Baseline HPG runtime performance for imaging

ngVLA Computing Memo No. 7

S. Bhatnagar, F. Madsen and J. Robnett

May, 2022

Abstract

Characterization of the runtime efficiency of the High Performance Gridder (HPG) for realistic use-cases for imaging requires that the HPG be interfaced with the rest of the imaging framework code executed on a CPU host. For this, modifications in the imaging framework are required to harvest the runtime performance improvements offered by the use of the HPG. A stand-alone application and a modified imaging framework were therefore developed, and used for the measurements and characterization of the runtime performance. A number of tests were done for specific use-cases that benefit from the use of the HPG (namely A-, W- and AW-Projection applied to wide-band mosaic imaging with corrections for antenna pointing offsets) to establish a baseline runtime performance for realistic applications.

This memo describes the design and implementation of the stand-alone application `roadrunner` and the re-factored imaging framework (`HTCSynthesisImager`) developed by the Scientific Computing Group (SCG). Together these allowed replacing the CPU gridder with the HPG running on a GPU for degridding/gridding operations. The details of the tests, runtime measurements and the analysis of the runtime performance is also discussed. Comparison of the runtime performance with the upper limit on the performance expected from the HPG implementation verifies that the interfacing of HPG with the CASA imaging framework does not lead to any unreasonable loss in efficiency. Furthermore, we show that it is possible to harvest most of the efficiency offered by the HPG for real-life imaging applications, ultimately limited by the Amdahl's law (e.g. when HPG runtime is comparable to the runtime for serial operations in the imaging workflow).

1 Introduction

The High Performance Gridder (HPG) is an implementation of the gridding step in the imaging workflow for GPUs. It is implemented in the `Kokkos` framework which allows, potentially, compilation of the *same* code for GPU and CPU. See Pokorny (2021) for details of the implementation and the runtime performance characterization of the HPG code. The measurements for performance characterization were made with the data and the gridding convolution functions (CF) produced locally on the GPU, and a single CF of a particular support size used for each test. The measured performance reported therefore does not include runtime overheads when interfaced with the rest of the imaging framework in CASA, or any overheads due to the i/o between the CPU (where the imaging framework runs) and the GPU (where the gridder runs). Those measurements therefore correspond to the upper limit on the expected performance.

To characterize the HPG implementation for realistic imaging setup, the HPG needs to be interfaced with the imaging framework so that data read from a Measurement Set (MS) can be transported to the GPU for gridding, FFT applied to the gridded data at the end of the gridding cycle and the resulting image copied from the GPU to the CPU memory for use in the rest of the imaging workflow. This interfacing was done by replacing the gridder in the `AWProjectFT` framework with the HPG. However with the HPG, gridding for only one type of image can be done at a time (for efficiency reasons). Since the workflow in the imaging framework is optimized for the CPU, it couples the creation of different types of images in a

single pass through the MS. A standalone application and a modified CASA tool interface were therefore also developed to decouple the gridding for various images required for end-to-end imaging.

The design and implementation of the standalone application `roadrunner` and the modified CASA tool interface are briefly described in Sec. 2. Section 3 has the description of the runtime measurements and the analysis to characterize the performance of the HPG for a few use-cases of interest that benefit from the use of HPG, namely, W-, A- and AW-Projection applied to mosaic imaging with corrections for time- and antenna-dependent pointing offsets.

2 The `roadrunner` control flow

The `roadrunner` is a stand-alone application designed to make the various types of images required for end-to-end imaging, one at a time (see Fig. 1). Since predicted visibilities are a by-product of residual image computations, `roadrunner` also supports a mode for computing model visibilities and writing them to the MS. For gridding and de-gridding, CASA's `FTMachine` framework that drives the HPG via the `VisibilityResampler` framework is used. Some modifications in the workflow of the `FTMachine` framework are necessary primarily because the HPG runs in a memory address space different from the CPU address space where all of the rest of the `FTMachine` workflow is executed. These modifications are encapsulated in the `AWProjectWBFTHPG` C++ class which has the `AWProjectFT` class lineage (see Bhatnagar (2022) for the detailed design). This is constructed with the `AWVisResamplerHPG` plug-in object which is a descendant of the `VisibilityResampler` class. The workflow is managed within the main `roadrunner` code, using the services of the `AWProjectWBFTHPG` object for workflow management, which in-turn uses the `AWVisResamplerHPG` object to drive the HPG.

The other significant change in `roadrunner` is the use of multiple threads. Currently, one extra thread is used for pre-fetching and packing the Convolution Functions (CF) into the HPG data structures in parallel with gridding which is managed via the main thread (see Fig. 2). In the future (or in the production implementations), particularly for imaging larger data sets more CPU threads may be necessary for serial operations that run on the CPU for optimal performance overall. This, along with the aforementioned fact that the HPG runs in a separate memory address space are the primary reasons for the changes needed in the `FTMachine` and `VisibilityResampler` frameworks. While the design intention is to keep the HPG interface inside the `AWVisResamplerHPG` object, in order to manage the producer-consumer pattern of CPU threads and CUDA limitations w.r.t. multi-threaded CPU code, a few extra calls related to HPG usage pattern have spilled into the `roadrunner` code. Figure 1 shows the control flow of the `roadrunner` application. Function/method calls at each stage of the flow are light-weight and attempt to replicate the flow in CASA's imaging framework, albeit with reduced functionality.

