Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Paralellization of the ddf-pipeline & introduction to radio astronomy

Author: Tom van Hateren
VU: 2532560 / UVA: 12013536

1st supervisor:	Adam Belloum
daily supervisor:	Hanno Spreeuw (Netherlands eScience Center)
2nd reader:	Reggie Cushing

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 2019

Abstract

With the ever increasing demands for sensitivity in radio astronomy, telescopes have made a switch from large dishes to many smaller antennas. This brings new issues to the field in combining the signals from the individual antennas or stations. With the large number of antennas and stations, data rates increase and processing becomes a challenge. We introduce the processing that is performed on the LOFAR radio telescope to turn signals from the antennas into an image. We explore two direction-dependent calibration and imaging pipelines, Factor and ddf-pipeline, that are used for this purpose and the parallelism already found in these. We discuss how the ddf-pipeline can be further parallelized to make use of multiple nodes. And finally we discuss and test an implementation for the direction-dependent calibration package in the ddf-pipeline, killMS.

Contents

Abstract **iii**

Contents **v**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

2 LOFAR architecture and calibration **5**
2.1 From antenna to archive 5
2.2 Calibration 6
2.2.1 Self-calibration 7
2.2.2 Direction-independent calibration 7
2.2.3 Direction-dependent calibration 8
2.3 Imaging 10

3 Running calibration packages **13**
3.1 Data 13
3.1.1 Comparison of the images 17
3.2 Containerization 17

4 Architecture of the ddf-pipeline **19**
4.1 DDFacet 19
4.1.1 Deconvolution 19
4.1.2 Parallelization 21
4.2 killMS 22
4.3 Time is short 22

5 Implementation and testing **25**
5.1 Input 25
5.2 Design decisions 25
5.3 Workers 26

5.4 Performance testing 27

6 Conclusions and Future Work **29**

A Code samples **31**

References **35**

List of Figures

1.1 LOFAR 2

2.1 Self-calibration 7
2.2 *checkfactor* tool: showing the facets and their processing status. 9
2.3 ddf-pipepline architecture with killMS and DDFacet 10

3.1 Images of the L232875 observation side by side. This is showing a section from the center of the images. 15
3.2 Example of the self-calibration of a facet in Factor. The captions describe the phase of calibration. Image (a) is made with only the direction independent solutions. Image (b) and (c) show the image during the TEC phase. TEC, Total Electron Content, is used to describe the ionospheric turbulence. In images (d),(e) and (f) the direction-dependent gain solutions are also included. 16
3.3 Artificial small test dataset with 9 sources aligned in a grid. . 16
3.4 Pixel distribution for the images of L232875 17

4.1 1-dimensional (de)convolution example 20
4.2 ddf-parallel design: with X_D the 'dirty' sky, $\hat{X}$ the sky model, and PSF the point spread function. 21

5.1 killMS_parallel: For each worker a queue is maintained of the MeasurementSets that still need to be processed. From each queue a item is removed, and processed on the respective worker node. When the worker finishes, a log file is saved and the solutions are copied to the kMS_parallel node. 27

List of Tables

3.1 Image noise level 17

5.1 Average killMS runtime in seconds over 10 runs for the small dataset on a personal system. The parallel version runs all worker processes on the same system. 27
5.2 Average killMS runtime in seconds over 10 runs for the small dataset on DAS5. The parallel version runs the worker processes on separate systems. 28

Chapter 1

Introduction

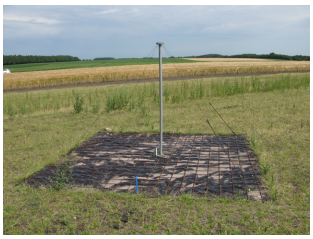
LOFAR[8] is a radio telescope that operates in the 10 to 240 MHz range. The telescope consists of many small, static antennas grouped into stations. Two antenna designs are employed. A low frequency design is used to capture 10 to 80 MHz (Figure 1.1a). The low frequency antennas are dominated by sky noise. For the higher frequencies, from 120 to 240 MHz a different design is used (Figure 1.1b). At the higher frequencies the sky noise is less of a problem. The high frequency antennas are designed to reduce the contributions of the electronics. The stations are spread around Europe, with the core stations located in the northern part of the Netherlands. The LOFAR radio telescope produces tens of terabytes of observations every day.

Traditional radio telescopes use large dishes as antennas. For LOFAR the choice was made to use many smaller antennas, due to the frequencies and sensitivity that are targeted with the project. As the sensitivity of a radio telescope is dependent on the ratio of antenna size to frequency, the low frequency and high sensitivity requirements would result in a very large dish. This is problematic in terms of construction difficulty and cost [5]. The signals from the individual antennas are combined per station to form a single antenna, comparable to a large dish antenna.

The LOFAR telescope is considered to be a pathfinder for the Square Kilometer Array[1], a next-generation radio telescope currently in development. The use of a large number of simple antennas and the data handling and processing challenges that accompany them form a good test-bed for the technologies needed for the new telescope.

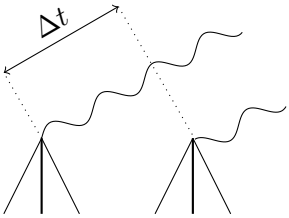
The observations from LOFAR are stored in terms of the correlation of the signal between pairs of stations. This information is not directly usable for science, further processing is needed. A full measurement consists of an observation of the target field generally lasting eight to twelve hours, with a short observation of a calibrator generally lasting five to ten minutes before and after the main observation. A calibrator is a bright source which

[1] `https://www.skatelescope.org/precursors-pathfinders-design-studies/`



(a) LBA antenna[1]. 10 - 80 MHz.

(b) HBA antenna[1]. 120 - 240 MHz.



(c) Delay in reception of the signal between different antennas or stations.

Figure 1.1: LOFAR

has well known properties, in terms of brightness and shape. It is used to estimate the characteristics of the instrument and the environment during the observation of the target field. With the environment we mean any external influences that change the observed signal. This includes factors like the ionosphere (discussed next), and the temperature at the station [7].

Multiple sources contribute to the inaccuracies in the measurements made by the telescope. One of the main contributing factors is the ionosphere. It causes unknown gain phase and amplitude changes [4] to the signal. These differ from station to station and change over time as the ionosphere changes. The sky can be divided into isoplanatic patches [11]. These are areas that have approximately the same ionospheric conditions. Depending on the location of the antenna and the direction the signal is coming from, the signal can pass through a different isoplanatic patch and thus have different conditions affecting it. These are called the direction-dependent effects, and are corrected for by modern calibration packages.

In the LOFAR design, the antennas do not have any moving parts, meaning that the instrument can not physically be pointed at a source, instead targeting happens in software by adjusting the delays at which signals from stations are brought in to be correlated. The signal from the 'closer' stations is delayed by the amount of extra time it takes the signal to reach the 'further' station (Figure 1.1c). For example: by delaying the signal from stations in the east, a pointing towards the east is created. A signal traveling from an eastern direction arrives first at the stations in the east and later at the stations in the west. With the added delay to the eastern stations the arrival times now line up and the signals from that direction are thus the signals found with the highest correlation. The artificial delays compensate for the delays introduced by the physical distance between stations.

This brings us to another source of inaccuracies, the pointing system. The antennas have a particular beam shape. This generally means that

[1] Pictures from:
`http://www.lofar.org/about-lofar/system/sensor-fields/sensor-fields`

the antennas are more sensitive to signals directly in front of the antenna compared to signals coming in at an angle. Specifically for the LOFAR antennas, this means that they are most sensitive for signals straight above the antenna, coming from the zenith. As the pointing gets adjusted to keep track of the target, different parts of the antenna beam are measured in observing the target. This means that the sensitivity of the instrument constantly changes. This needs to also be accounted for in calibration as it would otherwise lead to a lower image quality.

