P9 Towards distributed image reconstruction for radio interferometers

Jonas Schwammberger

December 9, 2019

Abstract

Naive distribution approaches are not efficient. They do not use the available resources, and get bogged down with communication. New approximation method. Way for easy parallel processing and later distribution.

Contents

1 The image reconstruction problem of radio interferometers **1**

2 Introduction to radio interferometric imaging **4**
2.1 Introduction to the ill-posed inverse problem . . . 4
2.2 Introduction to the MeerKAT observations . . . 6
2.2.1 Introduction to the third visibility term . . . 7
2.3 Introduction to image reconstruction algorithm for radio interferometer . . . 8
2.3.1 CLEAN deconvolution algorithm . . . 9
2.3.2 The Major/Minor cycle . . . 10

3 State of the Art image reconstruction **12**
3.1 w-stacking Gridder . . . 12
3.2 Image Domain Gridding Algorithm . . . 12
3.3 Deconvolution . . . 12
3.3.1 CLEAN . . . 12
3.3.2 MORESANE . . . 12
3.4 Coordinate Descent . . . 12

4 Coordinate descent deconvolution **13**
4.1 Elastic net regularization . . . 13
4.2 Serial coordinate descent algorithm . . . 15
4.2.1 Step 1: Choosing single a pixel . . . 15
4.2.2 Step 2: Optimizing a single pixel . . . 16
4.2.3 Inefficient implementation pseudo-code . . . 17
4.3 Efficient implementation . . . 18
4.3.1 Edge handling of the convolution . . . 18
4.3.2 Efficient calculation of the Lipschitz constants . . . 19
4.3.3 Using a map of gradients . . . 19
4.4 Efficient implementation pseudo-code . . . 21
4.5 GPU implementation . . . 22
4.6 Distributed implementaion MPI . . . 23
4.7 Serial coordinate descent and similarities to the CLEAN algorithm . . . 23

5 PSF approximation for parallel and distributed deconvolution **24**
5.1 Intuition for approximating the PSF . . . 24
5.2 Method 1: Approximate gradient update . . . 25
5.3 Method 2:Approximate deconvolution . . . 26
5.4 Major Cycle convergence and implicit path regularization . . . 27

6 Tests on MeerKAT LMC observation **29**
6.1 Comparison with CLEAN reconstructions . . . 30
6.2 Coordinate descent acceleration with MPI or GPU . . . 32
6.3 Effect of approximating the PSF . . . 32
6.3.1 Method 1: Approximate gradient update . . . 33
6.3.2 Method 2: Approximate deconvolution . . . 34
6.3.3 Combination of Method 1 and 2 . . . 35

7 Parallel coordinate descent methods **37**
7.1 From serial to parallel . . . 37
7.2 Parallel (Block) Coordinate Descent Method (PCDM) . . . 37

7.2.1 Block coordinate descent deconvolution . . . 37
7.2.2 Parallel block coordinate descent deconvolution . . . 38
7.3 Accelerated parallel block coordinate descent method . . . 40
7.4 Asynchronous implementation . . . 41
7.5 The problem with random selection for deconvolution . . . 41
7.5.1 Active set heuristic . . . 42
7.5.2 Restarting heuristic . . . 42
7.5.3 Re-introduction of a 'Minor' cycles . . . 44
7.6 Tests on the LMC Observation . . . 45
7.6.1 PSF approximation . . . 45
7.6.2 Block size . . . 46
7.6.3 Pseudo-random strategy . . . 47
7.6.4 Acceleration . . . 47
7.7 Comparison to the serial coordinate descent algorithm . . . 48
7.8 Scalability of the parallel coordinate descent algorithm . . . 49

8 Discussion **50**
8.1 Multi frequency extension . . . 50

9 Conclusion **51**

10 attachment **55**

11 Larger runtime costs for Compressed Sensing Reconstructions **56**
11.1 CLEAN: The Major Cycle Architecture . . . 57
11.2 Compressed Sensing Architecture . . . 57
11.3 Hypothesis for reducing costs of Compressed Sensing Algorithms . . . 58
11.4 State of the art: WSCLEAN Software Package . . . 58
11.4.1 W-Stacking Major Cycle . . . 58
11.4.2 Deconvolution Algorithms . . . 58
11.5 Distributing the Image Reconstruction . . . 58
11.5.1 Distributing the Non-uniform FFT . . . 58
11.5.2 Distributing the Deconvolution . . . 58

12 Handling the Data Volume **58**
12.1 Fully distributed imaging algorithm . . . 58

13 Image Reconstruction for Radio Interferometers **59**
13.1 Distributed Image Reconstruction . . . 59
13.2 First steps towards a distributed Algorithm . . . 60

14 Ehrlichkeitserklärung **61**

1 The image reconstruction problem of radio interferometers

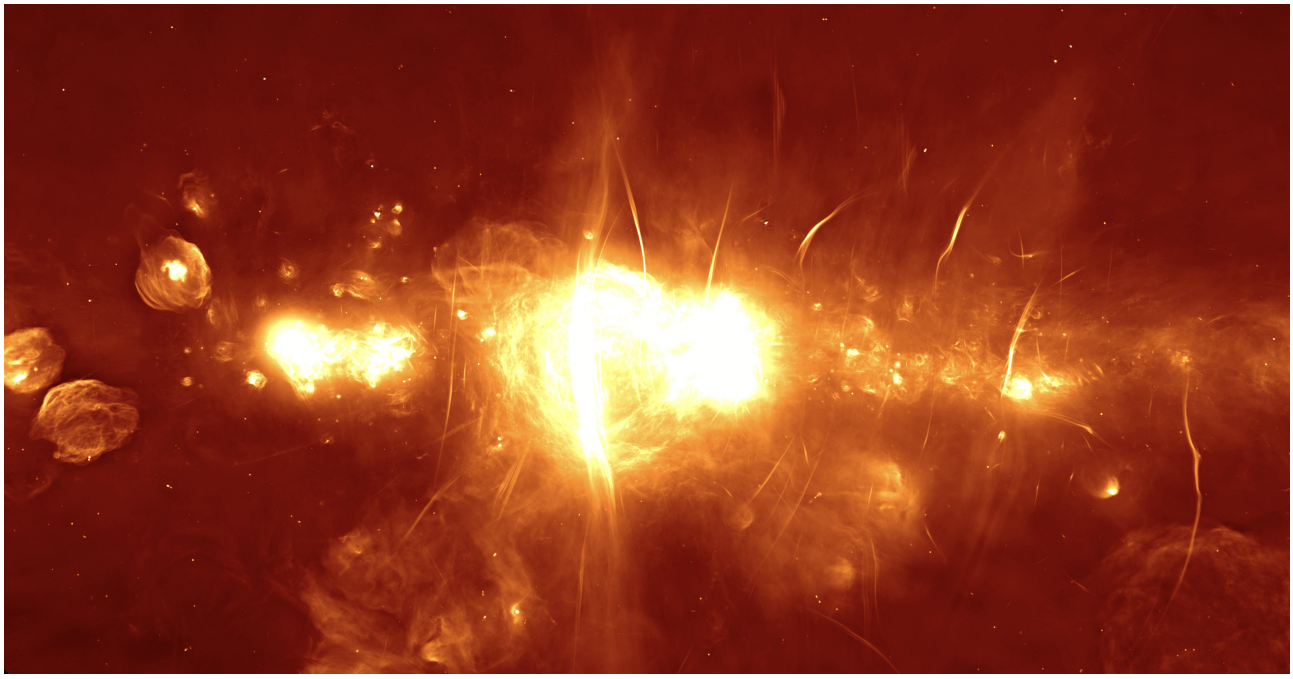


Figure 1: Center of the Milky Way galaxy at radio wavelengths. Observed by the MeerKAT interferometer.