The `UI()` function implements a user interface using an external library (parafeed[1]) that allows setting the various parameters via an embedded interactive shell, or via `parameter=value(s)` styled command-line interface (see the user manual[2] for details). `loadMS()` opens the MS using CASACore interface, applies data selection using the `casacore::MSSelection` module and constructs a reference to the selected MS. The `Setup VI2` stage sets up the data iterator (`VisibilityIterator2`). The `weightor()` function (implemented in `rWeightor.cc`) initializes and sets up the data iterators to apply the chosen weighting scheme. The `MakeEmptyImages()` function constructs the in-memory images for receiving the final image (single precision) and the double-precision complex grid(s) for receiving the gridded data. `hpg::initialize()` (and `hpg::finalize()` later) initializes the HPG library. `createAWPFTMachine()` function constructs the top-level `FTMachine` object instance based on user setting (of type `AWProjectWBFTHPG` or `AWProjectWBFT`), along with the plug-in objects it needs (instances of the `CFCache`, `VisibilityResampler`, and `AWConvolutionFunction` classes). The next step initializes this `FTMachine` instance. The `Data Iterations` step then iterates over all of the selected data and triggers gridding (`AWProjectWBFTHPG::put()`) or de-gridding (`AWProjectWBFTHPG::get()`), which in-turn drives the HPG via the `AWVisResamplerHPG` instance. Note that in contrast with `AWProjectWBFT::put()` which triggers only gridding, `AWProjectWBFTHPG::put()` also first triggers degridding if a model image is set.

Detailed implementation of the `Data Iterations` step is shown in Fig. 2. A separate thread is launched for loading, preparing and packing the CFs required for gridding (managed via the main thread). The execution in these threads need to be coordinated to ensure that the CFs being prepared in the `CFServer` thread for loading on the GPU are in sync with the data being gridded via the `main` thread. This coordination is managed using the classes `std::mutex` and `std::condition_variable` from STL via the `ThreadCoordinator` object.

[1] https://github.com/sanbee/parafeed

[2] https://github.com/sanbee/parafeed/blob/ead590819d673ec182f9f76555d70b791e4af694/UserDoc.md

Finally, the `finalizeToSky()` and `getImage()` methods apply the FFT on the final complex grid(s), copy the images from the HPG to the CPU memory, normalize the images before converting the image from polarization to Stokes basis, and saving the requested images to the disk before exiting the application.

2.1 `hphtclean` workflow

Figure 3 shows the control flow in the `hphtclean.sh` script (see Appendix A) developed by the Scientific Computing Group (SCG). The script uses the Python class called `HTCSynthesisImager`[3] (also developed by the SCG) which is a specialization of the CASA's `PySynthesisImager` class. The `PySynthesisImager` class in CASA drives the top-level task `tclean` and the `HTCSynthesisImager` class uses the same calls as used in `tclean`. It therefore exposes the exact same imaging framework code as used in production CASA, but which can now be driven from an independent Python instance or, as here, via a `bash` script. This allowed replacing the CASA imaging framework calls for making various images by `roadrunner` in the `hphtclean.sh` script while keeping the rest of the calls for image normalization, application of minor cycle algorithms and implementation of exit criterion same as in the `tclean` task. This also naturally enables decoupling the making of the various images (psf-, weight-, sumwt-, residual-image) as separate independent steps.

3 Runtime performance

This section documents the runtime performance of the HPG. As characterized by Pokorny (2021), the intrinsic speedup of the HPG implementation is in the $100-250$x range depending on the CF support size, under the following conditions:

- no overheads due to data i/o between the CPU and the GPU
- no runtime overheads of the imaging framework code (which is necessary for real-life application)
- using a single CF generated internally on the GPU per test

3.1 Baseline performance of HPG interfaced with CASA imaging framework

In order to characterize the baseline performance of the HPG code running on the GPU interfaced with the CASA imaging framework, we first measured the runtime for the calculation of just the *Dirty Image* which requires *only* gridding of the data (i.e. no de-gridding required). To further eliminate any overheads involved with the use of multiple CFs, this experiment was done with a single CF of support size of 31 pixels simulated on the CPU in the host memory. This *single* CF was then transferred to the GPU at the beginning of the test. A MS for a typical VLASS observation[4] was used as the test data. Data iterations were run on the CPU and the chunks of the visibility data per iteration (encapsulated in the `casa::VisBuffer` object) were transferred iteratively to the GPU and gridded using the host-GPU interface library (`libhpg`).

For reference, the intrinsic upper limit on the expected speedup of HPG is ~150x for the CF support size used for this test.

VLASS observing setup for on-the-fly (OTF) mosaic imaging requires correction for the resulting time-dependent antenna pointing offsets during imaging. For Epoch-1 data, these offsets are also antenna dependent. The test data used has 407 fields and 16 SPWs each with 64 frequency channels. For Stokes-I imaging this corresponds to $O(10^8)$ visibilities. For a single CF with a support size of 31 pixels, the rate of gridding was measured to be 4×10^6 vis/sec, which corresponds to a speed-up of $120-150$x compared to the rate of gridding with a single-threaded gridder on the CPU. This is consistent with the raw performance figures reported by Pokorny (2021). The test therefore verifies that for an *equivalent* setup there is no significant loss in runtime efficiency due to the CASA imaging framework.

