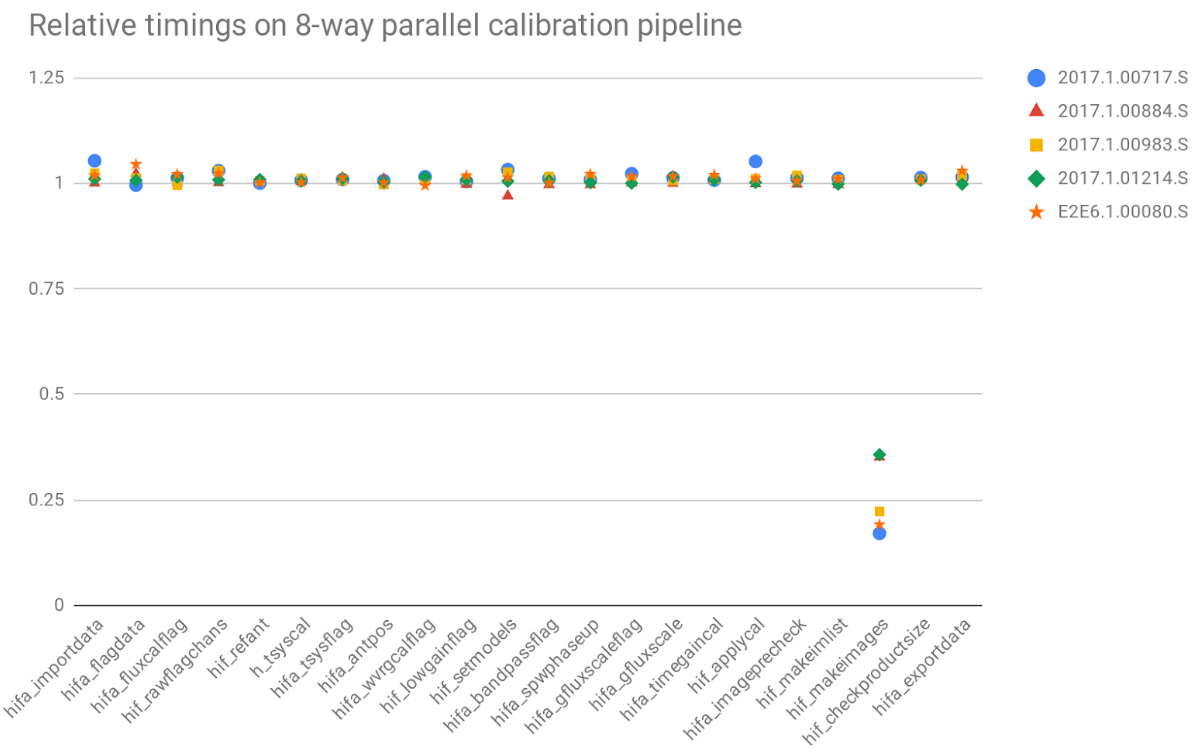

Figure 3: Relative timings for calibration pipeline runs with 8-way parallelization. Relative timings are computed with respect to the run time of the serial runs for each dataset.

The plot in Figure 3 shows that, for the calibration pipeline, run time only differs significantly within the pipeline task hif_makeimages() for serial and 8-way parallelization. While the parallel executions demonstrated a 65% to 83% decrease in run time within hif_makeimages(), the full pipeline run time was only reduced by 1% to 15%.

These results confirm that the run time of the calibration pipeline is only marginally reduced by increasing parallelization. Since it is only while imaging the calibrators that the calibration run time is reduced with increasing parallelization breadth, at this point we turn our attention to the imaging pipeline and focus on it for the remainder of this document.

The results for tests of the imaging pipeline with 2-, 4-, 8- and 16-way parallelization on AOC cluster and 8-, 16- and 24-way parallelization on AWS are summarized in Table 1.

Table 1: Run times of the imaging pipeline with all datasets and all parallelization breadth test cases

Dataset		2017.1.00717.S		2017.1.00884.S		2017.1.00983.S		2017.1.01214.S		E2E6.1.00080.S	
CPU		E5-2670	m5.12xlarge	E5-2670	m5.12xlarge	E5-2670	m5.12xlarge	E5-2670	m5.12xlarge	E5-2670	m5.12xlarge
Number of MPI processes	1	69h44m55s	35h36m19s	32h05m30s	14h58m33s	829h38m52s		18h41m15s	06h53m20s	304h24m49s	110h53m05s
	2	64h42m39s		17h53m00s		397h19m58s		12h20m53s		138h17m48s	92h01m43s
	4	47h17m52s		13h37m18s		229h03m40s		10h03m31s		86h24m04s	56h34m59s
	8	30h18m16s	19h54m54s	14h46m49s	07h01m10s	149h43m54s	100h17m48s	08h27m49s	05h09m30s	57h10m05s	37h04m39s
	16	24h50m40s	16h29m53s	14h06m46s	06h58m50s	110h22m54s	74h05m00s	08h39m06s	05h19m57s	45h19m57s	26h56m13s
	24						67h27m08s				

The run times are also shown in Figure 4, computed relative to the serial run time, to allow comparison of the performance between datasets with different run times.

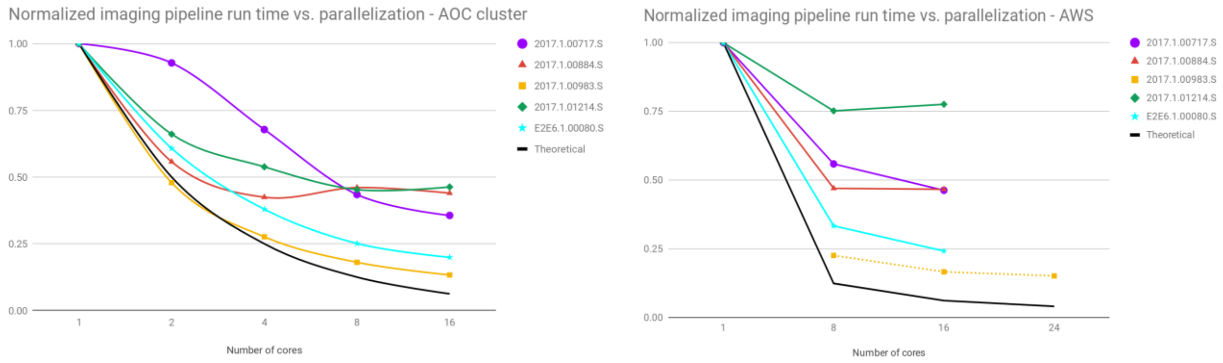

Figure 4: Relative run times (computed w. r. t. serial run times) as a function of parallelization breadth on AOC cluster (left) and AWS (right). The black solid line is the theoretical limit - points below that line are attributed to artificially slow serial runs. The yellow dashed line is based on a serial run time estimate and was added to the plot to illustrate the run time trend for that dataset.