Our observable universe emits a wide band of electromagnetic wavelengths. The spectrum visible to our eyes is only a narrow section of all the electromagnetic radiation in the universe. At different wavelengths, the same celestial objects reveal another picture. The center of the milky way galaxy is surrounded by dust clouds, making the center invisible to optical telescopes. The longer radio waves however pass right through the dust clouds, revealing the center of the milky way galaxy to the MeerKAT radio telescope in Figure 1.

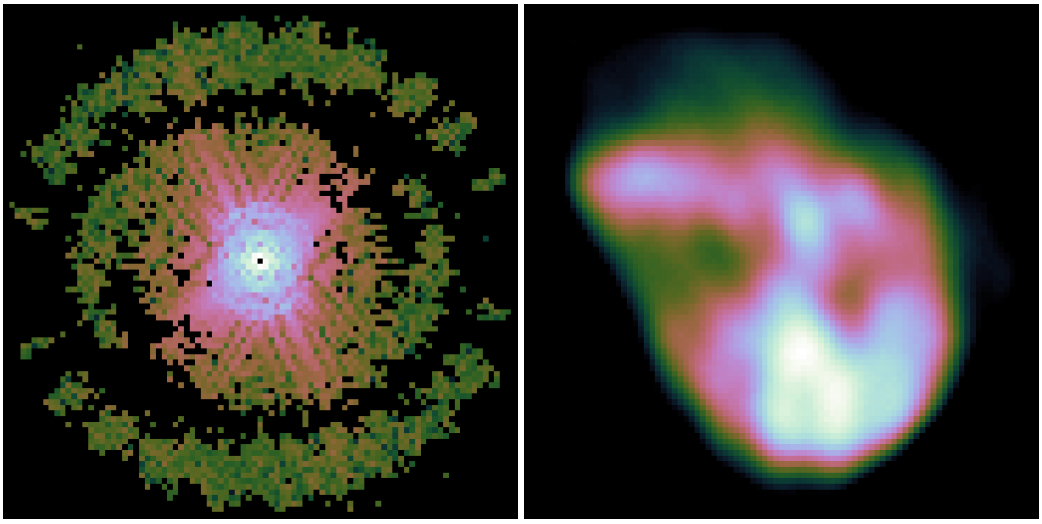
Radio telescopes are build with an ever increasing angular resolution. The higher resolution allows the instruments to detect ever smaller objects in the sky. But radio telescopes have a problem, which they do not share with their optical counterpart: We have already reached a practical resolution limit for single-dish telescopes. Angular resolution depends on the dish-diameter and the observed wavelength. With longer wavelengths, we need bigger dishes to achieve a similar angular resolution. The long radio wavelengths require huge dishes for a similar angular resolution to optical telescopes.

There is a practical limit on the antenna-dish diameter we can build. The famous Arecibo observatory is one of the largest single-dish radio telescopes with a diameter of 305 meters. Antennas with such a large diameter become difficult to steer accurately, let alone the construct economically. If we require higher angular resolution, we need to look at another type of instrument: The radio interferometer. They use several smaller antennas together, acting as a single large dish. An interferometer can achieve angular resolutions which are comparable to dishes with a diameter of several kilometers.

But there are drawbacks: The interferometer does not measure the sky in pixels, but in the Fourier space. It measures the amplitude and phase of a set of Fourier components. The image has to be calculated from the Fourier measurements. This is the image reconstruction problem of radio interferometers. The measured Fourier components are called visibilities in the radio astronomy literature. From here on forward, we will call the measured Fourier components visibilities.

The Figure 2 shows an example of the image reconstruction problem. The radio interferometer measures a set of visibilities (Fourier components), shown in Figure 2a. The set of visibility measurements is incomplete, meaning the interferometer has not measured all the relevant information for the image reconstruction. The

missing visibilities are shown as black dots in Figure 2a. The image reconstruction is tasked to find the observed image, shown in Figure 2b, from the incomplete set of visibilities.



(a) Measured visibilities (Fourier components) of the N132 supernova-remnant.

(b) Observed image of the N132 supernova-remnant.

Figure 2: Example of an image reconstruction problem of radio interferometers

Furthermore the visibility measurements are corrupted by noise. The atmosphere of the earth can change the true amplitude and phase of each measured visibility. Under unfavorable conditions, the atmosphere can introduce a high level of noise when compared to the signal.

These two difficulties, the noise and the incomplete measurements, lead to the fact that there are many different candidate images that fit any measurements. This is known as an ill-posed inverse problem. We want to find the observed image, even though all we have are an incomplete set of noisy visibility measurements. From the measurements alone, we cannot decide which candidate is the truly observed image. However, we have additional knowledge that simplifies the inverse problem: We know it is an image of the celestial objects. The image consists of stars, hydrogen clouds, and other celestial objects which emit radio waves. By including prior knowledge in the reconstruction, we can find the most likely image given the measurements.

The question remains is: How close is the most likely image to the observed one? Is exact reconstruction possible where the most likely and observed image are equal? Surprisingly the answer is yes. It is possible in theory[1, 2], and was shown in practice on low noise measurements[3, 4]. However, not all algorithms perform equally well when the noise level in the measurements is high. Also, computing resources required for each algorithm can vary significantly. In short, a reconstruction algorithm has three opposing goals:

1. Produce a reconstruction which is as close to the truly observed image as possible.
2. Robust against even heavy noise in the measurements.
3. Use as few computing resources as possible.

No reconstruction algorithm performs equally well on all three goals. One of the most widely used reconstruction algorithms is CLEAN [5, 6]. It has shown to be robust against heavy noise[7] and, depending on the observation and is one of the oldest algorithms still in use today. As such, it was developed before the advent of distributed and GPU-accelerated computing. Today's new radio interferometers produce ever more measurements. The recently finished MeerKAT radio interferometer produces roughly 80 million Fourier measurements each second. Astronomers wish to reconstruct an image from several hours worth of measurement data. Reconstructing an image from this data volume requires GPU and distributed computation. But how to use GPU and distributed computing effectively is still an open problem.

Coordinate descent methods have been successfully applied in other inverse problems, such as reconstruction of CT scans[8], or X-Ray imaging[9]. GPU accelerated[10] and distributed[11] variants have been developed. To our knowledge, coordinate descent methods have not been explored for the inverse problem in Radio Astronomy.

In this work, we develop our own proof-of-concept image reconstruction algorithm based on coordinate descent methods. We apply the reconstruction on a real world MeerKAT observation provided by SARAO. We explore the possible speedups we can achieve by using GPU and distributed computation. The algorithm is implemented platform independent in .netcore.

The rest of this work is structured as follows. First in section 2, we give an introduction to radio interferometric imaging, and give the theoretical background to why a reconstruction can even achieve a higher resolution than the instrument. Next we present the current state-of-the-art in image reconstruction for radio interferometers in section 3. Then we derive a basic image reconstruction algorithm based on coordinate descent in section 4, and show how we can use GPU acceleration and distribution to speed up reconstruction.