3.2 Performance comparison for gridding: The W-only projection case

The memory footprint of the CFs required for AW-Projection for VLASS imaging is too large to fit in the available resources. A `CFCache` paging algorithm is therefore used to mitigate the memory footprint. On the CPU this loads and keeps in memory a slice of the CFCache for all W-terms and all polarizations corresponding to a single SPW at a time. Furthermore

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

[4] `VLASS2.1.sb38453816.eb38509426.59047.17567765046_split.ms`

with HPG this CF-slice is loaded on the GPU each time the SPW ID in the data iterations change. The default data access pattern in the CASA framework (iterations along the `FIELD` axis) adds significant i/o overhead due to frequent updates of the CF across the CPU-GPU bus. These overheads dominate the total run-time.

We therefore tested the HPG performance with a *single* CF-slice along the SPW axis from a standard VLASS CF-Cache loaded *once* and re-used for the data from all SPWs. This eliminated the i/o overheads of loading a new slice of `CFCache` for every change in the SPW ID, and thus allowed measurement of the runtime performance for a more realistic and a valid use-case, namely, W-only projection which corresponds to an imaging setup of `wprojplanes=32` and `wbawp=false` for `gridder=awproject`. The support size of the 32 CFs in units of the grid pixels used for this test were:

```
9 9 10 11 12 13 15 17 20 25 31 37 39 39 45 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50
```

Table 1 shows comparison between the measured run-time with the single-threaded gridder on the CPU from production CASA and the run-time with the gridder replaced by the HPG running on a single GPU using the `roadrunner` and the `hphtclean` script. The total run time for both cases includes the time for 16K×16K complex FFT of the grid (on the

	CPU	GPU
Total number of visibilities (a)	2.8×10^8	2.5×10^8
Total runtime (sec)	50820	540
Time for gridding (sec) (b)	50252	153
Average rate of gridding (vis/sec) (a/b)	5.6×10^3	1.5×10^6
Cumulative time for computing CF phase screens (sec)	∼166	∼156

Table 1: Comparison between a single-threaded CPU gridder and HPG on a single GPU for W-only projection imaging.

CPU this takes ∼5 min, while on the GPU it takes ∼1 min). In terms of the end-to-end run-time, the implied speed-up is therefore about ∼94x.

For mosaic imaging, a phase screen for the CF is also needed per unique `FIELD` ID. For the comparison shown in Table 1 this calculation was done on the CPU for tests with both, the CPU-gridder and the HPG. The total run-time also includes the cumulative time to compute the CF phase screen per FIELD on the CPU (∼2 min). However for the HPG this calculation can be done more efficiently on-the-fly on the GPU without any measurable impact on the runtime performance (i.e., essentially for free). Since the time of writing this report, these calculations are indeed moved to the GPU with the expected improvement in the total runtime. With the calculations for the CF phase screen done on-demand on the GPU the runtime reduces to ∼7 min with the implied speed-up of ∼120x.

3.3 Performance comparison for gridding: The A-only projection case

Similar tests were also done for the `wprojplanes=1` and `wbawp=True` case. This loads CFs for `w=0` and all SPWs once on the GPU. Algorithmically this corresponds to wide-band A-only projection imaging (no W-term correction). The support sizes in pixels for the 16 CFs used for imaging were:

```
9 10 10 11 11 12 12 13 13 14 14 15 15 16 16 17
```

The total run time for this configuration of AW-Projection was ∼8 min which corresponds to a speedup ∼105x. The (average) time for gridding alone was ∼120 sec.

3.4 Performance comparison for gridding: The AW-Projection case

Tests for the full AW-Projection were also similarly done using `roadrunner` with the `hphtclean` script configured for end-to-end imaging (to convergence) of a VLASS dataset with the imaging setup as in Tobin et al. (2020). Full AW-Projection requires calculations for $N_w \times N_\nu$ CFs where N_w is the number of w-terms and N_ν is the number of bins along the frequency axis for A-term. With $N_w = 32$ and $N_\nu = 16, 512$ CFs are required. The distribution of the support sizes of the CFs is shown in Table 2. Runtimes for the various significant steps are shown in Table 3.

SPW \ W —>	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
00	09	09	10	11	12	13	15	17	20	25	31	37	39	39	45	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
01	10	10	10	11	12	13	15	17	20	24	29	35	38	41	43	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
02	10	10	11	11	12	13	15	17	20	23	28	33	39	45	45	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
03	11	11	11	12	13	14	15	17	19	23	27	31	37	44	49	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
04	11	11	12	12	13	14	15	18	19	22	25	30	35	42	48	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
05	12	12	12	13	14	15	16	17	19	22	25	29	34	40	46	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
06	12	12	13	13	14	15	16	17	20	21	25	28	33	38	44	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
07	13	13	13	14	14	15	17	18	20	21	24	27	32	36	44	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
08	13	13	14	14	15	16	17	18	20	21	24	27	30	35	40	49	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
09	14	14	14	15	15	16	18	18	20	22	24	27	30	34	39	46	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
10	14	14	15	15	16	17	18	19	20	22	23	26	31	33	38	45	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
11	15	15	15	16	16	17	18	19	20	22	24	26	29	35	37	41	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
12	15	15	16	16	17	18	18	19	21	22	24	26	29	34	35	40	48	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
13	16	16	16	17	17	18	19	20	21	22	25	26	28	32	38	39	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
14	16	16	17	17	18	18	19	21	22	23	25	26	28	31	37	38	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
15	17	17	17	17	18	19	20	21	22	23	25	26	28	31	36	40	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50

Table 2: CF support size distribution for the AW-Projection test with `wprojplanes=32`. The support size saturates at 50 pixels for higher w-values due to the size of the intermediate buffer size used for making the CFs. The impact of this saturation on the imaging performance was insignificant. The aggregate volume of these CFs is ~20 GB.