Due to the large number of antennas and stations involved in LOFAR, the processing of observations into images usable for astronomers is a compute intensive task. Modern calibration packages have run-times in the order of days or even weeks [14, 9] to process a single eight-hour observation into an image. The ddf-pipeline specifically takes about four days to complete such an observation [15], when performed on a single fat node[2]. Some calibration packages allow for the use of multiple nodes to speed up the processing (e.g. Factor [19]). For ddf-pipeline this is not yet possible. In this thesis we look into the parallelization of ddf-pipeline from the perspective of a computer scientist.

[2]These fat nodes have 24 physical ‘CPU cores and 512 GB ram

Chapter 2

LOFAR architecture and calibration

In this chapter we will give a general introduction to the data processing happening in the LOFAR telescope and what steps are involved in the calibration of observations. Additionally we discuss another calibration package, Factor, that is also based on the facet calibration technique.

2.1 From antenna to archive

At the stations the signal from each antenna is fed into dedicated hardware that performs analog to digital conversion, signal sampling, and filtering. For each of the individual frequency bands, the signals from the antennas are coherently summed to form the station subbands. This is known as beamforming. In the most common 8-bit mode, 488 subbands can be selected for observation. These subbands can either all be used for one pointing, or spread over multiple pointings. The combined signal is sent via 10 Gigabit Ethernet connections to the central processing cluster located in Groningen for further processing and storage [2].

Originally the central processing facility was implemented using various iterations of a Blue Gene super computer. This was replaced in 2014 by general purpose hardware in the form of COBALT [3]. At the cluster data from each station is processed as it arrives. The data from stations is correlated with the other stations, outlier data is flagged, and the resulting visibilities are averaged in time and frequency to reduce data size to a manageable level for the end user [10]. These first stages of processing are performed in realtime on the COBALT system. Further processing can also be performed as part of the LOFAR processing pipeline. This takes place on a separate 'offline' cluster. For example: additional steps to create images or source catalogs can be included in the pipeline. The early designs for LOFAR already include the automatic processing pipelines, however they are still in

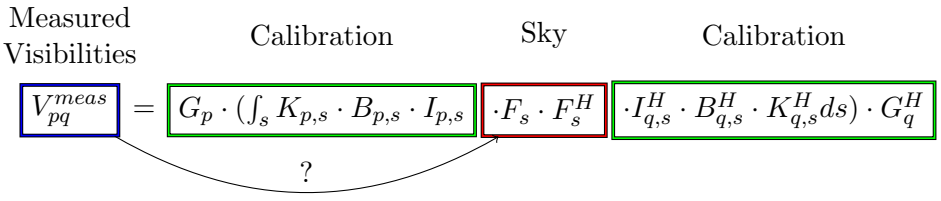
development. Instead all data is processed manually by users. As we want to process the data ourselves, we use the data that results from the Averaging Pipeline. The averaged visibilities are stored in the archive to allow new developments in calibration technology to be retroactively applied to old observations.

The LTA currently consists of three sites in: Amsterdam (NL), Jülich (DE), and Poznań (PL). These sites are accessible via GRID or, if the user does not have a GRID certificate, via a HTTP proxy located at each of the storage sites. Data is stored on tapes. When a request for data comes in, the data is scheduled to be copied from the tapes to hard drive storage from where it can be downloaded by the user.

2.2 Calibration

Once we have acquired the data from the LTA, it is time to start calibration. When we directly image the data from the LTA we get an image dominated by artifacts. To remove these artifacts we perform calibration. The general idea of calibration is that we assume a certain sky, a model, based on previous observations. Then we can calculate what the visibilities would be if we observed that model. From the difference between the observed and calculated visibilities we can derive how the signal is affected and what transformations need to be applied to remove the artifacts from the image.

The changes that transform the signals coming from the sky can be modeled using a so-called Measurement Equation. Each of the effects is modeled as a 2 by 2 Jones matrix and is applied in the sequence in which they affect the signal. Equation 2.1 gives an example of a measurement equation with: G: Gain, K: Geometry, B: Beam, and I: Ionosphere.



(2.1)

The goal of the calibration is to minimize the difference between the predicted values from the model and the measured values from the observation. The calibration processes uses a gradient descent algorithm.

Ongoing developments are aimed at reducing the solution space to speed up calibration and to find more accurate solutions.

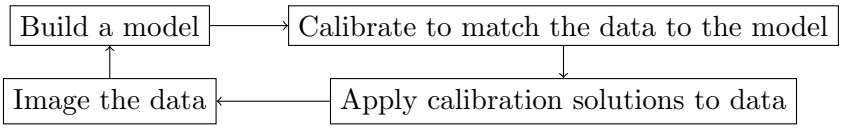


Figure 2.1: Self-calibration

2.2.1 Self-calibration

Both the direction-independent and the direction-dependent calibration make use of self-calibration in which the target is used to calibrate itself. This works by calibrating for bright sources within the target field. Self-calibration is an iterative process. It is illustrated in Figure 2.1. Each iteration a model is generated. For the first iteration, the model is based upon previous observations, or the dirty (uncorrected) image if no data is available. In the later iterations, the model is created from an image of the data with the corrections of the previous iteration applied. From the model, visibilities are generated. Now the parameters of the measurement equation are optimized to fit the generated visibilities to the observed visibilities. The observed data is corrected with the found solutions and a new model is generated from the corrected data. The loop repeats for a number of iterations until the signal to noise ratio limits are hit, or convergence has been reached.

2.2.2 Direction-independent calibration

Both ddf-pipeline and Factor only perform the direction-dependent calibration. The direction-independent calibration is performed by prefactor[1].

The prefactor calibration is separated in a number of steps [19]. The first step in the calibration consists of the flagging of RFI. In the LOFAR frequency range there are a number of strong sources of RFI that need to be filtered out.

Next, the contributions of bright sources are removed. This generally includes A-team sources, which are the brightest sources in the low frequency radio sky. Even though the stations are not necessarily pointed in the direction of these bright sources, via the sidelobes of the antenna beam these can still have a significant contribution. This is followed by a calibration on the calibrator.

The LOFAR core stations are all connected to a single clock for time reference. The remote and international stations have independent clocks, that are synchronized using the GPS signal. The synchronization is not perfect, and thus timing differences are introduced. These are corrected for in the next step.

[1]Source: https://github.com/lofar-astron/prefactor

In the last steps, the solutions from the calibrator are transferred to the target field and another round of calibration is performed, this time on the target field. A solutions file is generated that contains the solutions for the direction independent effects.

2.2.3 Direction-dependent calibration

Direction-dependent calibration primarily involves correcting for the ionosphere and the beam. Both Factor and ddf-pipeline perform calibration using the faceting technique.

The corrections calculated as part of the direction-dependent calibration process are only good for a single point in the sky, for example the image center. If we only calibrate for a single point, then as we get further away from that point, the image quality degrades. Ideally we would perform the calibration for every source in the field. However, this is prohibitively expensive. Instead only a subset of the sources are calibrated for.

An algorithm to solve this is peeling. Peeling is an iterative algorithm where in each iteration the brightest problematic source is calibrated and subtracted from the image.

Another solution is to split up into smaller sections called 'facets'. In Figure 2.2 an example of such a division can be seen. For each facet a solution is calculated. These solutions are then smoothed over to prevent sudden changes in the image across facet borders. The center point of each facet is the brightest source or group of sources in the area. Together, the facets form all the directions that the calibration is performed in. Faceting is seen as an evolution of peeling.

Factor

Factor works by dividing the field of view into facets and solving the direction-dependent corrections in each of them. Factor was designed to be able to work in parallel, making use of a cluster to distribute the work load over multiple nodes. Individual facets can be processed in parallel.