The overall trend in Figure 4 is in agreement with the expectation that run times should monotonically decrease with increasing parallelization breadth, at the theoretical limit of 1/X with respect to the serial run time, where X is the number of parallel processes.

Visible on the AOC plot (left in Figure 4) is a data point slightly below the theoretical limit. Since it is not possible that the run time of a job with 2 processes is reduced by more than 50% with respect to the serial run, we consider that this is an artifact caused by the serial run being artificially slow. In this particular case, the serial run time (Table 1) is slightly over one month, which increases the likelihood of that run being affected by service instabilities. Two other attempts to rerun that serial case failed due to such instabilities.

Also present in Figure 4 are unexpected data points where the time increases with increasing parallelization breadth. These were selected for an in depth investigation of hif_makeimages() and ultimately tclean(), as the latter is currently the only task configured to use MPI in the imaging pipeline. This investigation is presented and discussed in detail in Section 6.

5.1.1 *Memory footprint scaling with parallelization breadth*

An important aspect of imaging parallelization using MPI is how memory footprint scales with parallelization breadth. Since one full size image is created for each MPI process in parallel imaging, the memory footprint is expected to increase linearly with the number of MPI processes, except in the cube imaging case where the use of chanchunks can mitigate the effect.

This scaling can be demonstrated indirectly by analyzing the effect of parallelization breadth on the aggregate memory footprint from tclean() and the number of chanchunks. As seen in Figure 5, our measurements of the memory footprint of tclean() do not appear to increase linearly with parallelization, rather staying roughly close to constant over the range, except for the smallest memory footprint in the sample, that shows an increase close to linear.

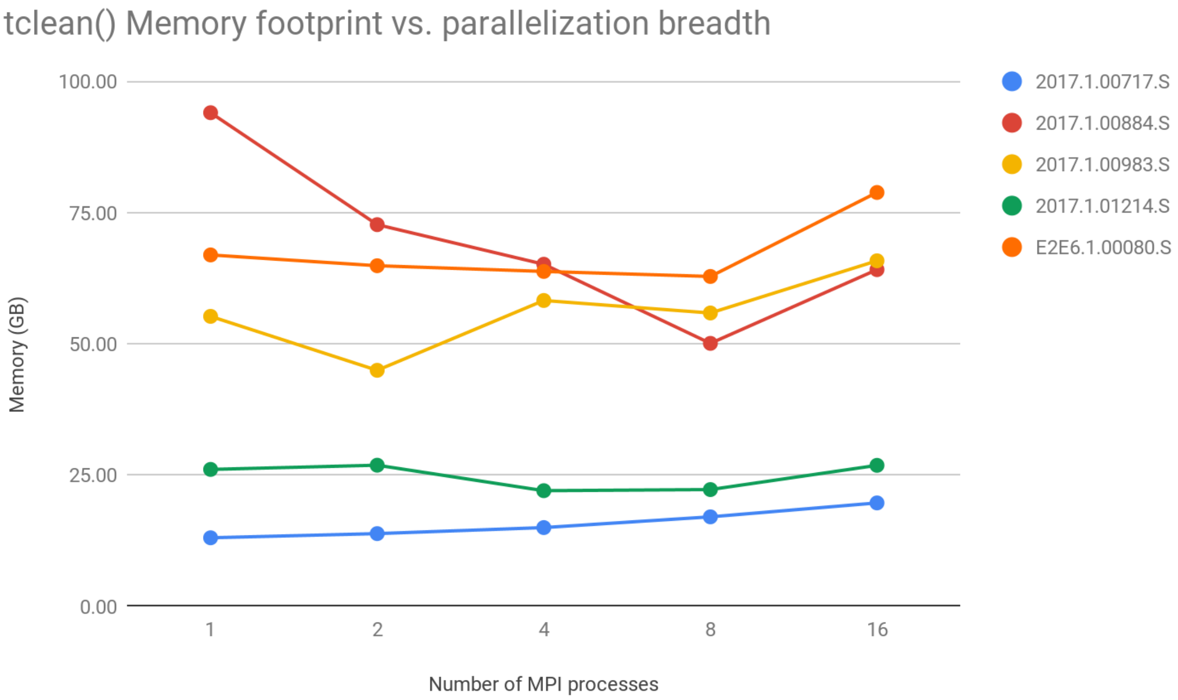

Figure 5: Memory footprint of tclean() as a function of parallelization breadth. The measurements were obtained from pipeline runs in which the chanchunks mechanism was enabled.

Since the memory footprint stays roughly constant, we analyze the number of chanchunks as a function of parallelization breadth, as shown in Figure 6.

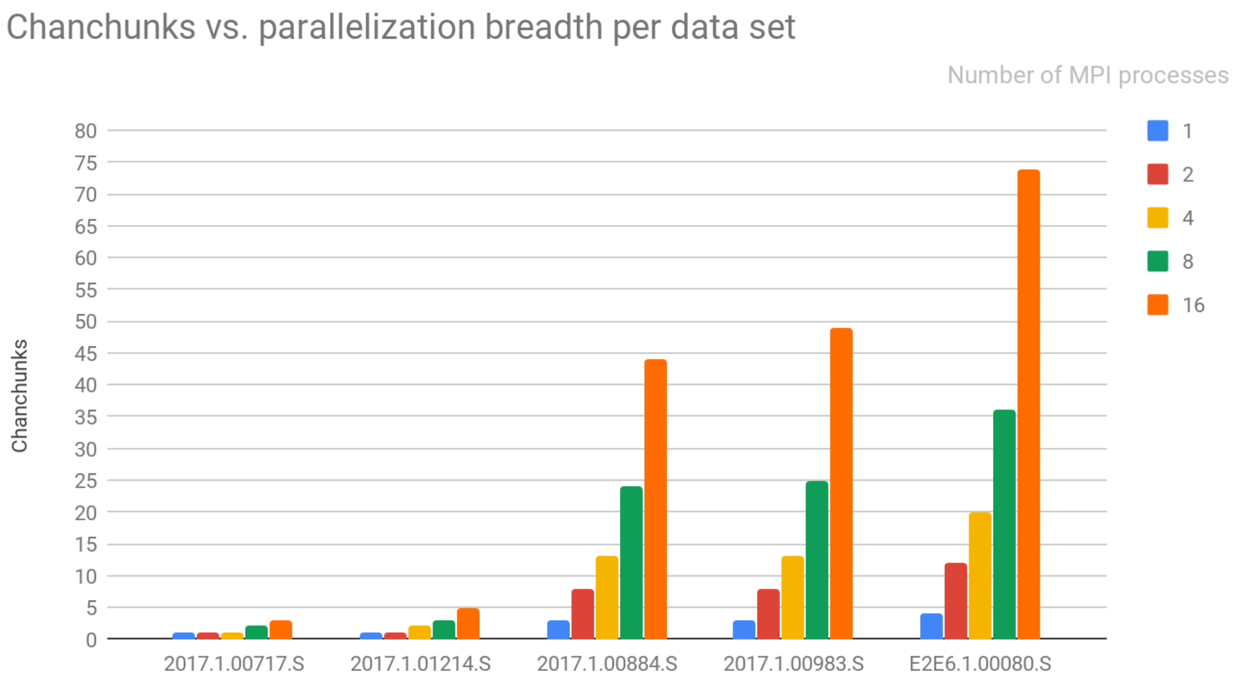

Figure 6: Chanchunks as a function of parallelization breadth, grouped by dataset.