2 Introduction to radio interferometric imaging

In radio astronomy, there are two classes of radio sources: Point sources and extended emissions. Point sources are typically far-away stars, their emission is concentrated around a single pixel in the observed image. Extended emissions are objects that span a large area of the sky, like dust or hydrogen clouds. They span an area over several pixels. Both classes emit radio waves. The radio waves travel to earth, may get distorted by Earths Ionosphere, and finally arrive at the interferometer. Figure 3 shows the radio interferometry system.

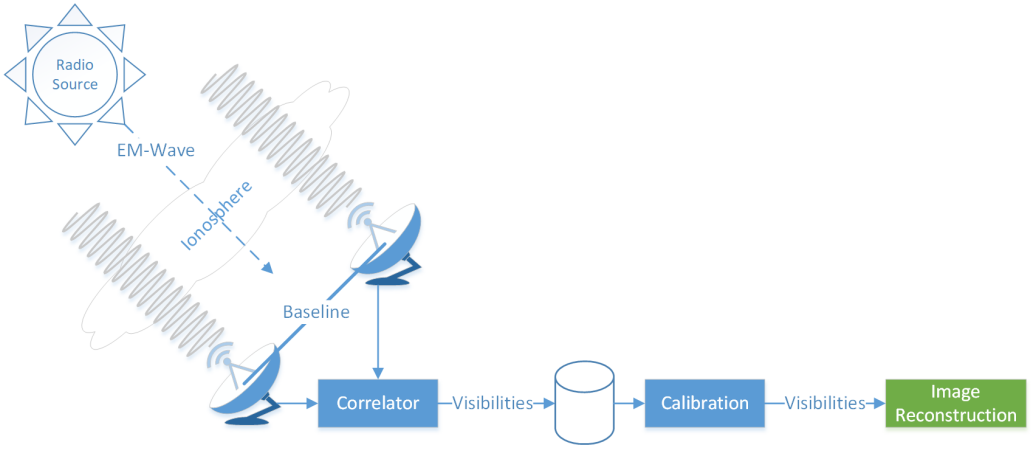


Figure 3: Radio interferometry system

The electromagnetic waves arrive at each antenna of the interferometer. The Correlator then takes the measurement of each antenna pair. Each antenna pair is called a baseline. The correlator creates a complex visibility measurement for each baseline, consisting of amplitude and phase. After this step, the visibilities are saved for later reconstruction.

Before the visibilities can be reconstructed into an image, they need to be calibrated. Each antenna has a different sensitivity which drifts over time. Also, the distortion by Earths Ionosphere can be approximated and removed, depending on the observation.

The calibration process is imperfect. The visibility measurements are still noisy, meaning their true amplitude and phase may differ from what was measured. The image reconstruction step is tasked to find the most likely image from noisy and incomplete visibilities.

This project is focused on the image reconstruction step. We will develop our own image reconstruction algorithm that finds the most-likely image given the calibrated visibility measurements. However, the specifics of radio interferometric measurements influence every part of the image reconstruction algorithm. Efficient reconstruction algorithms have to deal with the specifics of radio interferometry.

This section first introduces how image reconstruction works in theory. Then, we introduce the MeerKAT radio interferometer, for which we have real-world visibility measurements. We show the difficulties the MeerKAT interferometer introduces for the image reconstruction. Finally, we show how real-world image reconstruction algorithms work to deal with MeerKAT data, and show where our contributions lie.

2.1 Introduction to the ill-posed inverse problem

Radio interferometers like MeerKAT generally produce more visibility measurements than pixels in the reconstruction. Still, the measurements are incomplete, and from the measurements alone, we cannot reconstruct the observed image. In the literature this is known as an ill-posed inverse problem. We wish to find the observed image from measurements in the Fourier space, but the measurements alone do not contain all the

relevant information.

However, we can include prior information about the image, we can reconstruct the most-likely image given the measurements. Here we introduce how a reconstruction can find the most-likely image given the measurements. First, we formulate the image reconstruction as a minimization problem with the following objective function:

$$\underset{x}{minimize} \ \|V - MFx\|_2^2 \tag{2.1}$$

Where x is the reconstructed image we wish to find, V are the measured visibilities, F is the Fourier transform matrix and M is the masking matrix. The masking matrix sets the visibilities that were are missing in the image.

We can reconstruct the image by finding the optimum of the objective function (2.1). The objective function is convex, meaning it has only one global minimum, and we can use the class of convex optimization algorithms to search the minimum. However, our measurements V are incomplete, meaning we do not have all the data we need for reconstruction. This means our objective function (2.1) does not "point" to the observed image. It still has a global minimum, but observed image is not guaranteed to be near the global minimum.

A side note: We are guaranteed to find the observed image at the minimum of (2.1) is when the measurements fulfill the Nyquist-Shannon sampling theorem. In that case, we can find the minimum by calculating the inverse Fourier transform: $x = F^{-1}V$. We can still calculate the inverse Fourier transform when we are dealing with incomplete measurements, but it does not result in the observed image.

The objective (2.1) only includes information about the measurements. As we have mentioned before, we have prior knowledge about the image. We know for exampe it is likely to contain point sources. In that case, most pixels of the image will be zero, except for the locations where the interferometer has located the point sources. In other words, we know that the image is sparse. We can add a regularization to the objective function (2.1) and force the reconstructed image to be sparse. This results in the modified objective function:

$$\underset{x}{minimize} \ \|V - MFx\|_2^2 + \lambda \|x\|_1 \tag{2.2}$$

Note the two terms in the objective (2.2): We have the same term from our measurements, which we call the "data term". But we also have an additional "regularization term", which is the L1 norm[1] and forces our reconstruction to be sparse. The parameter λ represents how much emphasis we put on the regularization. The new objective function is still convex, it still has a global minimum. The regularization term simply shifted the global minimum to a different location when compared to the first objective (2.1). Now the question is: Does it point to the observed image? In practice, if the regularization is a good model for the image content, the image we retrieve at the minimum of the objective (2.2) is indistinguishable from the observed image.

The reconstruction quality also depends on how well the regularization models the observed image. In our example the L1 norm is a good model for images only consisting of stars. It penalizes images that have a large number of non-zero pixels. But radio interferometer also measure extended emissions like gas-clouds. Those images lead to a large number of non-zero pixels and by extent a large regularization penalty. The L1 regularization may force non-zero pixels of the extended emission to zero.