For a large-area mosaic image, in general there is significant emission in the direction of the 1^{st} PB-sidelobe. The A-term also therefore needs to represent at least the first sidelobe sufficiently accurately. With the CFs oversampled by 20x the total memory footprint of all the CFs is larger than the available memory on the GPU used (NVIDIA V100). As mentioned earlier also, a `CFCache`-paging algorithm (the `LAZYFILL` mode with `gridder=awproject`) is therefore used that loads a subset of the `CFCache` and pages-out the subset that is not required anymore (or not required immediately). With the default data iteration setup in the imaging framework, the required CF-subset for VLASS data (all w-CFs for a given SPW) needs to be loaded frequently on the GPU and the resulting runtime overhead becomes a limiting factor in the overall runtime performance.

After some analysis it was found that changing the data iteration axis to `SPW` ID (data served for all fields for a given `SPW` ID) offers two significant advantages. First, the CF-paging frequency reduces since a CF-slice along the w-axis (all w-CFs for a given SPW/frequency) is loaded only once per gridding cycle per SPW. Secondly, with the number of fields in the database greater than the number of frequency bins (here SPWs), the frequency of CF paging-in and -out also automatically reduces. This also improves the i/o efficiency for reading CFs from the disk `CFCache`[5]. This data iteration setup was therefore used for end-to-end VLASS imaging test for AW-Projection.

With the significant reduction in the runtime for gridding with the HPG, the time to read, package and load a CF-subset, all of which is done serially on the CPU was still the limiting factor resulting in an efficiency loss of $2-3$x. To mitigate this overhead, the data iteration pattern in `roadrunner` was modified to pre-fetch and package the CF-slice in parallel with the gridding on the GPU, as shown in Fig. 2. With these modifications (change in data iteration axis and pre-fetching a CF-slice) the runtime for degridding+gridding operation alone dropped by about 3x while the overall runtime improved by about 2x as shown in Fig. 4. Measurements of the runtime for a number of residual image computations are shown this plot. The curves with the "`1Th`" label (open square symbols) are for a single thread used for both data iterations and CF loading. Measurements shown as individual data points were collected for individual major cycles in an end-to-end imaging that included application of the minor cycle algorithm at the end of each major cycle. This results in the fluctuations seen in these curves due to the compute Operating System's memory management. The fluctuations therefore do not reflect real variations in the runtime for residual image computation. For the comparisons below, the smallest value was therefore used as the representative value. The curves with the "`2Th`" label (star symbols) correspond to measurements made with the data iteration pattern shown in Fig. 4. For these measurements only the residual image was computed repeatedly (i.e. no minor cycle algorithms invoked). These curves therefore do not show similar fluctuations.

Fig. 5 shows a similar test for full image reconstruction using the `hphtclean` script (see Fig. 3 and Appendix A) configured for AW-Projection with the runtime shown separately for the various significant steps in the process. The runtime using the HPG for degridding+gridding compared to a 16x parallel run on the CPU corresponds to a speedup of about 12x. Corresponding speed-up in the total runtime is about 6.5x (see Table 3 and Fig. 5). Compared to the runtime on a single CPU-core, these translate to a speedup of about 120x and 70x respectively. Note that the end-to-end run time includes the contribution of all the steps including the overheads as shown in Table 3. The runtime for the minor cycle (labeled as "`Model`" in Fig. 5) is comparable to degridding+gridding runtime. At this point Amdahl's law limits the overall

[5]Specifically, it reduced the load on the `luster` file system's Meta Data Server which otherwise led to sever degradation in the i/o bandwidth.

	Model/minor cycle (sec)	**Weights+PSF** (sec)	**Gather** (sec)	**Residual cycle overhead** (sec)	**De-gridding+ gridding** (sec)	**Total runtime** (sec)
htclean (16x parallel)	1078	1014	77	1200	13204	16573
HPG singlethread	958	349	38	286	2717	4348
HPG multithread	931	320	26	202	1062	2541

Table 3: Comparison between a single-threaded CPU gridder and HPG on a single GPU for an imaging-to-convergence run. The runtime for the various steps is listed in the columns in seconds, with the total run time in the last column. "HPG multithread" timing corresponds to data iteration pattern shown in Fig. 2 (Data courtesy the SCG).

efficiency (speedup of 70x) due to the minor-cycle and other serial operations in the imaging workflow. Few of these serial operations can also be done in parallel on the CPU. Once this is done, further improvements in the overall efficiency is expected in general, e.g. for pointed mosaic imaging with larger amount of data per FIELD than in the test data used here.

4 Conclusions

The primary goal of the work described here is to establish a baseline runtime performance for end-to-end imaging (imaging to convergence) using the HPG. The development work needed to interface the HPG implementation with the rest of the CASA's imaging framework was done as part of this work and is described in Section 2. The resulting software was then used to measure the runtime efficiency (as the speedup compared to a single CPU core) of imaging for use-cases that should benefit from the use of the HPG – namely the application of the A-, W- and AW-Projection algorithms. VLASS observing setup requires use of most of the terms in the imaging equation that exercise degridding/gridding implementation used. It therefore covers a large range of use cases and is good for verifying the HPG implementation and characterizing its runtime performance. A typical VLASS dataset was therefore used for these tests.

To compare with the runtime tests by Pokorny (2021), first a single CF was used for making the residual image (Sec. 3.1). The CF was transferred once upfront and the data were transferred iteratively to the GPU from the CPU over the CPU-GPU bus. The measured speedup was in the $120-150$x range, which is consistent with the measurements by Pokorny (2021) for CF of comparable support size. I.e., we measure no significant loss in overall runtime efficiency due to the use of the imaging framework and data i/o overheads in transferring data over the CPU-GPU bus.