Analysis of the plots in Figures 5 and 6 indicate that the number of chanchunks show an approximate linear scaling with parallelization breadth, while the memory footprint remains nearly constant. This is consistent with a memory footprint scaling linearly with parallelization breadth if chanchunks was not enabled. This analysis also suggests that the memory footprints are in good agreement with what CASA estimates before imaging, as this estimate is the basis for determining the number of chanchunks to use.

5.2 Storage type

The imaging pipeline was run against local storage on a RAID0 array and an NVMe device, to investigate the effect of different I/O data rates on pipeline run time. The NVMe device is expected to be faster in all test cases, however the precise improvement depends on the I/O access patterns for each dataset and task.

The results for tests of different storage types are shown in Figure 7. The tests were executed with 16-way parallelization as this is the most demanding case in terms of I/O. The NVMe device used for this series of tests was relatively small (1.5 TB), so only 3 out of 5 datasets could be imaged with this device. Similarly, the largest dataset (2017.1.00983.S) was also too big for the RAID0 array at run time, so we could not run imaging for that dataset with any storage type other than Lustre.

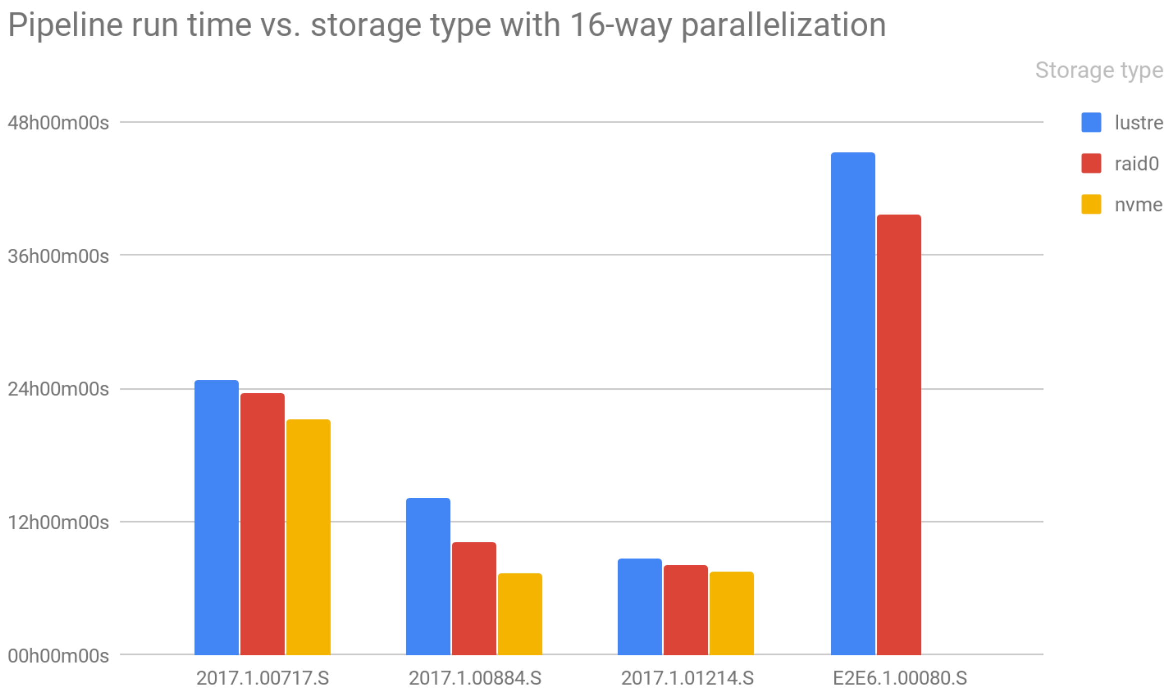

Figure 7: Pipeline run times for different storage types for all datasets with 16-way parallelization. The missing NVMe data points are because that dataset requires more space than the capacity of the device (1.5 TB) at run time.

The data plotted in Figure 7 indicates that both the RAID0 array and NVMe device were faster than Lustre to run the imaging pipeline. However, the run times were only slightly reduced (~ 8% RAID0 vs. Lustre; ~ 13% NVMe vs. Lustre), except for one particular case with significantly shorter run times (2017.1.00884.S run times reduced by 28% RAID0 vs. Lustre and 48% NVMe vs. Lustre).

Our sample shows consistently faster runs with RAID0 and NVMe. We consider that further investigation would be required to understand the performance variations as a function of dataset. Further tests will require either additional representative small dataset sizes or larger storage devices.

5.3 Memory limit

We investigate the effect of memory contention on the run times of the imaging pipeline by constraining the memory that CASA is allowed to use. This is set using the parameter "system.resources.memory" in the ~/.casarc file, and for the tests reported here it was set to 64 and 128 GB. We also ran tests in which a memory limit was not set to let CASA figure out the memory available (should be most of the memory in the node as these are dedicated jobs). All the tests ran on AWS instance r5.8xlarge, with 32 vCPUs, Intel Xeon Platinum 8175, and 256 GB RAM. The results for these tests are shown in Figure 8.

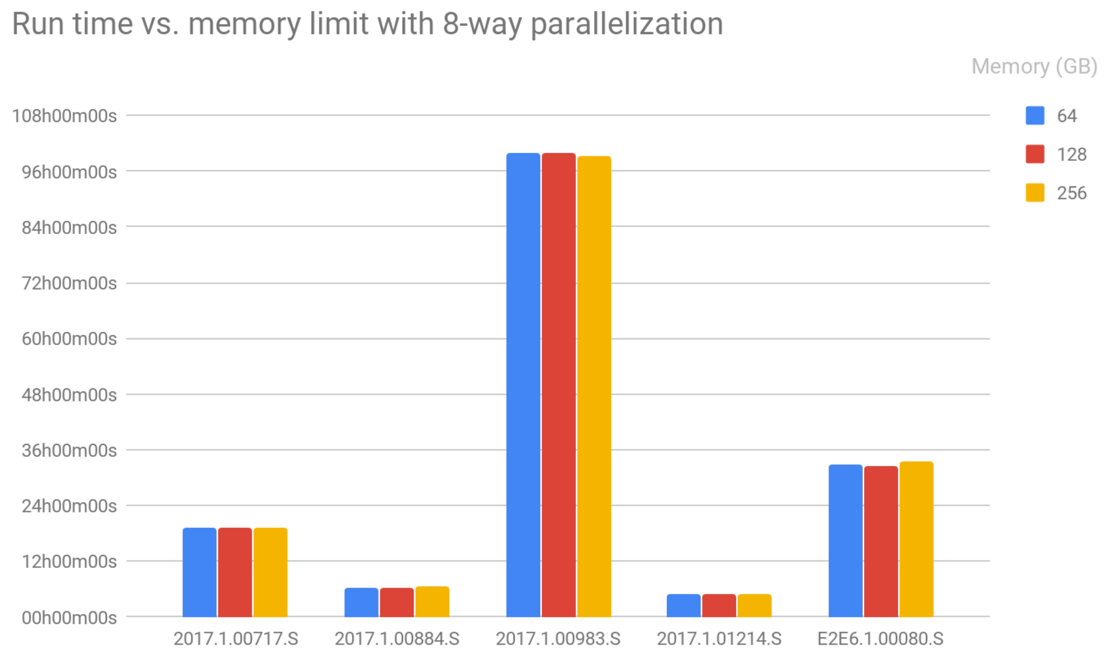

Figure 8: Pipeline run time as a function of memory limit for all datasets with 8-way parallelization.