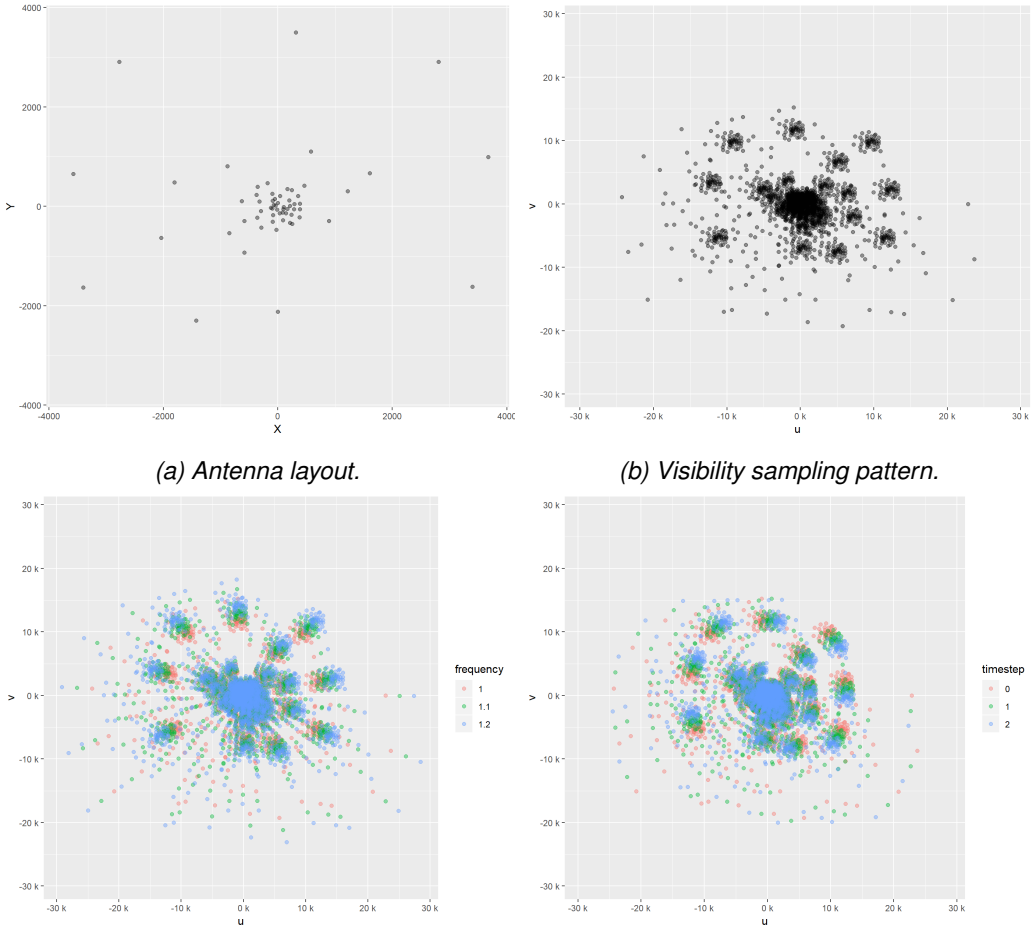
In a sense there is data modeling task involved in the image reconstruction. The better we understand what a plausible image looks like, the higher the reconstruction quality. Recent work managed to reconstruct super-resolved images for the VLA interferometer[3], where the reconstruction achieved a resolution higher than the limit of the instrument itself.

[1] Sum of absolute values of the pixels, $\sum_i \sum_j \|x_{ij}\|$

The image reconstruction problem is ill-posed, but with the right regularization we can reconstruct an image which even surpasses the accuracy limit of the instrument. However, the regularization also influences other properties of the reconstruction algorithm, such as the wall-clock time for a single reconstruction, or its behavior with very noisy visibilities. The question of the optimal regularization is, as of the time of writing, still open in radio astronomy.

2.2 Introduction to the MeerKAT observations

So far, we looked at how image reconstruction works in theory. We need a regularization to model our prior knowledge, that the image is a mixture of point sources and extended emissions. Then, all we need is a convex optimization algorithm, which finds us the most likely image. Real-world radio interferometer introduce complexities which have to be handled by the reconstruction algorithm. First, there is a large number of visibility measurements. Any operation on the measurements is potentially time consuming. Secondly, the measurements of modern radio interferometers become more complicated, as we will show in the next section. Here, we show how MeerKAT can produce an almost arbitrary large amount of visibility measurements.



(a) Antenna layout.

(b) Visibility sampling pattern.

(c) Visibilities added from multiple channels.

(d) Visibilities added from multiple timesteps.

Figure 4: Sampling regime of the MeerKAT radio interferometer.

The MeerKAT radio interferometer consists of 64 antennas. Each antenna pair creates a single baseline, and each baseline results in a single visibility. This results in 2016 visibilities for the MeerKAT interferometer. The distance between the antennas of each pair is important: The distance defines what point in the Fourier space gets sampled. A short baseline samples close to the origin of the Fourier space, and contain low-frequency Fourier components of the sky image. Long baselines measure points further away from the origin. They

sample the high-frequency Fourier components, and contain information about sharp edges in the image. Figure 4a shows the 64 antennas of MeerKAT, while the Figure 4b shows the 2016 visibility samples in the Fourier space.

The sampling pattern of the MeerKAT interferometer is not uniform in the Fourier space. We have areas which are densely sampled, and areas which are sparsely sampled. Note that we only have a few samples of the high-frequency Fourier components. We are missing measurements from a large portion of the Fourier space. Modern radio interferometers use two tricks to fill up the Fourier space with more visibility samples: Adding visiblities from different channels, and from different timesteps.

The interferometer measures the sky at different radio channels simultaneously. We can add the visibilities from neighboring channels together, resulting in the visibility pattern shown in Figure 4c. Each channel measures the Fourier space using the same pattern, but scaled by the radio frequency.

The interferometer observes the sky over a period of time. Depending on the observation, this can be minutes, up to several hours. During this time, the rotation of the Earth rotates the sampling pattern in Fourier space, shown in Figure 4d.

The MeerKAT radio interferometer measures 2016 visibilities, for each channel at each timestep. It has 20 thousand channels, and the time resolution can be as low as half a second. This results in roughly 80 million visibility measurements for each second of observation. A real-world observation can span over several hours, and take up several terabytes of disk space. The data volume alone creates difficulties in any practical reconstruction algorithm. The large number of visibility measurements is difficult to keep in memory. Any operation on the whole set of visibilities becomes impractical.

But MeerKAT introduces further complications apart from the large data volume. Adding the visibilities from 20 thousand channels together becomes inaccurate. For an accurate reconstruction, the reconstruction algorithm has to perform wide-band imaging: Instead of reconstructing a single image containing the visibilities from all frequencies, the algorithm reconstructs an image cube, where each image represents a single, or a small fraction of all channels. Wide-band imaging is out of scope for our project, and our reconstruction algorithm only considers narrow-band imaging.

Furthermore, the reconstructed image from MeerKAT observations generally span a wide field-of-view. For an accurate reconstruction, the algorithm has to account for the fact that the visibilities are three dimensional. We include wide field-of-view imaging in our reconstruction algorithm, and discuss the effects of the third visibility term in more detail.

2.2.1 Introduction to the third visibility term

As we discussed so far, the radio interferometer measures visibilities of the sky image, and we wish to find the observed image from the measurements. We have ignored the fact that the visibility measurements not only have a $u-$ and v-, but also a third w-term. Visibilities are in fact three dimensional.

The third w component arises from the fact that the baselines (antenna pairs) are not on the same plane. The w-term becomes relevant for wide field-of-view imaging in radio interferometry. We will show in later sections how the w-term affects the design of the image reconstruction algorithm. Let us start with the measurement equation for small field-of-views:

$$V(u,v) = \int\int I(l,m) e^{2\pi i[ul+vm]}\, dl\, dm \tag{2.3}$$

The radio interferometer measures Visibilities V from the sky image I. Each visibility is a measurement over all pixels l, m. The term $e^{2\pi i[ul+vm]}$ is simply the Fourier transform. l and m are the directions from which we measured radiation. They can be thought of as the pixel indices of the image. Index $(0,0)$ is the center

pixel of the image. So far the visibilities are two dimensional. But this measurement equation is only valid for a small field-of-view. For a large field-of-view the measurement equation expands to:

$$V(u,v,w) = \int\int \frac{I(l,m)}{c(l,m)} e^{2\pi i[ul+vm+w(c(l,m)-1)]}\, dl\, dm\,, \quad c(l,m) = \sqrt{1-l^2-m^2} \tag{2.4}$$

We still have the Fourier transform $e^{2\pi i[ul+vm...]}$ at the core. But now the visibilities have a third w-term, the image has a normalization factor of $c(l,m)$ and the Fourier transform has a phase-shift term $+w(c(l,m)-1)$ added to it. The phase-shift of the Fourier transform depends on the direction from which the radiation was measured. As such, the phase shift changes over the image. It is called a Directionally Dependent Effect (DDE).