Factor has been designed from the ground up to be parallelized and is build upon LOFAR Generic Pipeline framework. This is an extensible framework for writing data processing applications, developed for the LOFAR project. It is designed to make it easier to run jobs on a cluster. It handles the multiprocessing transparently. Factor assumes a shared filesystem.

The majority of processing in Factor is done in operations [6]. These are steps grouped together into pipelines. Some operations can run in parallel, others can not. Factor defines the following operations: *outlierpeel*: In this step sources that lie outside of the faceting region are removed. *facetpeel*: The facet calibrator is removed, and the facet is imaged. *facetselfcal*: The

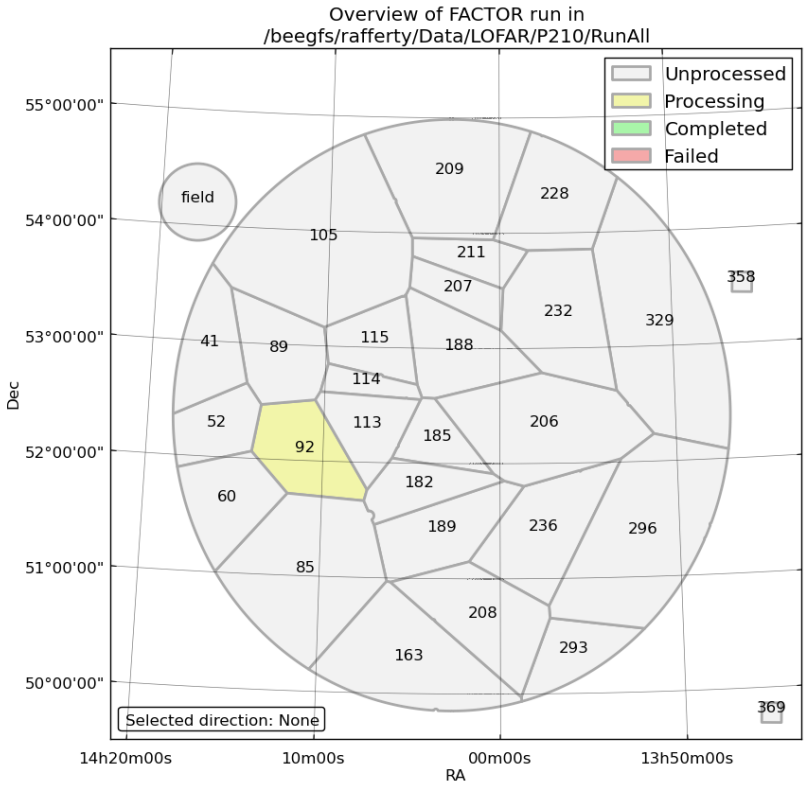


Figure 2.2: *checkfactor* tool: showing the facets and their processing status.

calibrator in the facet is self calibrated, and an image is made with a sampled subset of the full bandwidth. *facetsub*: The improved model from one of the previous operations is used to subtract the facet. *facetimage*: An image is created using the full bandwidth using the solutions from the self calibration step. *fieldmosaic*: Combine the individual facet images together to create the full image.

Once the processing is complete, we get two full images built up from the individual facet images. The image with a filename ending in *correct_mosaic.pbcor.fits* is used for accurate source flux density measurements. The image with a filename ending in *correct_mosaic.pbcut.fits* is used for source detection. Unlike the 'pbcor' image it has not been corrected for the primary beam.

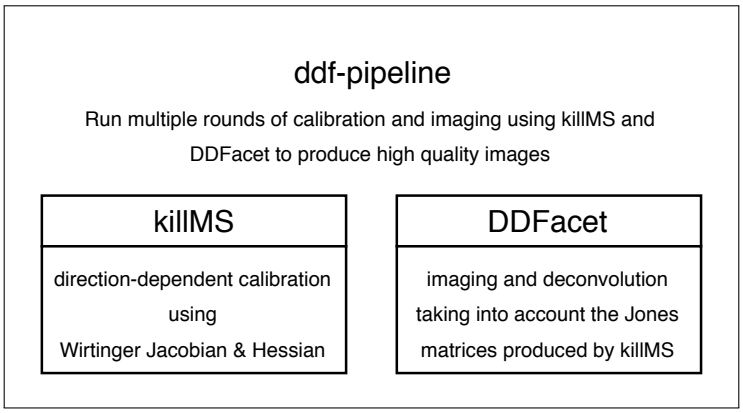


Figure 2.3: ddf-pipepline architecture with killMS and DDFacet

killMS

In the ddf-pipeline (Figure 2.3) calibration is performed by killMS. It can perform direction-dependent self calibration based on a model or image. The calibration in killMS is using the Wirtinger derivative to be able to take some algorithmic shortcuts and simplify calibration [17]. It makes the calibration problem antenna separable. Many of the calculations made in killMS are embarrassingly parallel and can be performed in parallel for directions and stations. These calculations are performed by worker pools working from a job queue with each worker using a single core. The killMS code calls on the DDFacet imager to perform the imaging during the calibration. killMS (and also DDFacet) make use of shared memory for communication of data and results between the different processes.

2.3 Imaging

Imaging is complementary to calibration. In calibration we calculate the Jones Matrices given the measurements and sky model, whereas in imaging we have values for the Jones Matrices and instead want to calculate the sky map.

DDFacet is a facet-based imager that can integrate the killMS calibrations into the imaging process. In Factor the calibrations are used to correct the visibilities and then those corrected visibilities are imaged. In ddf-pipeline, the imager also gets the calibration solutions and can take these into account during the imaging process. The majority of DDFacet is implemented in Python, with a small subset of high-performance code written in C.

In imaging the visibilities are decomposed into the constituent frequencies using the Fast Fourier Transfer (FFT). The FFT requires the data to

be aligned to a regular grid, and so the imaging process includes a step to interpolate the visibilities onto a grid.

The direction-dependent effects can either be taken into account in the Fourier domain using A-projection (used in AWImager), or in the image domain using a facet approach (Factor & DDFacet). The advantage of correcting in the Fourier domain is that the solutions correct the whole image, while in the image domain the solution is only correct for the given direction. However the corrections in the Fourier domain require the Jones Matrices to be provided for the continuous image plane, while most calibration strategies only calculate for a set of discrete directions [18].

Chapter 3

Running calibration packages

We tested ddf-pipeline and Factor to get some experience with calibration and evaluate the performance of the pipelines. The experiments have been performed on a small scale on our local hardware and on the DAS-5[1] cluster at Astron. At the cluster we have nodes available with up to 56 physical cores and up to 512 GB of ram.

Note that we started our experiments with the version of ddf-pipeline used for the first data-release of the LoTSS project. This was the latest public release of DDF & kMS at that moment. In the mean time a new release of both project is in progress as of writing this thesis. With this new release the software has been used on data from a variety of telescopes and has gained in robustness and reporting of potential issues with the data has been added.

3.1 Data

Initially we were given an observation of the calibrator 3C295 from the LOFAR archive to start our experiments and see if we could run the pipeline. With that, we decided to try and directly image the calibrator. As the calibrator observation duration is very short and the calibrator model is already of high quality this should allow the calibration and imaging to finish quickly. The dataset, L151880, is a 2 minute observation of the 3C 295 calibrator spanning 244 frequency bands, each in a separate MeasurementSet. This dataset takes up 4.6 GB of storage and was acquired from the LTA. After processing with prefactor and the averaging that it performs we are left with 60 frequency bands.

When feeding those into ddf-pipeline we ran into issues at the bootstrapping phase. In the bootstrapping phase of ddf-pipeline the flux density scale is adjusted to fit the data to observations made with other telescopes. The large number of frequency bands and the small duration of the observation lead to small MeasurementSets. killMS processes each MeasurementSet in-

dividually, and thus has very little data to work with. It crashes when it does not find any sources in our data, not even the bright calibrator. So then we decided that we better follow the normal processing pipeline and work on the target field instead, to reduce the probability of further problems.