We did not observe any significant changes in run time as a function of memory limit, with only one data point showing a 5% variation with respect to the mean, and all other data points within a ± 3% range. According to these results, a memory limit of 64 GB does not have significant impact on run time for the range of ALMA operations represented by our sample.

5.4 Concurrency

The aim of these tests is to investigate if and how throughput can be improved by running multiple jobs on a node. The following terms are used in this section:

- isolated - an isolated job is the only job running on a node
- concurrent - denotes a group of jobs that run simultaneously on a node
- siblings - the jobs that run concurrently are siblings of each other
- clones - siblings that are copies of the same pipeline and data

It is expected that siblings run slower due to contention when compared to isolated jobs, but that N concurrent jobs will complete faster than N isolated jobs run sequentially. For the tests, we utilized clones rather than siblings with different datasets. This scenario is likely to be more demanding in terms of resources than a real-world scenario where different jobs are running concurrently since clones are more likely to compete for the same resources at the same time. The concurrency test cases were designed to maximize the number of concurrent jobs on a node with 16 CPUs and 256 GB of memory:

- 2 jobs x 8 way parallelization & 125 GB / job
- 4 jobs x 4 way parallelization & 62 GB / job

To run these tests, we selected one dataset for which serial run-time is ~ 200 hours and memory footprint is ~ 100 GB. The results are shown in Figure 9.

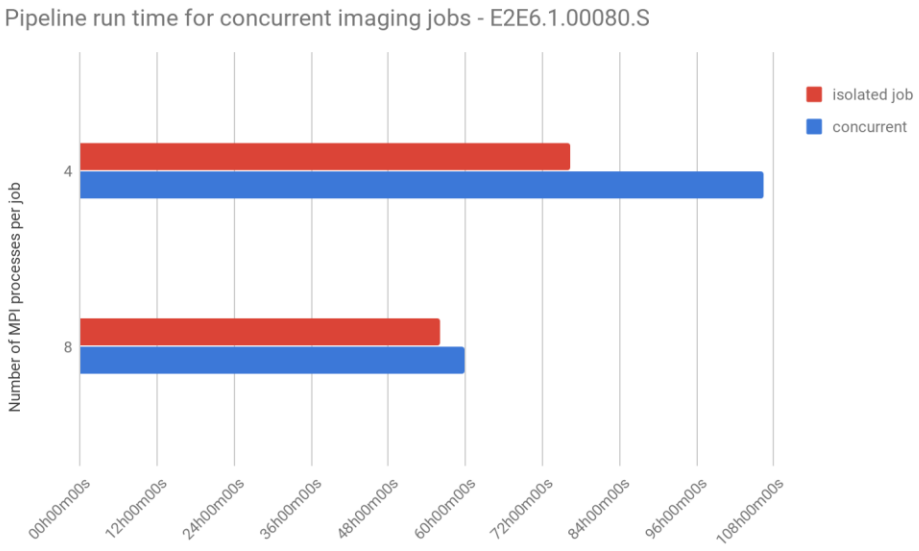

Figure 9: Imaging pipeline run times with dataset E2E6.1.00080.S for all concurrent jobs on a node is shown in blue. The run time of an isolated job on a node with the same number of MPI processes is shown in red.

The impact of contention on the timing of concurrent siblings versus the same dataset run in isolation is larger for 4-way parallelization. The jobs complete in 106 hours, while an isolated job runs with 4 cores on a dedicated node in 76 hours. For 8-way parallelization, the concurrent jobs complete in 59 hours, while the isolated job runs with 8 cores in 56 hours. That is 40% longer with the 4-way, and 7% longer with 8-way parallelization.

The penalty in performance with respect to isolated jobs is most likely due to memory swapping. Shown in Figure 10 is the swap memory as a function of progress for concurrent and isolated jobs. The plot shows heavy swapping in the concurrent case with 4 processes per job as compared to any other curve.

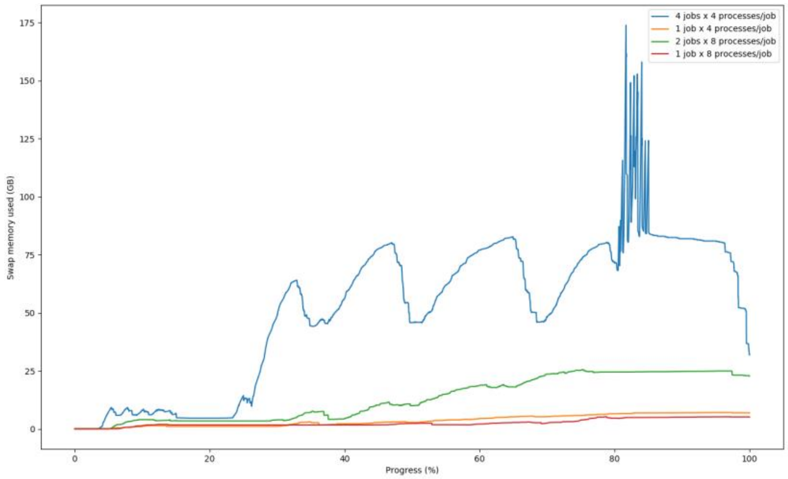

Figure 10: Swap memory as a function of progress for the concurrent 4 jobs with 4 processes per job (blue) and 2 jobs with 8 processes per job (green). The single job cases are shown for comparison with 4 processes per job (orange) and 8 processes per job (red).

There are two likely causes for this swapping:

1. Failure to limit CASA memory through both casarc and job submission (cgroups) resulting in CASA assuming there is more free memory for each job.
2. Static memory allocation within Python per pipeline execution.

More investigation in Phase 3 will be necessary to determine the precise cause.

Although concurrent jobs are slower when compared to isolated jobs with the same number of processes, they still run faster than the same number of jobs running in sequence in a dedicated node with 16-way parallelization, as shown in Figure 11.