There exist several sources of DDE's. The Ionoshpere for example is a natural source that distorts the signal depending on the direction. From different directions, the distortion by the Ionosphere changes. The w-term arises from the properties of the interferometer itself. DDE's are costly in terms of computation time. The w-term breaks the two dimensional Fourier relationship between the image and the visibilities. We cannot use the FFT, resulting in a more expensive Fourier transformation. The difficulty lies in handling DDE's efficiently in the image reconstruction algorithm.

2.3 Introduction to image reconstruction algorithm for radio interferometer

This section introduces the commonly used architecture for radio interferometric image reconstruction, the Major/Minor cycle. It shows how DDE's like the w-term are handled, and introduces the basic reconstruction algorithm for radio astronomy: CLEAN.

First, let us look back at the objective function. In the previous section, the objective function (2.2) contained the visiblity measurements V in Fourier space, the reconstructed image x in image space, and the Fourier transform matrix F that represents the relationship between image and Fourier space. However, this is only one way to formulate the reconstruction problem:

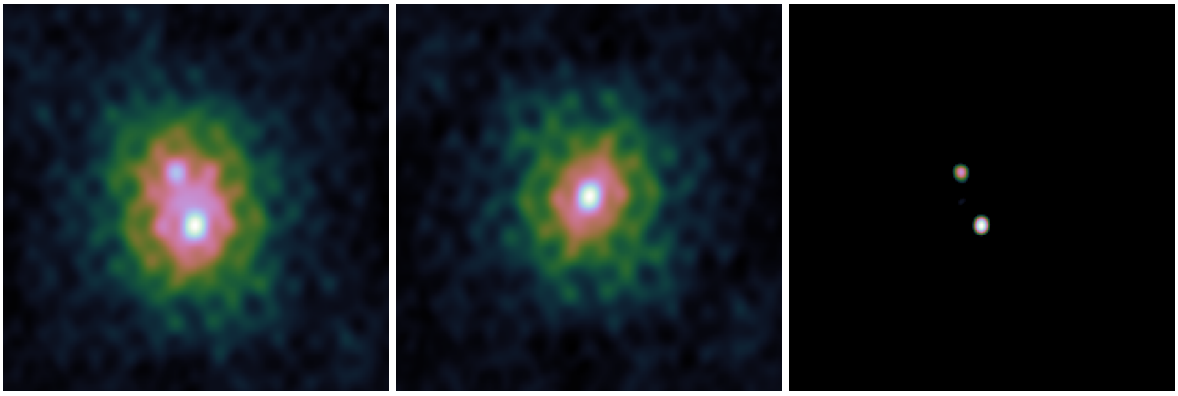
$$\begin{aligned}
&\underset{x}{minimize}\ \|V - MFx\|_2^2 + \lambda \|x\|_1 \\
&\underset{V_2}{minimize}\ \|V - MV_2\|_2^2 + \lambda \left\|F^{-1}V_2\right\|_1 \\
&\underset{x}{minimize}\ \|I_{dirty} - x * PSF\|_2^2 + \lambda \|x\|_1
\end{aligned} \tag{2.5}$$

We can also reconstruct the missing visibilities directly. In that case, we solve an in-painting problem in the Fourier space. Or we can transform the measurements into image space, and solve an equivalent deconvolution problem, where the masking matrix M in Fourier space becomes the convolution kernel called the Point Spread Function PSF (Remember that a multiplication in Fourier space is a convolution in image space.)

These three problems are equivalent in theory. The key difference between the three formulations is that the we generally have magnitudes fewer pixels than visibility measurements in the reconstruction problem. The deconvolution problem therefore consumes less memory.

Image reconstruction algorithms for radio interferometers generally use the deconvolution formulation. They first transform the visibility measurements to an image, called the dirty image. Then, they transform the masking matrix M into the PSF. At this point, the algorithms reconstruct the image by calculating a deconvolution. Figure 5 shows a simulated example of a dirt image, PSF and deconvolved image.

In radio astronomy, image reconstruction is split into two separate algorithms. The first algorithm is responsible for transforming the measurements from Fourier to image space. The second algorithm is responsible for



(a) Dirty Image. *(b) Point Spread Function.* *(c) Deconvolution*

Figure 5: Example deconvolution problem with two point sources.

deconvolution. The first algorithm is referred as the Major cycle in radio astronomy, and the deconvolution algorithm as the minor cycle in radio astronomy literature. The reason why they are referred to as ”cycles” will become clear in Section 2.3.2. In short: The whole process, transforming the measurements to image space and calculating the deconvolution has to be repeated several times to correct for DDE's like the w-term. First, we will take a closer look to the most commonly used deconvolution algorithm, CLEAN.

2.3.1 CLEAN deconvolution algorithm

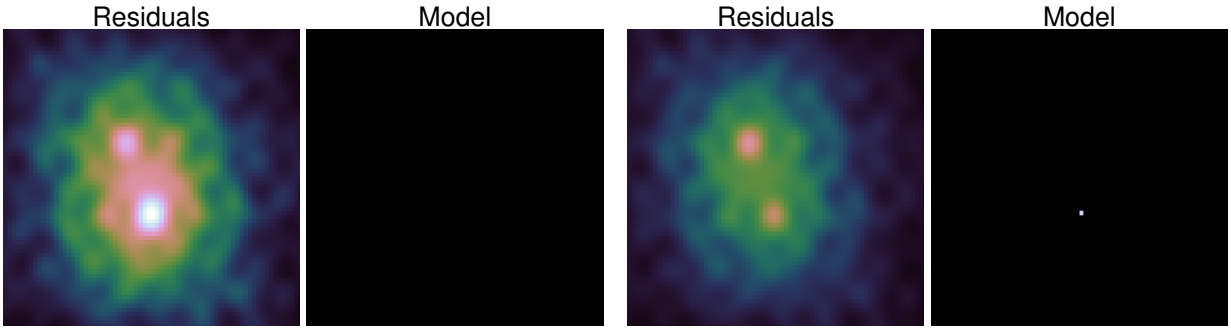
CLEAN[5] is the basic deconvolution algorithm in radio astronomy. It has been extended over the years, and in its more sophisticated form as multi-scale multi-frequency CLEAN [7] is still used today. We introduce the basic CLEAN algorithm in this section.

The CLEAN deconvolution algorithm keeps three images in memory: The residual image, the model image, and the PSF. In general, all three images are the same size. At the start of the algorithm, the residual image is equal to the dirty image shown in Figure 5a. The model image starts with every pixel set to zero. It is updated in each iteration and will, at the end, contain the deconvolved version of the dirty image.