Tests for W- and A-Projection require use of multiple CFs and associated mapping of the CFs to the individual visibilities (Secs. 3.2 and 3.3). The aggregate memory footprint for these CFs was small enough to fit in the available memory on the GPU. All the CFs could therefore be transferred to the GPU at the beginning to the gridding cycle and used internally on the GPU for gridding. The measured speedup for the residual image computation was ∼105x and ∼120x for W- and A-Projection respectively. Runtime efficiency with HPG varies with the CF support size. With the CF support size varying with frequency (for A-Projection test) and W (for W-Projection test), the measured efficiency for these cases is also consistent with the upper limits measured by Pokorny (2021). We therefore conclude that even for use-cases which require multiple CFs, but all of which fit in the GPU memory and are transferred once, it is possible to harvest most of the runtime efficiency.

The total memory footprint for all CFs required for the application of the AW-Projection algorithm is potentially larger than the available memory on typical GPUs. For AW-Projection, CFs for appropriate slices through the A-W plane of CFs is required to be *repeatedly* transferred to the GPU as a function of frequency and time in the data. With the default data iteration pattern in CASA's imaging framework, the overhead of this repeated transfer is prohibitive for VLASS observing setup and reduces the overall runtime efficiency by $2-3$x. As described in Sec. 3.4, with changes in the data iteration axis and pre-fetching the appropriate set of CFs on the CPU in parallel with gridding on the GPU, the runtime dropped by $2-3$x recovering the runtime speedup of ∼120x for making the residual image, which is comparable to the expected value. At this point the runtime cost of gridding becomes comparable to the runtime for the minor cycle and other serial operations in the imaging workflow. The overall speedup for an end-to-end imaging that includes the cost of the minor-cycle algorithm is therefore limited by Amdahl's law to ∼70x. Some of these serial operations, currently done on the CPU, can also be done in parallel which should improve the end-to-end runtime efficiency in general. E.g., loading and packing the visibility data is currently done serially with gridding. While its runtime cost is low for the dataset used here (see Fig. 4), the cost scales with the data volume per FIELD and will become significant, e.g. for wide-band pointed mosaic imaging.

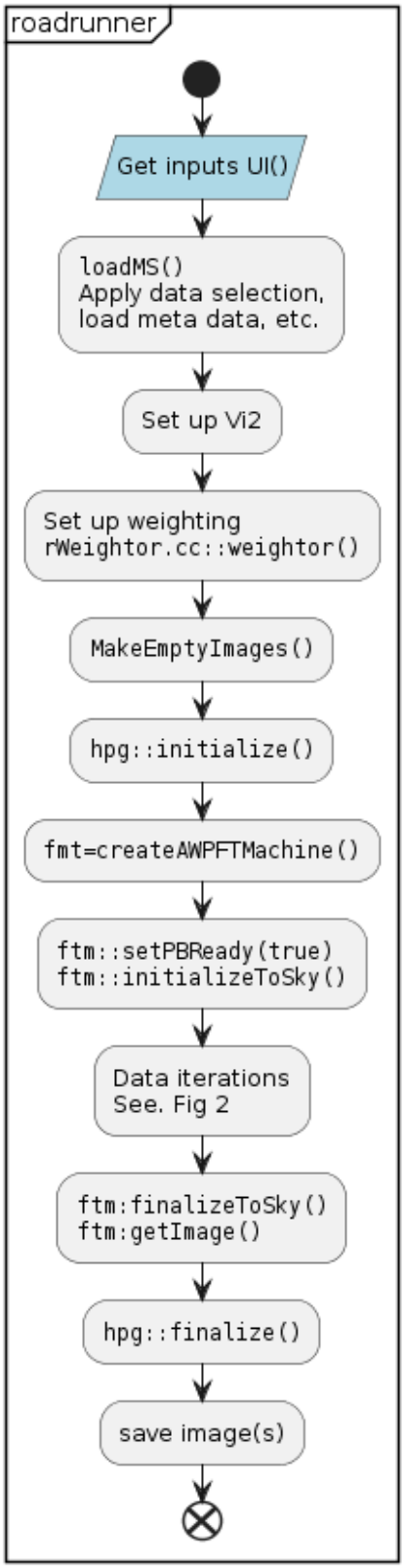

Figure 1: The `roadrunner` workflow. Also see Fig. 2 for the implementation of the `Data Iterations` step.