So now we need to find suitable data that represents a real workload without needing terabytes of storage and days of runtime. Initially we selected the LOFAR schools as a good source of datasets. The LOFAR schools datasets are short observations designed specifically as an introduction to the steps in processing observations into images. These observations are designed to be simple as to allow for relatively quick processing during tutorials, but still are real observations made using the telescope. However when running one of these small observations through ddf-pipeline we ran into similar issues as before, although in a different stage of the pipeline. In this case the pipeline crashed at the start on during the direction-independent imaging step with very low signal-to-noise ratios of the sources.

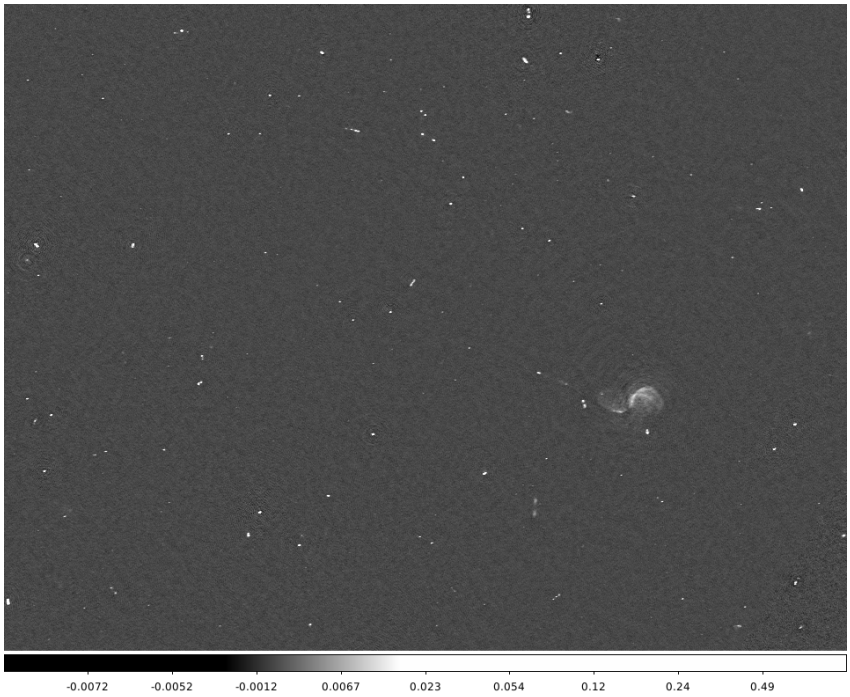
Then we came across the prefactor tutorial from the 2018 LOFAR school[1] which uses a subset of the available subbands of a standard eight hour observation. Here we are using L232873, a ten minute observation of the calibrator 3C196 of which we use 100 subbands. And L232875, an eight hour observation of the target field P23 of which we use 20 of the 237 available subbands in the LTA. The data volume of these two observations together takes up just over 250 GB in storage.

The tutorial discusses the data preparation and direction independent calibration that is performed using prefactor. It ends at the point where one would normally run Factor or ddf-pipeline.

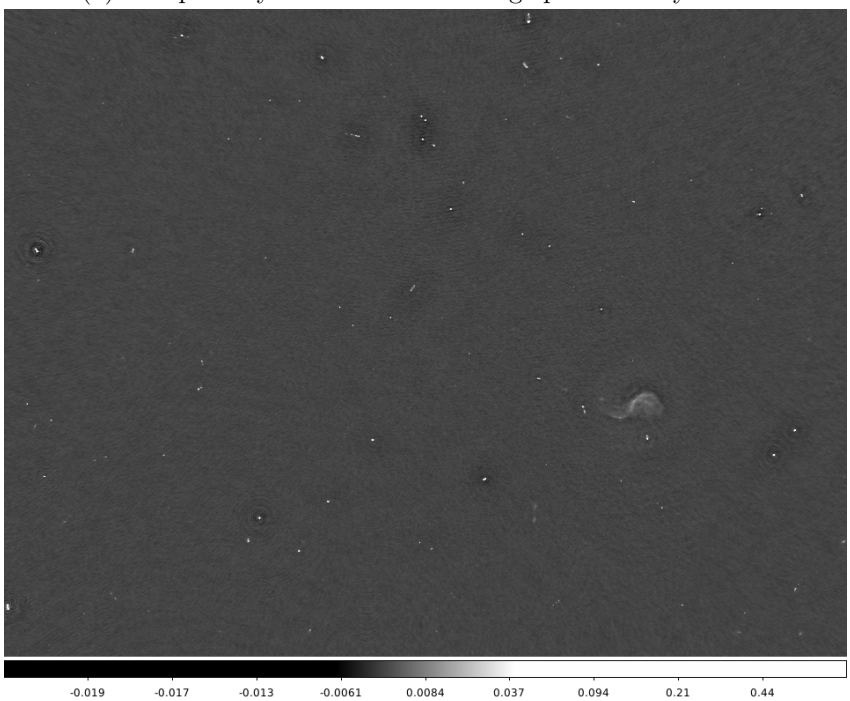
With this dataset we have our first success. After solving various simple problems in ddf-pipeline, bugs that had already been solved in the newer versions, we manage to get our first images. We were also able to run version 2 of Factor on the same data. In Figure 3.1 we show the same region of the image by both calibration packages. The self-calibration of a facet can be seen in Figure 3.2. The direction-dependent calibration and imaging process for both pipelines took approximately two days for ddf-pipeline and five days for Factor on a single 56 core / 512 GB ram node.

For quick testing of our work we got our hands on a small simulated dataset created using MSCreate (Figure 3.3). It contains nine point sources aligned on a grid. This dataset contains two 10 MHz wide subbands and takes up about 200MB. Processing is very quick with a round of calibration with killMS taking about a minute per frequency band.

[1] `http://www.astron.nl/lofarschool2018/Documents/Thursday/prefactor_tutorial.pdf`

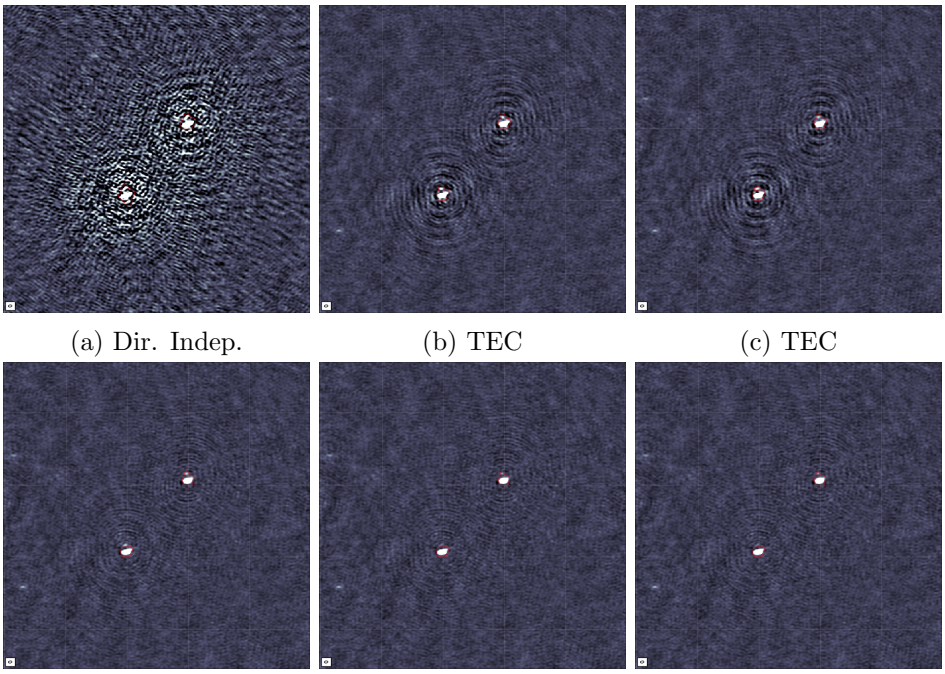


(a) The primary beam corrected image produced by Factor



(b) The image produced by ddf-pipeline

Figure 3.1: Images of the L232875 observation side by side. This is showing a section from the center of the images.