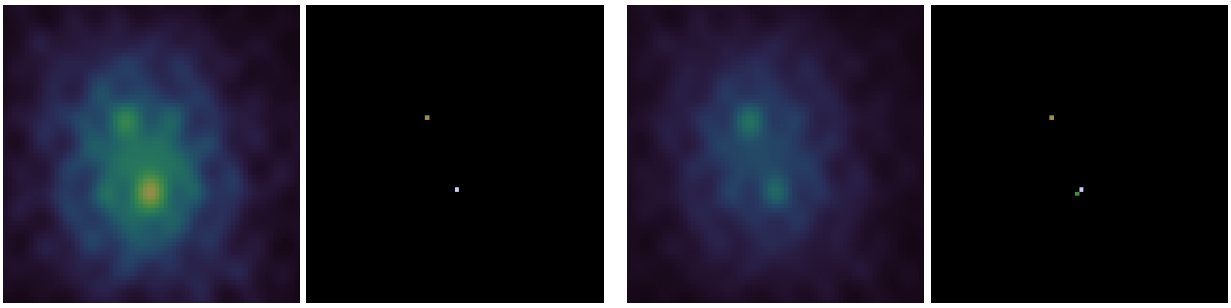
A CLEAN iteration begins with searching the maximum pixel in the residual image. In the next step, it adds a fraction of the maximum value to the model image. For example, if the maximum pixel is 2.7, CLEAN adds 1.35 to the model image at the same location (This fraction is called the CLEAN gain and is one of the parameters left for the user to define). The last step is to subtract the PSF at the location. The Figure 6 shows the residual and the model image over four iterations.

After a number of CLEAN iterations, the model image contains a number of non-zero pixels. But CLEAN also leaves artifacts in the model image. As we see in Figure 6d, iteration 3 adds another non-zero pixel close to the already detected one. CLEAN brushes over the artifacts by simply blurring the model image with the ”CLEAN-beam” after CLEAN has finished its iterations. The CLEAN-beam represents the accuracy limit of the radio interferometer. By blurring the image, CLEAN reconstructs the image at the limit of the instrument. The blurring with the CLEAN-beam is shown in Figure 7

The CLEAN algorithm assumes the image contains point sources. In each iteration, it essentially adds the most-likely point source. It is a greedy algorithm, because it searches the most-likely point source in each iteration. It approximately minimizes the deconvolution objective of (2.5), and uses a similar regularization to the L1 norm. The regularization in CLEAN is implicit in the algorithm. In practice, CLEAN implementations use several heuristics, which makes an exact analysis of the regularization difficult.



(a) Iteration 0 *(b) Iteration 1*



(c) Iteration 2 *(d) Iteration 3*

Figure 6: CLEAN deconvolution iterations.

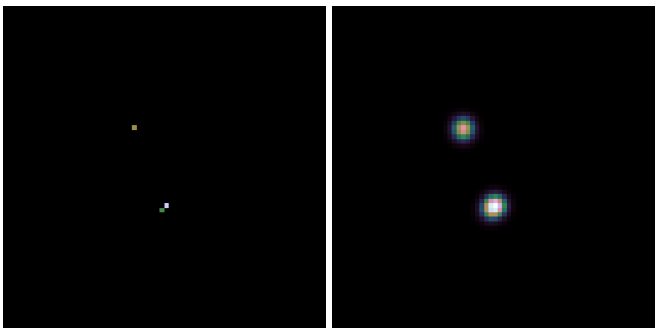


Figure 7: Blurring with the CLEAN-beam

2.3.2 The Major/Minor cycle

During CLEAN deconvolutions, the algorithm assumes that the PSF is constant over the image. However, this is not the case for wide field-of-view observations. The w-term changes the PSF depending on the location in the image. CLEAN deconvolutions only approximate the true PSF. With more CLEAN iterations the residuals image becomes less accurate. To solve this issue, a deconvolution algorithm is used inside the Major/Minor cycle framework.

The Minor cycle is responsible for deconvolution. Typically, a CLEAN based algorithm is used. Every CLEAN iteration is then called a ”Minor” cycle. After a number of CLEAN iterations, the residual image is too inaccurate. Then, the Major cycle takes the model image of CLEAN, and transforms it back to the original Fourier space, called the ”model visibilities. It then subtracts the model visibilities from the measured visibilities, and transforms the residual visibilities back to the image space. Figure 8 shows the Major/Minor cycle framework in more detail.

The major cycle has updated the residuals, and the deconvolution algorithm can continue. The Major cycle allows us to use an approximate PSF in the deconvolution algorithm. Typically, we need several thousand

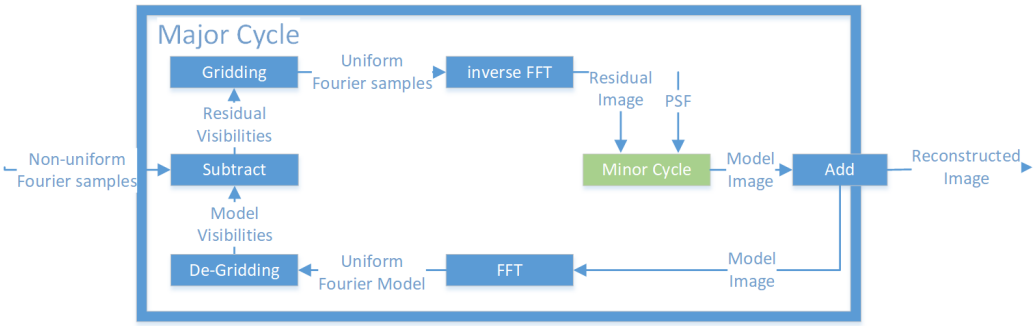


Figure 8: Major/Minor cycle

Minor cycles, and between three to ten major cycles to reconstruct an image. The exact numbers depend on the observation itself.

We already discussed the Minor cycle algorithm in detail, now let us turn to the Major cycle algorithm. The Major cycle uses two steps to transform the visibilities to image space. It first takes the visibilities and interpolates them on a uniform Fourier grid. This step is called ”Gridding”. Next, the inverse Fast Fourier Transform (FFT) is applied, resulting in the residual image. The combination of gridding and FFT is faster than any direct transformation from visibility to image space. Note that in this architecture the gridding step has to handle the w-term.

The gridding step and the Minor Cycles are the two bottlenecks in the reconstruction. The gridding step has to interpolate the visibility measurements, which generally has magnitudes more measurements than pixels, and correct for the w-term. The Minor Cycle can become more expensive if the image contains a large number of sources. Depending on the image, it can even become more expensive than the gridding step [7].

The gridding and the Minor cycle deconvolution algorithm are an active field of research in radio astronomy. Recently, the Image Domain Gridder (IDG)[12] has been developed. It uses GPU-acceleration to speed up the gridding step, shifting the bottleneck to the Minor cycle.

Our project focuses on the Minor cycle deconvolution algorithm. The Major cycle allows us to use an approximate PSF in the deconvolution. Our hypothesis is we can approximate the PSF further to simplify the deconvolution problem, and facilitate a deconvolution algorithm in the Minor cycle that uses distributed and GPU-accelerated computing.

3 State of the Art image reconstruction

Image reconstruction pipelines are split in two taks. Gridding and deconvolution. We introduce here the two latest gridding algorithms, w-stacking in 3.1 and Image Domain Gridder 3.2.

Deconvolution we discuss CLEAN and MORESANE.