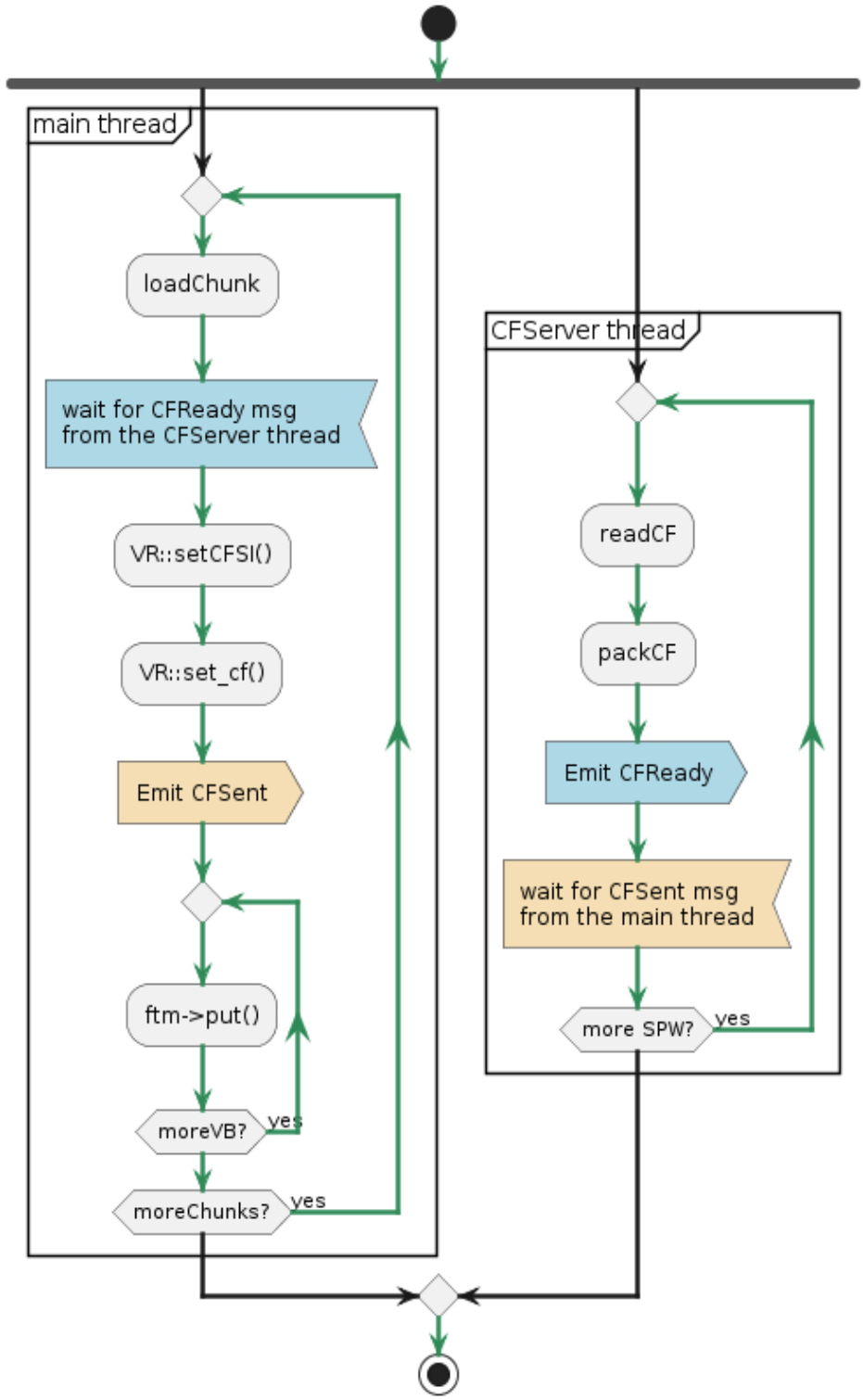

Figure 2: The implementation of the `Data Iterations` step in Fig. 1 for pre-fetching and packaging CFs in parallel with gridding. The boxes titled `main` and `cfserver` are the two threads. The wheat- and lightblue-coloured steps indicate coordinated messaging between the two threads using the `ThreadCoordinator` object.

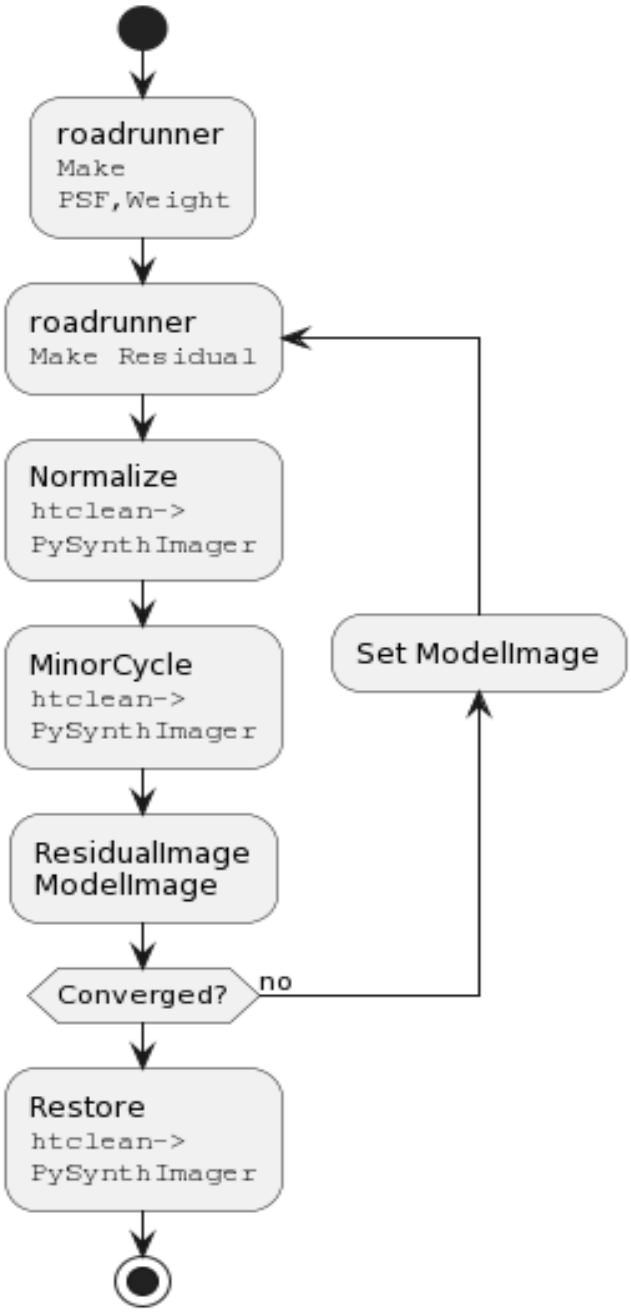

Figure 3: Implementation of the major-cycle iterations in the `hphtclean.sh` script (see Appendix A for the code listing). The `roadrunner` application is used for making images (PSF, Weight, Residual images). Image normalization, application of the minor cycle algorithms, and computation of the final (restored) images is done using the `HTCSynthesisImager` class (via `htclean.py`) which uses the `PySynthesisImager` tool of the CASA package.