(a) Dir. Indep. (b) TEC (c) TEC

(d) TEC + Gain (e) TEC + Gain (f) TEC + Gain

Figure 3.2: Example of the self-calibration of a facet in Factor. The captions describe the phase of calibration. Image (a) is made with only the direction independent solutions. Image (b) and (c) show the image during the TEC phase. TEC, Total Electron Content, is used to describe the ionospheric turbulence. In images (d),(e) and (f) the direction-dependent gain solutions are also included.

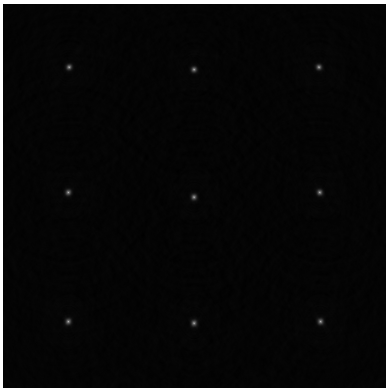


Figure 3.3: Artificial small test dataset with 9 sources aligned in a grid.

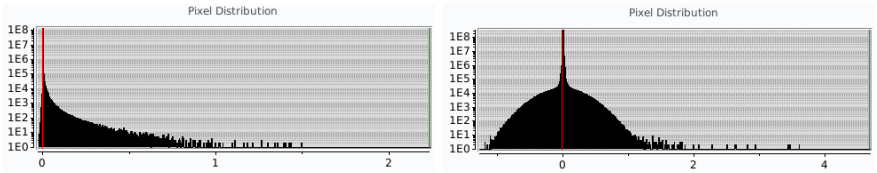
3.1.1 Comparison of the images

We begin by measuring the noise level of both images. We do this by selecting a region without any sources in our image viewer, DS9. We measure the standard deviation of the same empty regions in both images. The Factor images has a consistently lower noise level on this specific dataset (Table 3.1),

	Factor	ddf-pipeline
Region 1	3.21×10^{-4}	4.72×10^{-4}
Region 2	3.02×10^{-4}	4.19×10^{-4}
Region 3	2.90×10^{-4}	5.18×10^{-4}

Table 3.1: Image noise level

When comparing the scale below the images we can see that the maximum values are similar, but the minimal values differ. For ddf-pipeline the values go much deeper into the negatives, this can be seen as the dark areas around sources in the image. It seems the cleaning has gone to deep in the ddf-pipeline.



(a) Factor (b) ddf-pipeline

Figure 3.4: Pixel distribution for the images of L232875

We also include the pixel distribution, as we found an interesting difference in the values. There is a very bright source leading to the high maximum flux in the ddf-pipeline image. For Factor this source is located just outside of the cut off area, so it has been peeled from the image. This explains the difference in the maximum values for both images. For the minimum values we that Factor has a minimum value just below zero, but the minimum value lays past minus one.

3.2 Containerization

As LOFAR is a path finder project, the calibration packages and the software stack that they depend on are under active development. Additionally many of these projects do not have a stable release schedule, or any guarantees of a stable API. This makes it difficult to install many of the required packages.

There are attempts to make the installation easier, for example trough the KERN Suite[12]. However even that does not solve the issues, as currently the LOFAR offline software stack as published via the Astron SVN service is still in the process of migrating from version 2 of the Casacore library to version 3. Casacore is a widely used library for astronomical data processing. For KERN suite version 5, the current release, the software stack is using the newer version 3 of Casacore. This leads to the problem that the LOFAR offline software stack can not be packaged in KERN suite, as it does not compile with Casacore 3. Both factor and ddf-pipeline have a dependency on the LOFAR software stack. We tried to run the ddf-pipeline using KERN, and ran into this issue. The pipeline crashed because it was missing the LOFAR beam model from the offline software stack. Since then the LOFAR beam model has been separated from the main repository into a separate package, allowing it to be build independently and included in KERN.

The main method of using the software packages seems to be migrating from native installation to running inside of containers. Many of the projects now offer Docker templates to make it easier to run the software. In our case we made use of the containerization technology Singularity. The images and recipes we used and adjusted to fit our needs are provided on GitHub and the Singularity Hub by Frits Sweijen[2]. The images are designed to work with the newest releases of the ddf-pipeline. We have adjusted these to work with Factor and the older, public, version of the ddf-pipeline. These changes include patches to fix bugs and outdated code in the public version of the ddf-pipeline.

[2]https://github.com/tikk3r/lofar-grid-hpccloud

Chapter 4

Architecture of the ddf-pipeline

To create the final science-worthy image, ddf-pipeline uses two major components: DDFacet and killMS. DDFacet performs the imaging. killMS performs the direction-dependent calibration. The pipeline currently runs on a single machine. We will now look more deeply into the architecture of both software projects[1].

4.1 DDFacet

DDFacet is an imaging and deconvolution framework. In the ideal case, the visibilities are Nyquist sampled over the entire uv-plane during an observation. The uv-plane is the Fourier transform of the sky. Each pair of antennas forms a baseline, and each baseline measures a single point on the uv-plane. When performing Nyquist sampling of the visibilities there is enough information captured to fully recover the original signal. The Fourier transfer of these visibilities would produce an image of the sky. In practice the sampling of the uv-plane is incomplete. Deconvolution is the process of finding the 'true' signal that is sent out by astronomical sources from the incomplete visibilities.

4.1.1 Deconvolution

Deconvolution generally follows the following principle. The corrections to point sources are captured in a PSF or dirty beam. The PSF, or point spread function, is the Fourier transform of the sampling function. This sampling function has a value of 1 where we measured the data in the uv-plane and a

[1]For a more visual explanation `https://safe.nrao.edu/wiki/pub/Software/Algorithms/CALIM2016Program/CALIM_2016_TASSE.pdf`

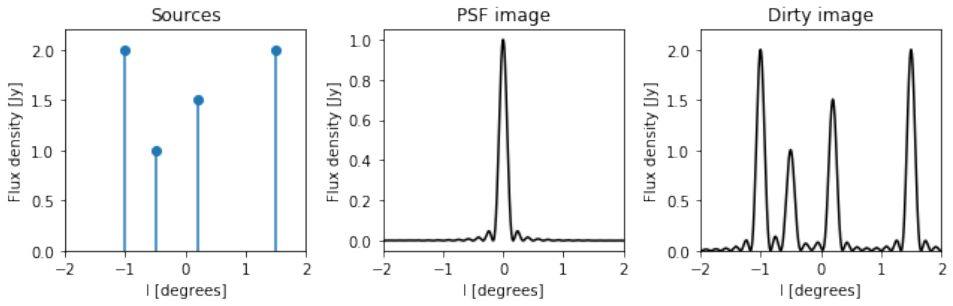


Figure 4.1: 1-dimensional (de)convolution example

value of 0 otherwise. The dirty image can be seen as the convolution of the true image with the PSF.