Not all algorithms are here from

3.1 w-stacking Gridder

3.2 Image Domain Gridding Algorithm

3.3 Deconvolution

3.3.1 CLEAN

Various Improvements and speedups[6, 7, 13, 14], but the core algorithm is still this.

3.3.2 MORESANE

3.4 Coordinate Descent

4 Coordinate descent deconvolution

In this section introduces the coordinate descent methods. We derive a coordinate descent deconvolution algorithm which replaces CLEAN in the Major/Minor cycle architecture.

Coordinate descent is a group of numerical minimization algorithm. In each iteration, they minimize the problem along a single coordinate. In our deconvolution problem, a coordinate descent method minimizes the objective function along a single pixel in each iteration. Over several iterations, possibly minimizing the same pixel multiple times, the coordinate descent method converges to a deconvolved image.

We call the algorithm derived in this section 'serial coordinate descent'. In each iteration, the serial coordinate descent algorithm minimizes exactly one pixel. Although it is called serial, we can use multiple processors for minimizing a single pixel. In Section 7 we introduce more sophisticated parallel coordinate descent deconvolution algorithm. Parallel coordinate descent methods update multiple pixels in parallel in each iteration.

As we mentioned before, a deconvolution algorithm in radio astronomy has three components: An objective function, a regularization and a numerical optimization algorithm, which is based on coordinate descent. The objective function we use for our method is:

$$\underset{x}{minimize}\ \frac{1}{2}\left\|I_{dirty} - x * PSF\right\|_2^2 + \lambda ElasticNet(x) \tag{4.1}$$

We want to find the minimum deconvolved image x, which is as close to the dirty image (remember the dirty image results from the visibilities, which get gridded and transformed to image space), but also has the smallest regularization penalty. The parameter λ is a weight that either forces more or less regularization. It is left to the user to define λ for each image.

We introduce the elastic net regularization in the next Section, and then continue with the serial coordinate descent deconvolution algorithm in Section 4.2. We then go over the efficient CPU, GPU and distributed implementations of the serial coordinate descent algorithm.

4.1 Elastic net regularization

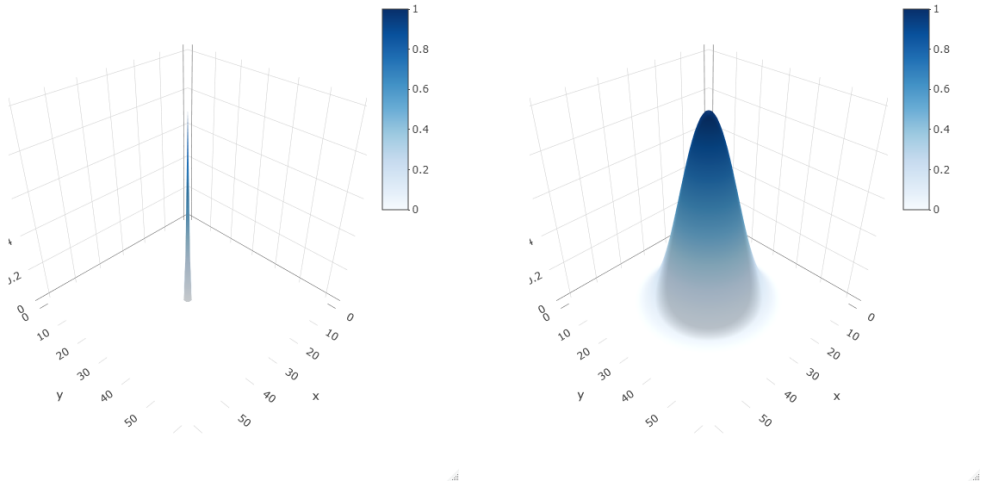
This regularization is a mixture between the L1 and L2 regularization. The L1 regularization is the absolute value of all pixels ($\|x\| = \sum_i \sum_j \|x_{ij}\|$), and the L2 norm is the squared sum of all pixels ($\|x\|_2 = \sum_i \sum_j \|x_{ij}\|^2$). he Figure 9 shows the effect of the L1 and L2 norm on a single star. The L1 regularization forces the image to contain few non-zero pixels as possible. It encodes our prior knowledge that the image will contain stars. The L2 regularization on the other hand "spreads" the single star across multiple pixels.

The L1 regularization alone models an image that consists of points sources. For extended emissions like hydrogen clouds, the L1 regularization often leads the reconstructed image to become a cluster of point sources, instead of a real extended emission.

The L2 norm alone was already used in other image reconstruction algorithms in radio astronomy[15], with the downside that the resulting image will not be sparse. I.e. all pixels in the reconstruction will be non-zero, even though all they contain is noise.

Elastic net mixes the L1 and L2 norm together, becoming "sparsifying L2 norm". It retains the sparsifying property of the L1 norm, while also keeping extended emissions in the image. Formally, elastic net regularization is defined as the following regularization function:

$$ElasticNet(x, \alpha) = \alpha \left\|x\right\|_1 + \frac{1-\alpha}{2}\left\|x\right\|_2 \tag{4.2}$$



(a) Effect of the pure L1 norm (λ = 1.0) on a single point source.

(b) Effect of the pure L2 norm (λ = 1.0) on a single point source.

Figure 9: Effect of the L1 and L2 Norm separately.

The parameter α is between 0 and 1, and mixes the two norms together. A value of 1 leads to L1 regularization only, and a value of 0 leads to L2 only. The elastic net regularization has two properties, which are relevant later for the serial coordinate descent deconvolution: It is separable, and has a proximal operator.

Separability means that we can calculate the elastic net regularization penalty independently for each pixel. We arrive at the same result if we evaluate (4.2) for each pixel and sum up the results, or if we evaluate (4.2) for the whole image. This is an important property when one tries to minimize the the elastic net penalty (as we will with the serial coordinate descent deconvolution algorithm). We can also calculate how much any pixel change reduces the elastic net penalty independently its neighbors.

The proximal operator of elastic net allows us to minimize the regularization penalty. Notice that the elastic net regularization (4.2) is not differentiable (the L1 norm is not continuous). We cannot calculate a gradient, and cannot use methods like gradient descent to minimize the regularization penalty. However, it has a proximal operator defined:

$$ElasticNetProximal(x, \lambda, \alpha) = \frac{max(x - \lambda\alpha, 0)}{1 + \lambda(1 - \alpha)} \tag{4.3}$$

In our deconvolution problem, we can apply the proximal operator (4.3) on each pixel, and we minimize the elastic net penalty. Again, the proximal operator can be applied on each pixel independently, as neighboring pixels do not influence its result.