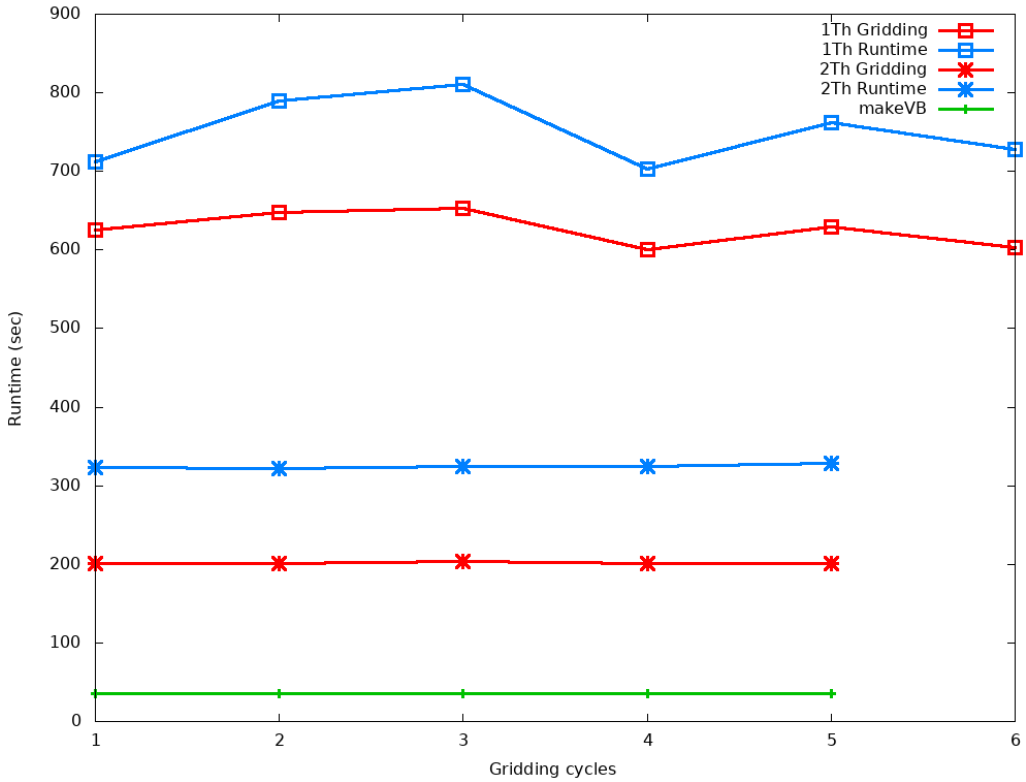

Figure 4: Curves labeled "`2Th`" show the baseline runtime performance improvement when using the data iteration pattern shown in Fig. 2. Graphs labeled "`1Th`" show the runtime with a single thread for gridding and CF preparation. Curve labeled "`makeVB`" shows the cumulative runtime cost of loading and packing the visibility data.

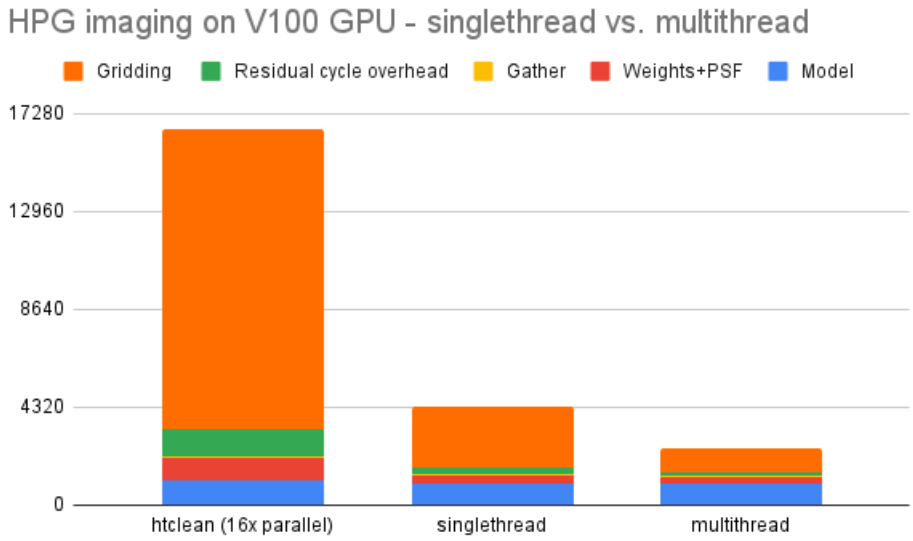

Figure 5: Comparative runtime distribution for the various steps in an iterative image reconstruction using a VLASS data set (16 SPW, `wprojplanes=32`). The y-axis shows the runtime in seconds. Label "`htclean (16x parallel)`" refers to a 16-way parallel run on the CPU. "`singlethread`" and "`multithread`" refer to the equivalent runs using the HPG with the data iteration pattern shown in Fig. 2 used for the latter (Figure courtesy the SCG).