For the deconvolution we start with the PSF and either an initial model of the sources or the dirty image. If no model is available, bright points in the dirty image are used as initial sources. For each found source, the PSF gets scaled to match size and gets subtracted from the dirty image at some percentage of the full brightness of the source. The resulting image after subtractions is known as the residual image. In the next iteration, the residual image is used to find sources. In the residual image we might find new sources that were hidden in the sidelobes of brighter sources that were subtracted in a previous iteration. Each iteration, the sources that have been found are added to the model and are subtracted from the dirty image. The result of these subtractions is called the residual image. When the signal to noise ration of the residual falls below a threshold, or the maximum number of iterations has been hit the process stops. Taking the final residual and adding to that the model convolved with the clean beam results in the final restored image. The dirty beam has sidelobes. For the clean beam a Gaussian function is used, which has no sidelobes.

For deconvolution in DDFacet, two algorithms are available.For both algorithms, the deconvolution is split into a major and a minor cycle. The minor cycle operates in the image domain. In the minor cycle the PSF is deconvolved from the residual image to create the model. In the major cycle the contribution of the model defined during the minor cycle is subtracted from the visibilities. The result is used to form a new residual image. For HMP, a variation of the MTMS-CLEAN/MS-MFS [13] algorithm, the minor cycle is inherently serial. For more information about this algorithm we refer to [18].

An alternative approach is offered with the SSD (Sub Space Deconvolution) algorithm. In SSD, regions with bright sources are processed in parallel. In the first step of the algorithm these regions, called islands, are isolated and deconvolved independently from each other. As a second step the models are combined and subtracted from the visibilities, which are then

re-imaged. When performing deconvolution on a single island, this will lead to inaccuracies in the flux level estimations as the contributions of other islands are ignored. However, with the combination of the models from individual islands and correction for those in the visibilities, the corrections for an individual island are also applied to the other islands. This still leads to good results over multiple iterations [18].

4.1.2 Parallelization

DDFacet is written with the assumption that it is running on a single node. It makes heavy use of shared memory for sharing data between main thread and the worker threads to reduce the overhead of copying data between different memory spaces.

Previous work has been done in making DDFacet run in parallel. The gridder code, written in C, has support for running in parallel using an OpenMP parallel for loop. However this parallelization is disabled in practice. Instead parallelization is achieved using Python multiprocessing. The deconvolution process has been made to run in parallel over multiple nodes using MPI. This implementation is also unused and has not released to the public.

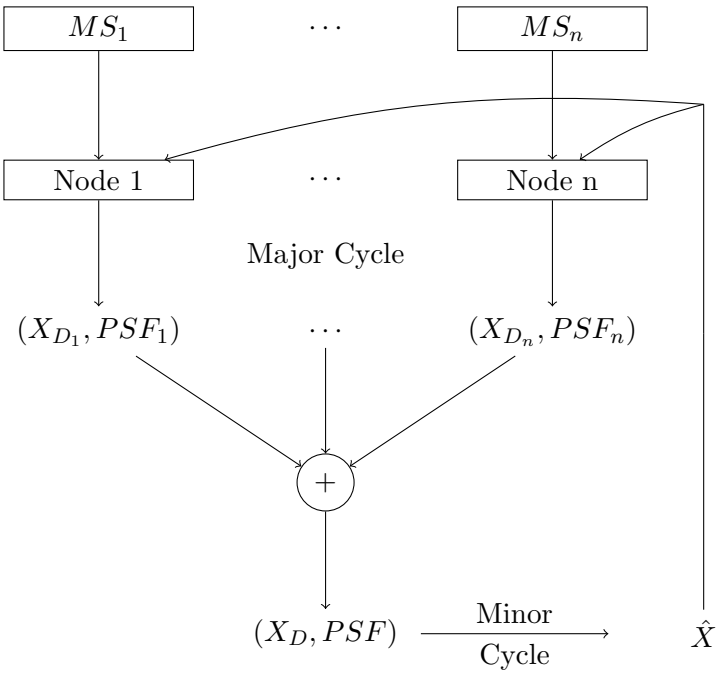


Figure 4.2: ddf-parallel design: with X_D the 'dirty' sky, $\hat{X}$ the sky model, and PSF the point spread function.

In our case we opted for a design where minimal work was expected to

be done on the DDFacet code. We would like to make use of the existing flexibility of DDFacet. The code that handles parallelism over nodes can be written as a separate script that calls DDFacet. In the design (Figure 4.2) each node gets one or more MeasurementSets as input. So, each node handles one or more subbands. These are located on a filesystem that is assumed to be local to the node. The nodes perform the major cycle of the algorithm on their locally available data. The results of the major cycle, the dirty sky X_D and the PSF get transported to a central node where they get combined. The minor cycle is performed by the central node on the combined sky and PSF. The resulting sky model is shared with all nodes which then can perform the next major cycle.

4.2 killMS

killMS is a package that can perform direction-dependent calibration.

Various steps in the calibration can be performed in parallel per direction and station. These jobs are performed using worker pools, like is done in DDFacet. From this we can see that an option for multi-node parallelism would be to distribute batches of station / directions to different nodes, however a much simpler approach is possible.

In [16], the algorithms employed in killMS are discussed. The description includes the specification of solution intervals over time and frequency as a means to achieve a higher signal to noise ratio. In practice killMS runs sequentially over the individual MeasurementSets.

The input has already been averaged in time and frequency as part of the pre-processing steps. Further combining multiple MeasurementSets is not nessecary to reach good calibration solutions. So, all that needs to be done to make it work on multiple nodes is create a wrapper around killMS that runs the calibration jobs for individual MeasurementSets in parallel. For this we do not need the MeasurementSets to be available on a shared filesystem. Each worker node can process the data that is available locally.

4.3 Time is short

In killMS we see that the data is kept separated whereas in DDFacet the data is merged before imaging. In principle the imaging process can also be designed to work with individual MeasurementSets. Then an image is created from the data in each subband, allowing the imaging processes to happen in parallel. Only to create the final image, the images from the individual frequency bands need to be combined together. However, this results in a lower image quality due to the lower signal to noise ratio. Basically, combining the visibilities from multiple subbands and imaging gives better image quality then imaging each subband individually and combining

the images. For Factor a similar thing holds, the *facetsub* operation that removes the effects of the sources in a facet from the visibilities runs on a single node. Even though most of the operations run on multiple nodes, this single operation that is performed for every facet can only run on a single node.

After getting experience with the ddf-pipeline, we started working on the parallel implementation of DDFacet. Due to the time constraints of this project, we decided to switch to implementing the parallel version of killMS instead. In DDFacet we are always limited by the centralized minor cycle, while in killMS all subbands can be processed in parallel. Additionally in a ddf-pipeline run on a full LOFAR dataset, both DDFacet and killMS take approximately equal shares of the total execution time. So the speedup that can be achieved by working on killMS is greater than what can be gained from work on DDFacet. We did not have enough time to work on DDFacet anymore. So, in the next chapter we will discuss the implementation details and show a performance analysis of the parallel implementation of killMS only.

Chapter 5

Implementation and testing

We will first describe some of the details of the parallel implementation for killMS. Then we test the performance of the implementation and extrapolate to the performance on a full dataset. The interesting sections of the code have been included in Appendix A.

5.1 Input

The input data, the MeasurementSets to be processed by DDFacet or killMS, can either be specified as the path to a single MeasurementSet, or as the path to an *mslist* file. This is a text file containing on each line the path to a MeasurementSet. If an mslist is used, killMS will run in batch mode, running itself for each MeasurementSet in the list. The batch mode includes options for skipping already existing solutions, and cleaning of the DDFacet cache that is created during calibration. These options are not available when running killMS on a single MeasurementSet at a time.

For the parallel implementation of killMS the mslist format has been extended to include hostnames next to the MeasurementSet filenames. The fields are separated by a semicolon. The user is responsible for acquiring and distributing the data over the nodes. The filesystem can either be local or shared. Then the user creates an mslist containing the paths to the MeasurementSets and the hosts that store them. Example: '`node103:/data/L232875/01.MS`'. Each node is responsible for processing the data local to that node.