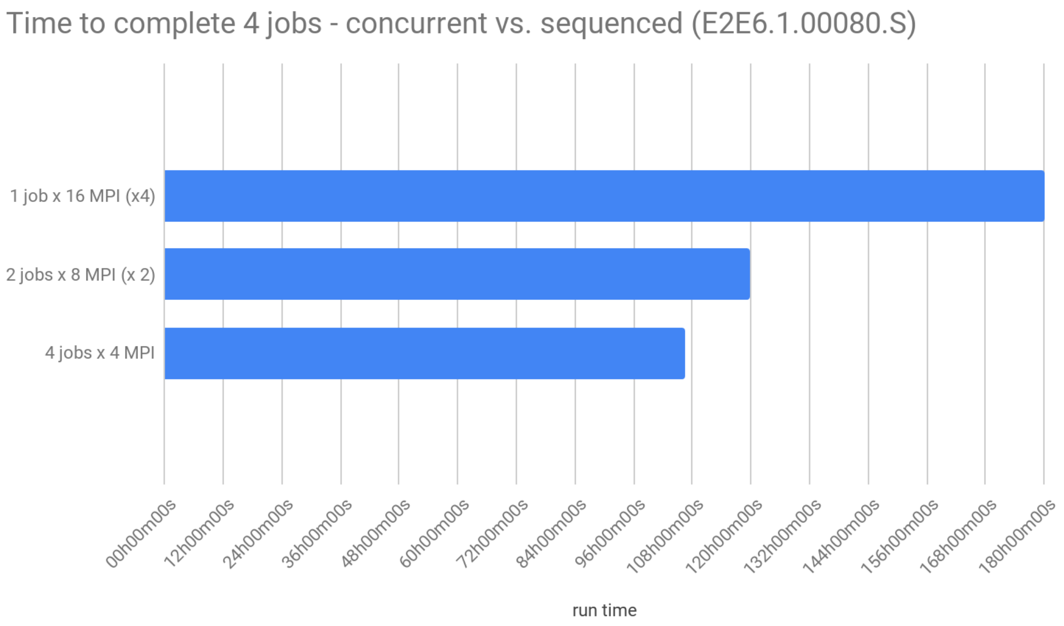

Figure 11: Time to complete 4 jobs with the Imaging pipeline and dataset E2E6.1.00080.S. The top bar shows 4 isolated jobs run sequentially with 16 MPI processes each. The bar in the middle shows the average time to run 2 clone jobs with 8 MPI processes each, repeated to complete 4 jobs. The bottom bar shows the average time to complete all 4 clone jobs with 4 MPI processes each.

In this case, 4 jobs taken 2 at a time with 8-way parallelization run in 35% less time than 4 jobs run sequentially with 16-way parallelization. Four jobs with 4-way parallelization run in 42% less time than the same 4 jobs run sequentially with 16-way parallelization. This shows that if there is sufficient memory to avoid swapping, then running more jobs on a node with lower parallelization breadth can potentially increase throughput, as well as cluster usage and efficiency.

5.5 OpenMP

As discussed in Section 5, MPI parallelization improves the performance of the imaging pipeline at the cost of increasing the memory footprint because it parallelizes in the visibility domain. On the other hand OpenMP is multi-threaded instead of multi-process, and in the case of CASA imaging parallelizes in the image domain. Due to computational redundancy, it is expected that OpenMP may be inferior to MPI in terms of performance, but has the advantage of keeping the memory footprint constant as the number of threads increase.

We selected dataset E2E6.1.00080.S to run OpenMP tests due to its typical run time and how it scales with parallelization breadth. The results are presented in Figure 12. All the tests reported in this section were run on AWS instance m5.12xlarge, with 48 vCPUs, Intel Xeon Platinum 8175, and 192 GB RAM, for consistency with the MPI parallel runs. All tests were setup so that the number of MPI parallel processes multiplied by the number of OpenMP threads was equal to the number of (physical) cores on the instance (16):

- 2 MPI workers x 8 OpenMP threads
- 4 MPI workers x 4 OpenMP threads
- 8 MPI workers x 2 OpenMP threads

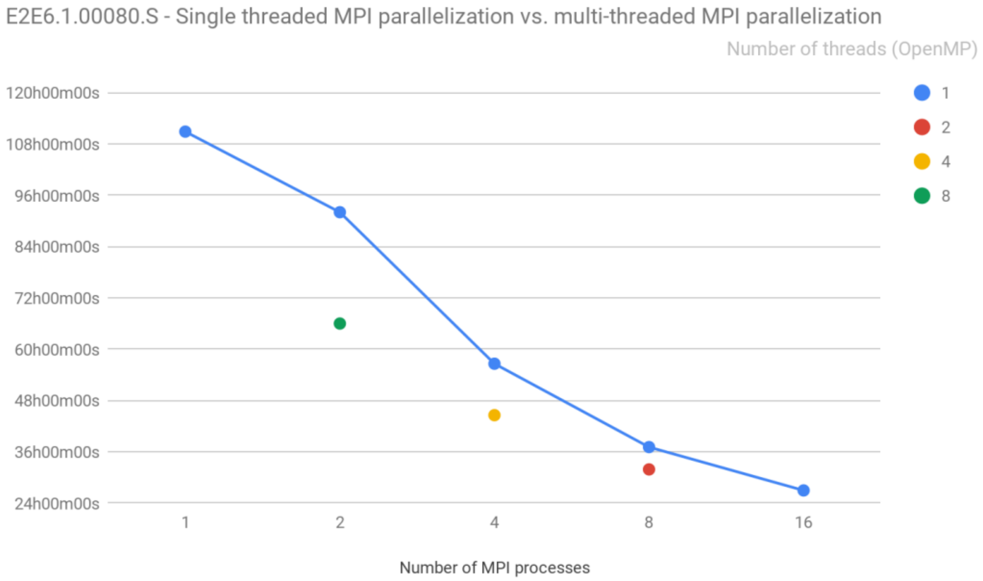

Figure 12: Pipeline run times for dataset E2E6.1.00080.S as a function of number of MPI processes and OpenMP threads.

The results in Figure 12 show the most significant reduction of run time as a function of increased MPI processes. Maximizing the number of threads for a given number of processes has a smaller effect than increasing number of processes.

The fastest run was with 16 single threaded MPI processes which suggests that MPI parallelization is more advantageous than OpenMP while there is sufficient memory to support additional processes.

5.6 Timing performance vs. CPU

To characterize the imaging pipeline performance with different CPU types, a series of tests was run on the following AOC cluster node class and AWS instances:

- nmpost001-nmpost050, Intel Xeon E5-2670, 16 CPUs, 192 GB RAM, Passmark 12114
- d2.4xlarge, Intel Xeon E5-2676 v3, 16 vCPUs, 122 GB RAM, Passmark: 17670
- r4.4xlarge, Intel Xeon E5-2686 v4, 16 vCPUs, 122 GB RAM, Passmark: 19801
- c4.8xlarge, Intel Xeon E5-2666 v3, 36 vCPUs, 60 GB RAM, Passmark: 24877
- m5.8xlarge, Intel Xeon Platinum 8175, 16 vCPUS, 128 GB RAM, Passmark: 28103

The Passmark is an industry standard benchmark that establishes a scale to quantify CPU performance differences.