References

Bhatnagar, S. 2022, Design of the AWProject framework, Tech. rep., ARDG Memo, https://safe.nrao.edu/wiki/pub/Software/Algorithms/WebHome/AWPDesign.pdf

Pokorny, M. 2021, High Performance Gridding, Tech. rep., https://library.nrao.edu/public/memos/ngvla/NGVLAC_05.pdf

Tobin, J. J., Marvil, J., Ott, J., & Bhatnagar, S. 2020, VLASS Imaging Project: Proposed Workflow, Tech. rep., VLASS Memo 15, https://library.nrao.edu/public/memos/vla/vlass/VLASS_015.pdf

Appendix

A Listing of the `hphtclean.sh` script

```
#!/bin/bash

##PBS -V    # Export all environment variables from the qsub commands environment to the batch job.
##PBS -l hostlist=nmpost0[21-50]
##PBS -l nodes=1:ppn=1,mem=30gb       # Amount of memory needed by each process (ppn) in the job.
##PBS -d /lustre/aoc/sciops/fmadsen/htclean/test
##PBS -m abe                 # Send mail on begin, end and abort

OMP_NUM_THREADS=1

CASAHOME=/home/yaghan3/sanjay/CAS-13581/casa5

CASABIN=${CASAHOME}/linux_64b/bin/casa
RRBIN=${CASAHOME}/code/build-linux_64b/synthesis/roadrunner

partname="serial"
imgname="htclean_casa"
parfile="htclean.params"
ncycle=20
nparts=1
restart=1
BIN=/lustre/aoc/sciops/fmadsen/tickets/scg-136/imaging_tests/bin/
GO="\n inp \n go \n";
GPUENGINE=1

if [ $# -lt 1 ];
then
    echo "Usage: $0 #ofModelCycles [0/1]"
    exit 1;
else
    ncycle=$1;
    if [ $# -gt 1 ];
    then
       restart=$2;
    fi
fi
#
#-------------------------------------------------------------------------------------
#
echo "Using CASA binary: "${CASABIN}
echo "Using roadrunner binary: "${RRBIN}

#mkdir -p cfcache

if [ "$restart" -eq "0" ]
then
    if [ "$GPUENGINE" -eq "1" ]
    then
        #echo "Computing initial images...";
        # makeWeights
        #echo "Making weight images using HPG...";
        echo -e "load weight.def $GO" | ${RRBIN} help=dbg 2>| weight.out

        # makePSF
        #echo "Making PSF using HPG...";
        echo -e "load psf.def $GO" | ${RRBIN} help=dbg 2>| psf.out

        # normalize the PSF
        ${CASABIN} --nologger --nogui -c ${BIN}htclean.py gatherPSF ${parfile} serial >& gatherPSF.out

        # make the Primary beam
        ${CASABIN} --nologger --nogui -c ${BIN}htclean.py makePrimaryBeam ${parfile} serial >& makePB.out
```

```
63
64          #echo "Making Dirty image using HPG...";
65          # make dirty image
66          echo -e "load dirty.def $GO" | ${RRBIN} help=dbg 2>| dirty.out
67
68      else
69          # CPU-based PSF
70          ${CASABIN} --nologger --nogui -c ${BIN}htclean.py makePSF ${parfile} serial >& makePSF.out
71
72          # normalize the PSF
73          ${CASABIN} --nologger --nogui -c ${BIN}htclean.py gatherPSF ${parfile} serial >& gatherPSF.out
74
75          # make the Primary beam
76          ${CASABIN} --nologger --nogui -c ${BIN}htclean.py makePrimaryBeam ${parfile} serial >& makePB.out
77
78          # CPU-based residual computation
79          ${CASABIN} --nologger --nogui -c ${BIN}htclean.py runResidualCycle ${parfile} serial >&
    runResidualCycle_cycle0.out
80      fi
81      # normalize the residual
82      ${CASABIN} --nologger --nogui -c ${BIN}htclean.py gather ${parfile} serial >& gather_cycle0.out
83  else
84      echo "Doing only the update step..."
85  fi
86
87  i=1
88
89  while [ ! -f stopIMCycles ] && [ "${i}" -lt "${ncycle}" ]
90  do
91      ${CASABIN} --nologger --nogui -c ${BIN}htclean.py runModelCycle ${parfile} ${partname} >&
    runModelCycle_cycle${i}.out
92
93      if [ "$GPUENGINE" -eq "1" ]
94      then
95          echo -e "load residual.def $GO" | ${RRBIN} help=dbg 2>| runResidualCycle_cycle${i}.out
96      else
97          ${CASABIN} --nologger --nogui -c ${BIN}htclean.py runResidualCycle ${parfile} ${partname} >&
    runResidualCycle_cycle${i}.out
98      fi
99
100     ${CASABIN} --nologger --nogui -c ${BIN}htclean.py gather ${parfile} ${partname} >&
    gather_cycle${i}.out
101     # \rm -rf ${imgname}.dirty.cycle${i}
102     # cp -r ${imgname}.residual ${imgname}.dirty.cycle${i}
103
104     i=$((i+1))
105 done
106
107 ${CASABIN} --nologger --nogui -c ${BIN}htclean.py makeFinalImages ${parfile} ${partname} >&
    makeFinalImages.out
```