5.2 Design decisions

The parallel implementation of killMS is designed to be used as an in-place upgrade. In the command that is used we can replace 'kMS.py' with 'kMS_parallel.py'. All arguments are passed through to the underlying

killMS processes. The only other change needed is to create the mslist specifying hosts and data. When running killMS within a pipeline, we assume the full pipeline is parallelized. When this is the case, the data (images and visibilities) on the individual nodes are kept up to date by the components of the pipeline. When only killMS is parallelized and the rest of the pipeline runs on a single node, data still needs be moved between nodes. This can either be done in killMS or in the pipeline.

We specifically designed the implementation to work on a non-shared filesystem, as code designed for such an environment will also function on a cluster with a shared filesystem. The code is more flexible and thus can run on a more varied set of systems.

A task can be deployed to nodes in many ways. For example on the DAS5 cluster that we used for testing, SLURM is used to schedule jobs. Integrating with a scheduler has the advantage of only allocating nodes when they are needed. However, as the data is assumed to be available only to a specific node, the dynamic nature of the scheduler makes less sense. It also puts a requirement on the infrastructure to provide the specific scheduler we implement for, which we prefer to avoid. So, we decided to use SSH for running jobs, as it does not require any special setup on the nodes and should be compatible with most if not all compute infrastructure. With SSH also comes support for SFTP. The SSH library we used, Paramiko, also allows us to use SFTP to copy configuration files and results back and forth between nodes.

5.3 Workers

The implementation makes use of a worker pool. For each distinct node we keep a queue of MeasurementSets to be process. These queues are filled with the respective entries from the mslist. The code has the same features as batch mode. These features are only enabled when the respective flags are set, as they are in batch mode. We can skip MeasurementSets for which solutions already exist and we can remove the cache for DDFacet after processing of an individual MeasurementSet has completed.

The main loop of the application consists of scheduling a round of processing on the worker nodes, and waiting for them to complete. As kMS_parallel starts we store the configuration, specified in a parset file and in the arguments passed when calling the application, to a new parset file. When a killMS job is about to run on a node, we copy the configuration file to the respective node over SFTP. The killMS process on the node can read the desired configuration from this file. When a worker node finished, we store the output of the process to a log file. We also copy the solutions to the master node, and clean up the DDFacet cache if requested.

The output of the parallel implementation has been verified to be equal

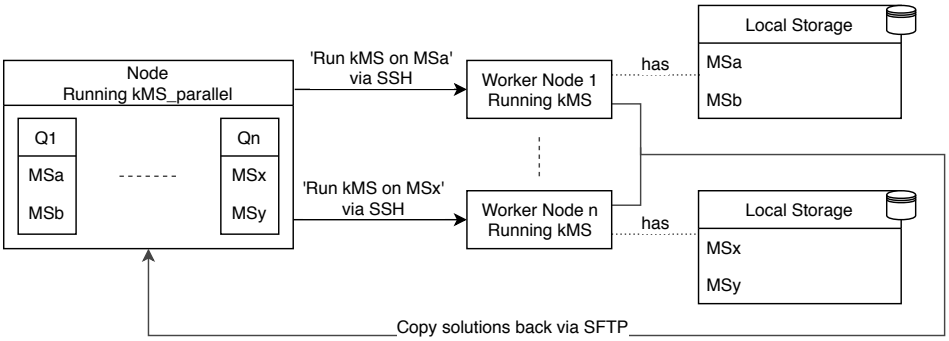


Figure 5.1: killMS_parallel: For each worker a queue is maintained of the MeasurementSets that still need to be processed. From each queue a item is removed, and processed on the respective worker node. When the worker finishes, a log file is saved and the solutions are copied to the kMS_parallel node.

to the solutions generated by a linear run on the same data.

5.4 Performance testing

For the first test we run killMS and the parallel version on the small dataset mentioned in section 3.1. We run both versions on a single system. For the parallel version, we create two entries in the system hosts file that point to the loopback address. This way both jobs are run on the same system in parallel.

	MS0	MS1	Total
Linear	317.4 ± 1.0	425.2 ± 4.6	742.6 ± 5.0
Parallel	452.8 ± 3.4	745.4 ± 7.1	748.0 ± 7.3

Table 5.1: Average killMS runtime in seconds over 10 runs for the small dataset on a personal system. The parallel version runs all worker processes on the same system.

In Table 5.1 we see that the linear and parallel versions take approximately the same total time to finish when the parallel version runs both processes on the same system. From this we can deduce that the parallelization within a single killMS instance is good enough to make full use of the system capabilities. Running multiple instances of killMS on the same node has no benefit.

In Table 5.2 we see that with two nodes the total processing time is slightly longer than the longest time for the individual MeasurementSets. That the parallel version takes two to three seconds longer per Measure-

	MS0	MS1	Total
Linear	178.0 ± 0.4	338.4 ± 0.9	516.4 ± 0.9
Parallel	181.1 ± 0.5	340.3 ± 0.8	343.6 ± 1.0

Table 5.2: Average killMS runtime in seconds over 10 runs for the small dataset on DAS5. The parallel version runs the worker processes on separate systems.

mentSet can be at least partially explained by the fact that we run the code inside containers. In the linear version we are already running within the container, and so no additional loading is needed. In the parallel version the measured time includes the time needed for startup and shutdown of the container.

Overall this means we can cut the processing time for killMS by a factor N, where N is the number of MeasurementSets or subbands available. For a standard LOFAR observation, after direction-independent calibration, this number is 24 subbands. For the LoTSS surveys, currently the main focus of the ddf-pipeline, this improvement is not important. They have enough data coming in to be able to fill a number of nodes each with individual observations. There is also no need to process observations within a short time frame, all observations are archived and can be retrieved for the foreseeable future. For a smaller scale, when only one or a couple of observations are needed, or when a specific observation needs to be imaged quickly, running the parallel version can make more sense.

Chapter 6

Conclusions and Future Work

We described the steps involved in creating an image from a LOFAR observation. From pre-processing to calibration to imaging. We explored factor and the ddf-pipeline. Factor has been designed from the ground up to be parallelized over multiple nodes. The ddf-pipeline has been gaining in parallelism in smaller steps. We presented a parallel design for the major components of the ddf-pipeline, DDFacet and killMS. We implemented and tested the parallel design for killMS. The linear version of killMS already makes good use of the performance available on a single node. The parallel version of killMS can scale out linearly over the available number of MeasurementSets, assuming enough nodes are available to handle each MeasurementSet separately. The processing takes as long as it takes to process the largest MeasurementSet. The processing of individual MeasurementSets takes approximately the same as processing them one by one. We have succeeded only partly in parallelizing the ddf-pipeline, but also provide a starting point for continued effort to do so.

For future work we can suggest the implementation of parallelism in DDFacet, to further speed up the ddf-pipeline. We hope that our work can help with this.

The killMS_parallel code can also be extended to better support the shared filesystem use case. For a shared filesystem it does not matter which node works on which data, so a single queue can be used instead of the queue per node strategy currently used. This allows for better utilization of the available nodes. As soon as a node finishes a job processing of any of the other MeasurementSets can be scheduled on that node.