To minimize expense of executing the full breadth of tests, datasets with shorter run times were selected to run these tests - 2017.1.00717.S, 2017.1.00884.S and 2017.1.01214.S. We ran the imaging pipeline for these three datasets with the four different CPUs on AWS (2017.1.00884.S only completed on m5.8xlarge due to its memory footprint). Along with the baseline runs on the AOC cluster, our total sample consists of 12 data points. The tests were run serially on dedicated nodes, with all the memory available. The run times resulting from these tests are shown in Figure 13.

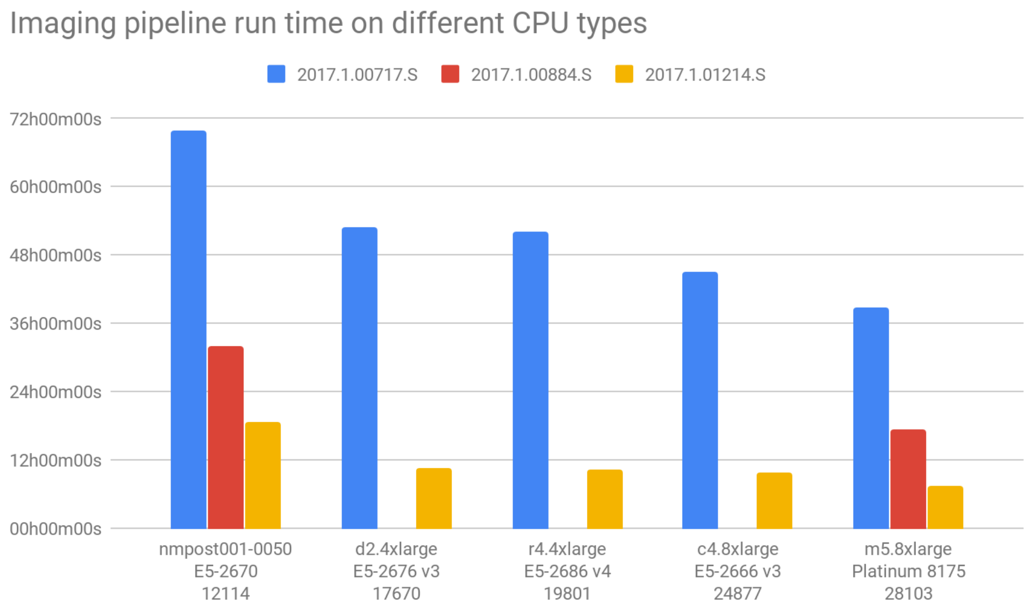

Figure 13: Imaging pipeline run times on different CPU types. The missing times on dataset 2017.1.00884.S are runs that did not complete due to insufficient memory. The labels on the horizontal axis contain the node/instance type, CPU type and Passmark, in this order.

The plot in Figure 13 shows that pipeline run times decrease with newer generation CPUs, consistent with their higher Passmark score. The Passmark benchmark is a measure of CPU performance, as such it can be a good proxy for CPU dominated tasks such as tclean(). Passmark's predictive ability is less for I/O dominated tasks. When considering new hardware, the Passmark score can help when comparing processors, but is not necessarily predictive of pipeline run times.

6 INVESTIGATION OF RUN TIMES INCREASING WITH PARALLELIZATION

The imaging pipeline run times were shown in Section 5 to decrease as expected with parallelization breadth (number of MPI processes) in the majority of our sample. While presenting and discussing those results, it was pointed out that there were some cases where the run time unexpectedly increased with parallelization breadth. We now turn our attention to these cases, going deeper on the analysis of task timings.

We start investigating run times increasing with parallelization breadth by computing the individual run times of the largest contributors to total pipeline run time: hif_findcont() and hif_makeimages(), which are the tasks that call tclean(). Again, we compute the relative timings with respect to the serial case to produce the plots shown in Figure 14.

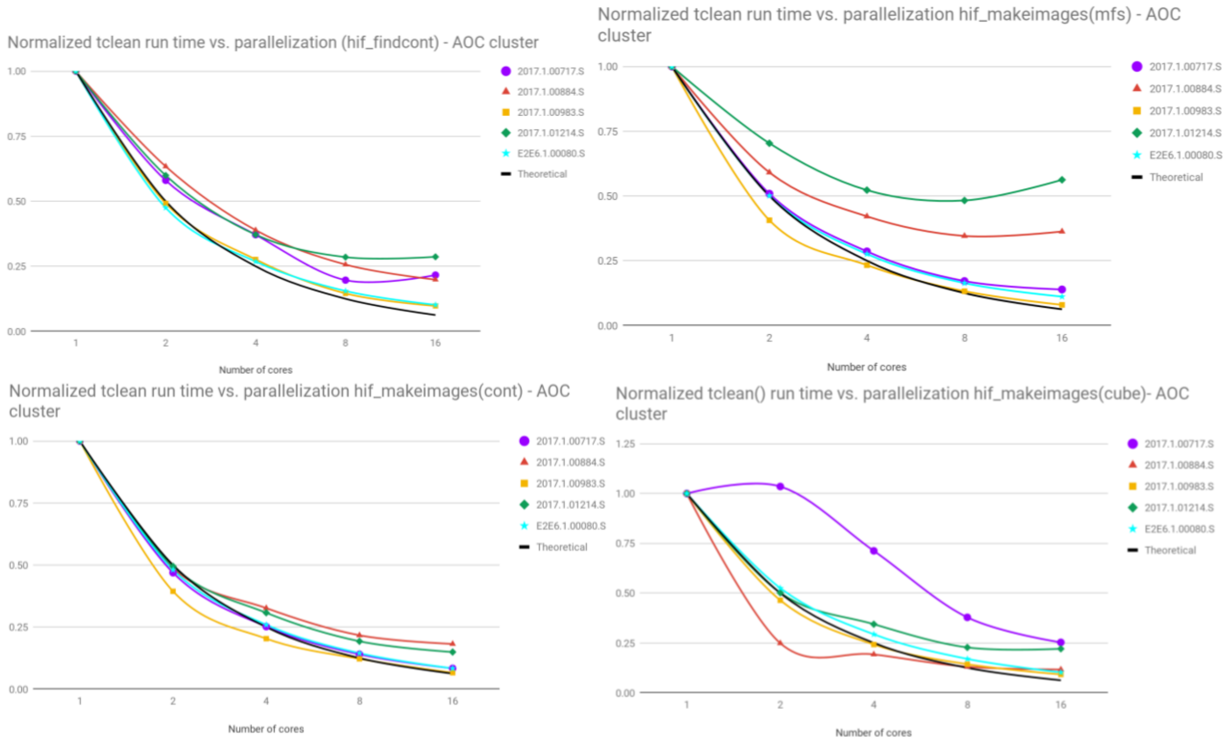

Figure 14: Relative aggregate run times of tclean() calls on individual pipeline tasks hif_findcont() (top left) and hif_makeimages() with specmode set to 'mfs' (top right), 'cont' (bottom left) and 'cube' (bottom right).