Appendix A

Code samples

```
# While there are still MS to process
for ThisNodeName in DicoNodes.keys():
    # If thisNode has more MS to process
    if len(DicoNodes[ThisNodeName]["ListMS"]) > 0:
        MSName = DicoNodes[ThisNodeName]["ListMS"].pop()
        MSBaseName = os.path.basename(os.path.normpath(MSName))

        SolsFilename = None
        if options.SolsDir is None:
            SolsFilename = os.path.join(MSName, "killMS.%s.sols.npz" %
             ↪  self.SolsName)
        else:
            SolsFilename = os.path.join(options.SolsDir,
                                        MSBaseName, "killMS.%s.sols.npz" %
                                         ↪  self.SolsName)

        if options.SkipExistingSols:
            print>> log, "Checking %s" % SolsFilename
            if os.path.isfile(SolsFilename):
                print>> log, ModColor.Str("Solution file %s exists" %
                 ↪  SolsFilename)
                print>> log, ModColor.Str("   SKIPPING")
                continue

        if options.RemoveDDFCache:
            CleanupPath = "%s*ddfcache" % MSName
        else:
            CleanupPath = None

        Command = "kMS.py %s --MSName=%s" % (BaseParset, MSName)
        PP.AppendCommand(
            "killMS_%s_%s" % (ThisNodeName, MSBaseName), Command,
             ↪  ParsetPath=BaseParset,
            NodeName=ThisNodeName, CheckFile=SolsFilename,
             ↪  CleanupPath=CleanupPath
        )

PP.WaitJob("killMS_*")
```

Listing 1: Main loop of killMS_parallel

```
print>> log, "Connecting to: %s" % NodeName
ssh.connect(NodeName)
atexit.register(ssh.close)

sftp = ssh.open_sftp()
sftp_mkdir(sftp, self.WorkDir)

# Copy parset file to node
if ParsetPath is not None and not sftp_path_exists(sftp, ParsetPath):
    ParsetDir = os.path.split(os.path.normpath(ParsetPath))[0]
    sftp_mkdir(sftp, ParsetDir)
    sftp.chdir(self.WorkDir)
    sftp.put(ParsetPath, ParsetPath)

sftp.close()

S = "cd '%s'; %s" % (self.WorkDir, Command)
print>> log, ModColor.Str("[%s] %s" % (NodeName, S), col="blue")

stdin, stdout, stderr = ssh.exec_command(S)
```

Listing 2: Connecting to a node, copying the configuration file and running the command

```
 # Wait for job to finish
print>> log, ModColor.Str("Waiting for %s..." % Job, col="blue")
STDOUT = self.JobPool[Job]["stdout"].read()
STDERR = self.JobPool[Job]["stderr"].read()
STDERROUT = STDOUT + STDERR

# Write job log to file
LogFilename = "%s_%s.log" % (Job, datetime.now().strftime("%Y%m%d-%H%M%S"))
LogFile = open(os.path.join(self.NodeLogDir, LogFilename), "w")
print>> LogFile, STDERROUT
LogFile.close()

# Report status
Condition0 = self.JobPool[Job]["stdin"].channel.recv_exit_status() != 0
Condition1 = "There was a problem after" in STDERROUT

if Condition0 or Condition1:
    print>> log, ModColor.Str("killMS has produced an error")
    print>> log, STDERROUT
    raise RuntimeError("killMS crashed")
else:
    print>> log, ModColor.Str("  Job %s has finished sucessfully" % Job,
     ↪  col="green")

# Transfer solutions to this node
SolsFilename = self.JobPool[Job]["CheckFile"]
sftp = self.JobPool[Job]["ssh"].open_sftp()
sftp.get(SolsFilename, SolsFilename)
sftp.close()

# Cleanup
CleanupPath = self.JobPool[Job]["CleanupPath"]
if CleanupPath is not None:
    print>> log, ModColor.Str("Cleaning up after command")
    s = "cd %s; rm -rf '%s'" % (self.WorkDir, CleanupPath)
    stdin, stdout, stderr = self.JobPool[Job]["ssh"].exec_command(s)
    # Make sure command is done, rm -rf will not print output
    stdout.read()

self.JobPool[Job]["ssh"].close()
print>> log, "   [done] %s" % Job
```

Listing 3: Waiting for a job to finish, and the work done afterwards

References

[1] Bal, H. et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *Computer* 49.5 (May 2016), pp. 54–63. ISSN: 0018-9162. DOI: 10.1109/MC.2016.127.

[2] Begeman, K. et al. "LOFAR information system". In: *Future generation computer systems* 27.3 (2011), pp. 319–328.

[3] Broekema, P. C. et al. "Cobalt: A GPU-based correlator and beamformer for LOFAR". In: *Astronomy and computing* 23 (2018), pp. 180–192.

[4] De Gasperin, F. et al. "The effect of the ionosphere on ultra-low-frequency radio-interferometric observations". In: *Astronomy & Astrophysics* 615 (2018), A179.

[5] Ellingson, S. W. "Antennas for the next generation of low-frequency radio telescopes". In: *IEEE Transactions on Antennas and Propagation* 53.8 (2005), pp. 2480–2489.

[6] *Factor: Facet Calibration for LOFAR*. 2019. URL: https://www.astron.nl/citt/facet-doc/index.html (visited on 07/15/2019).

[7] Gunst, A. W. and Schoonderbeek, G. *LOFAR Station - Architectural Design Document*. Tech. rep. ASTRON, 2007.

[8] Haarlem, M. P. van et al. "LOFAR: The LOw-Frequency ARray". In: *A&A* 556 (2013), A2. DOI: 10.1051/0004-6361/201220873. URL: https://doi.org/10.1051/0004-6361/201220873.

[9] Hardcastle, M. J. et al. "LOFAR/H-ATLAS: a deep low-frequency survey of the Herschel-ATLAS North Galactic Pole field". In: *Monthly Notices of the Royal Astronomical Society* 462.2 (July 2016), pp. 1910–1936. ISSN: 0035-8711. DOI: 10.1093/mnras/stw1763. eprint: http://oup.prod.sis.lan/mnras/article-pdf/462/2/1910/13774158/stw1763.pdf. URL: https://doi.org/10.1093/mnras/stw1763.

[10] Heald, G. et al. "Recent LOFAR imaging pipeline results". In: *ISKAF2010 Science Meeting*. Vol. 112. SISSA Medialab. 2010, p. 057.

[11] Intema, H. T. et al. "Ionospheric calibration of low frequency radio interferometric observations using the peeling scheme - I. Method description and first results". In: *A&A* 501.3 (2009), pp. 1185–1205. DOI: 10.1051/0004-6361/200811094. URL: https://doi.org/10.1051/0004-6361/200811094.

[12] Molenaar, G. and Smirnov, O. "KERN". In: *Astronomy and Computing* (2018).

[13] Rau, U. and Cornwell, T. J. "A multi-scale multi-frequency deconvolution algorithm for synthesis imaging in radio interferometry". In: *Astronomy & Astrophysics* 532 (2011), A71.

[14] Sabater, J. et al. "Calibration of LOFAR data on the cloud". In: *Astronomy and computing* 19 (2017), pp. 75–89.

[15] Shimwell, T. W. et al. "The LOFAR Two-metre Sky Survey - II. First data release". In: *A&A* 622 (2019), A1. DOI: 10.1051/0004-6361/201833559. URL: https://doi.org/10.1051/0004-6361/201833559.

[16] Smirnov, O. and Tasse, C. "Radio interferometric gain calibration as a complex optimization problem". In: *Monthly Notices of the Royal Astronomical Society* 449.3 (2015), pp. 2668–2684.

[17] Tasse, C. "Applying Wirtinger derivatives to the radio interferometry calibration problem". In: *arXiv e-prints*, arXiv:1410.8706 (Oct. 2014), arXiv:1410.8706. arXiv: 1410.8706 [astro-ph.IM].

[18] Tasse, C. et al. "Faceting for direction-dependent spectral deconvolution". In: *A&A* 611 (2018), A87. DOI: 10.1051/0004-6361/201731474. URL: https://doi.org/10.1051/0004-6361/201731474.

[19] Weeren, R. J. van et al. "LOFAR Facet Calibration". In: *The Astrophysical Journal Supplement Series* 223.1 (Mar. 2016), p. 2. DOI: 10.3847/0067-0049/223/1/2. URL: https://doi.org/10.3847%2F0067-0049%2F223%2F1%2F2.