Shown on Figure 14 are the aggregate run times of tclean() calls on each pipeline task. Analysis of these plots in conjunction with the Figure 4 plots indicates that the run times increasing with parallelization come from the CASA task layer, not the pipeline task layer. It also shows that continuum imaging scales as expected with parallelization breadth when all spectral windows are combined (identified in the pipeline as specmode='cont'; hif_makeimages() second call), while run time increases with parallelization breadth on individual spectral windows (identified as specmode='mfs'; hif_makeimages() first call).

From this analysis, we identify the tclean() calls on the first and third calls of hif_makeimages() (respectively specmode='mfs' and 'cube') for further investigation, as these are the pipeline tasks showing more run times increasing with parallelization breadth. In the next section, we present and discuss the results for an expanded investigation of tclean() cycle timings.

6.1 Expanded investigation of tclean()

In the previous section we identified pipeline tasks and datasets which showed unexpected increased tclean() timings for specific parallelization breadths. The datasets 2017.1.00884.S and 2017.1.01214.S show run times of continuum imaging per spectral window ('mfs') decreasing as expected up to 8 parallel processes, but increased in time with 16 processes. On the other hand, dataset 2017.1.00717.S shows an increase in run time of cube imaging with 2 parallel processes, turning to faster runs with increasing parallelization breadth for 4-, 8- and 16-way parallelization breadth.

In order to understand these unexpected variations in imaging run time, we computed major and minor cycle times, as well as number of major cycles, for each execution of tclean(). We compare cycle timings for all datasets as a function of parallelization breadth to determine what are the dominant sections in run time, and how that changes with parallelization breadth to result in the timing increases that we see on 'mfs' for datasets 2017.1.00884.S and 2017.1.01214.S. The cycle times are shown in Table 2, with lines grouped by dataset, and columns for values of parallelization breadth.

Table 2: tclean() cycle times of continuum imaging vs. parallelization breadth

		2 cores	4 cores	8 cores	16 cores
2017.1.00717.S	start up time	709	664	772	1060
	time on major cycles	8360	4473	2295	1432
	time on minor cycles	21	25	24	37
	scatter/update model time	16	15	32	52
	TOTAL	9106	5177	3123	2581
2017.1.00884.S	start up time	268	288	359	522
	time on major cycles	992	624	398	284
	time on minor cycles	18	19	21	21
	scatter/update model time	5	7	14	27
	TOTAL	1283	938	792	854
2017.1.00983.S	start up time	2598	2741	3084	3556
	time on major cycles	95753	54381	29545	16292
	time on minor cycles	86	93	85	102
	scatter/update model time	20	54	74	128
	TOTAL	98457	57269	32788	20078
2017.1.01214.S	start up time	633	657	813	1151
	time on major cycles	1089	644	421	323
	time on minor cycles	8	8	12	14
	scatter/update model time	9	18	29	43
	TOTAL	1739	1327	1275	1531
E2E6.1.00080.S	start up time	796	860	986	1257
	time on major cycles	5107	2643	1390	741
	time on minor cycles	0	0	0	0
	scatter/update model time	8	5	13	20
	TOTAL	5911	3508	2389	2018

The two dominant terms in data shown in Table 2 are 'start up time' and 'time on major cycles'. The 'start up time' is the entire time interval before the first major cycle starts. The 'time on major cycles', on the other hand, is actually the gridding phase of the major cycle, while ‘scatter/update model time’ refers to the degridding. For these two dominant terms, we see that ‘start up time’ increases with parallelization breadth, while ‘time on major cycle’ decreases. The latter is expected as we know from the implementation of tclean() that gridding is the stage that benefits the most with parallelization. By analyzing the total tclean() time we notice that as the ‘start up time’ becomes the dominant factor, time begins to increase with parallelization breadth, and that affects the two datasets with shortest imaging times in our sample. For the same reason, the imaging cases with longer run times are the ones that show better overall scaling with parallelization breadth.

The increase in tclean()’s ‘start up time’ with parallelization breadth as shown in Table 2 may warrant further investigation in order to increase the effectiveness and scalability of parallelization.

Next we analyze the anomalies in run time with 2-way parallelization observed in cube imaging for the datasets 2017.1.00717.S and 2017.1.00884.S. Figure 15 is a plot of tclean() run times for each spectral window with 2017.1.00717.S, showing that run times increase from serial to 2-way parallelization on all but two spectral windows, namely spw 45 and 47, and the latter shows an increase from 2- to 4-way parallelization.

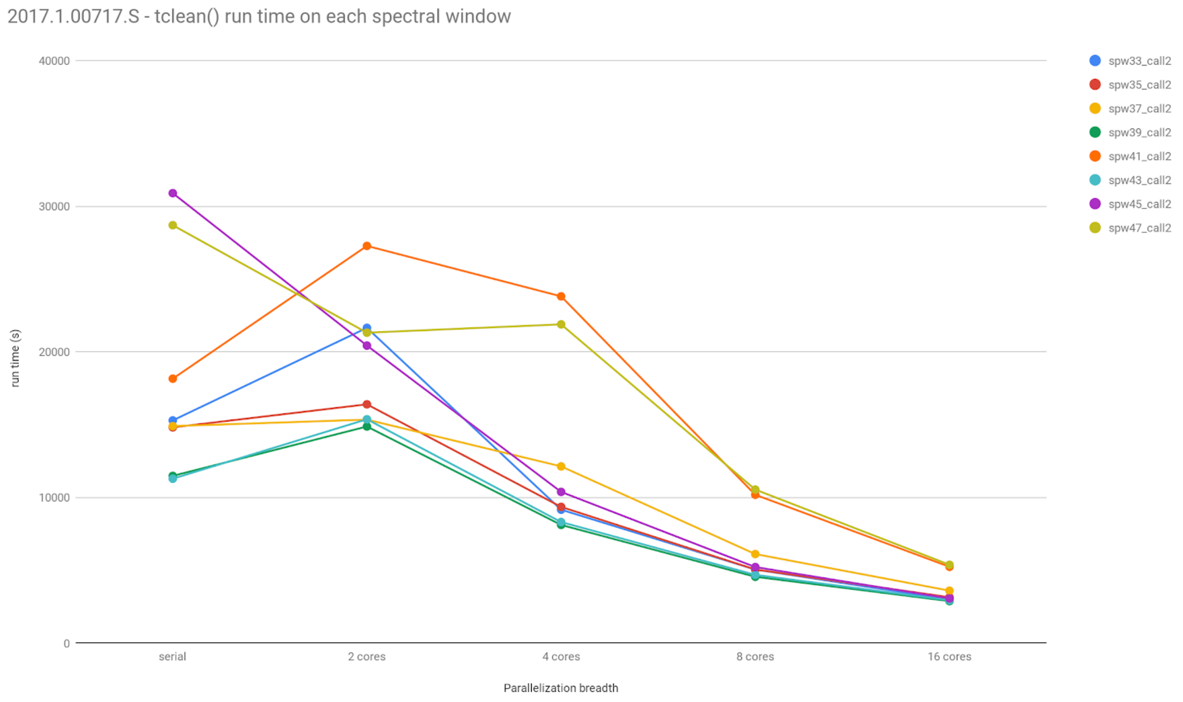

Figure 15: tclean() cube imaging run time on each spectral window as a function of parallelization breadth for dataset 2017.1.00717.S.

Again, we compute cycle times and number of major cycles for all tclean() calls, as shown in Figure 16. The average major cycle time decreases with parallelization breadth as expected. On the other hand, we found large variations in the number of major cycles on at least two spectral windows, while it was expected to be constant (or nearly constant) with parallelization breadth.

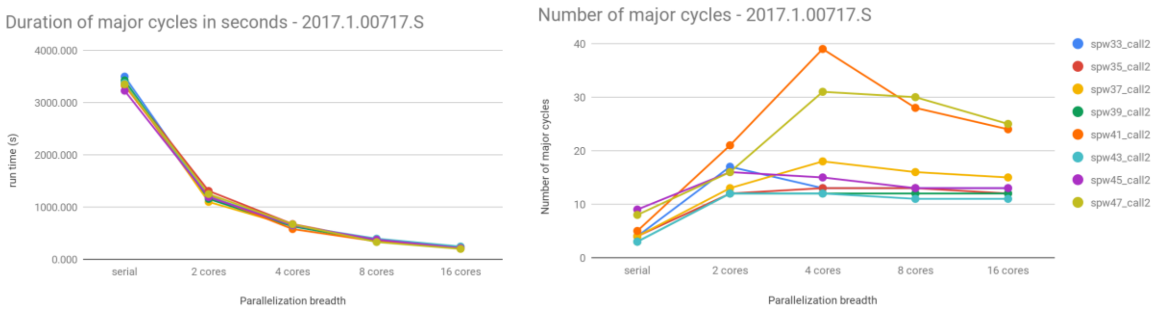

Figure 16: Average tclean() major cycle times (left) and number of major cycles (right) on each spectral window for dataset 2017.1.00717.S.

Our analysis indicates that the different number of major cycles in cube imaging is due to the data being split across the different parallel processes and CLEANed independently. As each process gets a subset of the channels in a spectral window, and executes major and minor cycles independent of the other processes, CLEAN convergence to a common threshold can be different for each subset of channels, and variations will also occur when the number of channels in each subset changes with parallelization breadth. This may warrant further investigation to ensure that science products are not impacted by changes in parallelization breadth.

When applying the same analysis to 2017.1.00884.S, it was found that the cubes were only produced in the serial case. All MPI parallel runs with this dataset reached 'zero mask' stopping criterion after making the dirty beam. This clearly explains the otherwise impossible decrease in run time, and also indicates that the behavior of this dataset against MPI runs requires further investigation.

An important remark is that the duration of each individual major cycle decreases as a function of increased parallelization breadth, even in the cases shown in this section where the total imaging time may have increased.

7 CONCLUSIONS

In this second report on benchmarking and profiling the ALMA pipeline, we present a more in depth analysis of the pipeline performance, indicated as next steps in the Phase 1 report. Tests of the ALMA pipeline were conducted with various hardware configurations, resource constraints (e.g. memory) and parallelization strategy to show how run times, memory footprints and I/O patterns vary in those different test cases. A summary of the main findings reported in this document is shown next.

Because the calibration pipeline does not trigger MMS generation, only the tclean() portion of the hif_makeimages() pipeline task benefits from parallelization. Therefore the reduction of calibration pipeline run times due to MPI parallelization is very small (15% or less in our sample). On the other hand, we found the imaging pipeline run times for the majority of datasets decreased nearly linearly with increasing number of MPI processes. Realized performance gains from parallelization are negatively impacted by the fact that tclean() initialization times increase with parallelization breadth. In some simple imaging cases this increase can be greater than the corresponding decrease in actual imaging time, resulting in a total increase in tclean() time. In addition, as described in Section 6, the number of major cycles in cube imaging can vary as a function of number of MPI processes, which drives tclean() run times. If desired, we will further investigate this behavior in phase 3.

Our tests consistently showed shorter run times of the imaging pipeline with RAID0 and NVMe storage types, with some variation in improvement as a function of dataset. The improvements observed in run times (typically less than 15%) with those storage devices, in particular the NVMe, are too small to justify the cost (~30% per node) of changing from the current storage model. Note that Terabyte class datasets, which may exhibit different behavior, could not be tested due to the size of the available storage devices. Demonstrating the effect of these storage types for large datasets will require new, larger devices.

The tests of the imaging pipeline with 8-way parallelization with memory constrained to 16 GB and 8 GB per process via casarc configuration (to mimic available physical memory) did not show any appreciable difference in run time versus the 32 GB per process non constrained case. Further tests would be needed to investigate behavior on environments below 8 GB per process.

We performed job concurrency tests with the imaging pipeline and a representative dataset, showing that if there is sufficient memory to avoid swapping, running more jobs on a node with lower parallelization breadth increases throughput, as well as cluster usage and efficiency. The 4-way concurrency test demonstrated periods of swapping but was still the most efficient case. While the 4-way concurrency test appears to be the most efficient mode, we recommend isolated or 2-way concurrency with 8-way parallelization until swapping behavior is better understood. We will investigate the cause of this swapping as part of a re-examination of memory usage in current versions of CASA during Phase 3.

We showed that MPI parallelization has a performance advantage over OpenMP when there is enough memory available to support more MPI processes. If system memory is exhausted and there are unused cores then OpenMP can be advantageous (e.g. 8 MPI processes of 2 threads each on a 16 core machine).

Reduction in imaging pipeline run times was demonstrated by running against different AWS instances with various newer generations of CPUs. When compared against Passmark, shorter run times were consistent with higher Passmark scores, suggesting that higher Passmark scores are likely to be indicative of faster runs.

8 NEXT STEPS

A third and final phase of this work is planned for the upcoming months, in which we intend to:

- Rerun a subset of the tests with a newer version of CASA (most likely the pre-release that is current when new tests begin)
- Investigate the following points in this report:
    - Different number of major cycles for different parallelization breadth
    - Low memory behavior including swap memory usage for concurrent jobs
    - Performance of different storage types with larger datasets
- Enable access to CPU event counters via the PAPI (Performance Application Programming Interface - https://icl.utk.edu/papi/overview/index.html), to allow relating software performance to processor events
- Investigate locking overhead in parallel runs of self calibration when updating the model column (CAS-12612)
- As time allows, perform tests identified by operations and CASA developers.