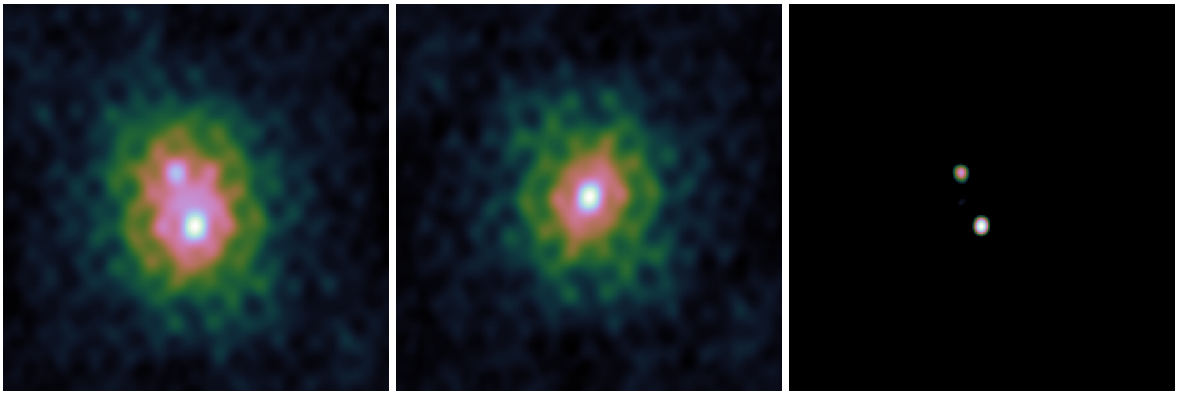
The elastic net regularization is separable. We can calculate its penalty for each pixel independently of its neighbors. As such, its proximal operator is also independent of the neighbors. It is the only regularization we use in this project. We now derive a coordinate descent based deconvolution algorithm that uses the proximal operator to efficiently reconstruct an image.

A side note on the proximal operator used in this project (4.3): The numerator always clamps negative pixels to zero. This is a conscious design decision. In radio astronomy, it is usual to constrain the reconstruction to be non-negative (because we cannot receive negative radio emissions from any direction). It is widely used in radio astronomy image reconstruction and may lead to improved reconstruction quality [16].

4.2 Serial coordinate descent algorithm

Our serial coordinate descent deconvolution algorithm minimizes the deconvolution objective (8.1). It is a convex optimization algorithm that optimizes a single pixel (coordinate) at each iteration. Each iteration consists of two steps. Step 1: Find the best pixel to optimize. Step 2: Calculate the gradient for this pixel, take a descent step and apply the elastic net proximal operator. We repeat these steps in each iteration until coordinate descent converges to a solution.

We demonstrate the serial deconvolution algorithm with the help of a simulated MeerKAT reconstruction problem of two point sources. Figure 10a shows the dirty image of two point sources, and Figure 10b the PSF. The deconvolved image with elastic net regularization is shown in Figure 10c. I.e. Figure 10c is the optimum x of the objective function (8.1).



(a) Dirty Image. *(b) Point Spread Function.* *(c) Elastic net reconstruction*

Figure 10: Example problem with two point sources.

Note that our algorithm calculates the gradient for the current pixel. This may raise the question: What exactly is the difference between gradient- and coordinate descent is that gradient descent optimizes all pixels in each iteration, while coordinate descent optimizes (generally) a single pixel at a time[2]. Also, Coordinate descent methods are not bound to use the gradient. It could use a line search approach, where we try different values and decide on the one leading to the lowest objective value.

Because coordinate descent methods only optimize a single pixel at a time, they generally need a large number of iterations to converge compared to other methods. But when each iteration is cheap to compute, coordinate descent methods can converge to a result in less time than competing methods[17, 18]. We discuss the efficient implementation in Section 4.3. First, we explain each step in the serial coordinate descent algorithm in more detail.

4.2.1 Step 1: Choosing single a pixel

Our serial coordinate descent algorithm uses a greedy strategy. From all possible pixels, it searches the pixel whose gradient has the largest magnitude. In each iteration, it chooses the pixel which reduces the objective function the most. There are two other strategies that are used in coordinate descent: Random and Cyclic.

A random strategy chooses, as the name implies, each pixel at random. Usually, the pixels are chosen from a uniform distribution. The greedy strategy leads to cheaper iteration compared to the greedy strategy, because we do not check the gradient of each pixel.

[2]There are block coordinate descent methods that optimize a block of coordinates at each iteration. They are also discussed together with parallel coordinate descent methods in Section 7

A cyclic strategy iterates over a subset of pixels until the subset converges. It then chooses another subset. Each iteration of the cyclic strategy is also cheaper than the greedy strategy.

For our image deconvolution problem, we choose the greedy strategy. Even though it is more expensive, it tends to be faster to converge. Our reconstructed image is sparse, meaning most pixels in the image will be zero. The greedy strategy tends to iterate on pixels which will be non-zero in the final reconstructed image. While the random or cyclic strategy add pixels in the intermediate image that will eventually have to be removed. For the deconvolution problem, removing pixels from an intermediate solution seems to slow down convergence significantly. We explore the different strategies systematically for the parallel coordinate descent methods.

4.2.2 Step 2: Optimizing a single pixel

At this point, the greedy strategy has selected a pixel at a location to optimize. Now, our deconvolution objective (8.1) is reduced to a one dimensional problem:

$$\underset{x}{minimize}\ \frac{1}{2}\left\|I_{dirty} - x_{location} * PSF\right\|_2^2 + \lambda ElasticNet(x_{location}) \tag{4.4}$$

Side note: The reason why this reduces nicely to one dimension is the elastic net regularization is separable. I.e. the regularization can be calculated independently of the surrounding pixels.

Optimizing the one dimensional problem (4.4) is a lot simpler. In essence, we calculate the gradient for the pixel at the selected location, and apply the proximal operator of elastic net. First, let us look at how the gradient is calculated and ignore the regularization. The gradient arises from the data term of the one dimensional objective (4.4) ($\|I_{dirty} - x_{location} * PSF\|_2^2$). After simplifying the partial derivative, we arrive at the calculation:

$$\begin{aligned} residuals &= I_{dirty} - x * PSF \\ gradient_{location} &= \langle residuals, PSF_{location} \rangle \\ Lipschitz_{location} &= \langle PSF_{location}, PSF_{location} \rangle \\ pixel_{opt} &= \frac{gradient_{location}}{Lipschitz_{location}} \end{aligned} \tag{4.5}$$

First, we calculate the residuals by convolving the current solution x with the PSF. Then, the gradient for the selected pixel location is the inner product(element-wise multiplication followed by a sum over all elements) of the residuals and the PSF, shifted at the current location. calculate the gradient for the selected location. The next step is to calculate the Lipschitz constant at the current location. Finally, we arrive at the optimal pixel value by dividing the gradient by the Lipschitz constant. Note, that we currently ignore the elastic net regularization.

The Lipschitz constant describes how fast a function $f(x)$ changes with x. If $f(x)$ changes slowly, we can descend larger distances along the gradient without the fear for divergence. The Lipschitz constant can be looked at as a data-defined step size.

An interesting point is that the update rule $pixel_{opt} = \frac{gradient_{location}}{Lipschitz_{location}}$ finds the optimal pixel value, if the pixel is independent. Remember that our objective function is convex. The data term of our one dimensional objective (4.4) actually forms a parabola, with the parameters: $x^2\langle PSF, PSF\rangle - 2x\langle resdiuals, PSF_{location}\rangle + c$. Calculating the optimum of the parabola $\frac{-b}{2a}$, is identical to calculating the gradient update (4.5). Note that $b = -2gradient_{location}$ and $a = Lipschitz_{location}$, and both the minimum of the parabola $\frac{-b}{2a}$ and the update rule based on partial derivatives (4.5) are identical.

This means if our reconstruction problem has point sources which are far away, such that their PSFs do not overlap, then the update rule finds the optimal value for each point source with one iteration. But when the PSFs overlap as in our example problem, shown in Figure 10, then we need several iterations over the same pixel until coordinate descent converges.

Including the elastic net regularization
So far, we ignored the regularization. We can calculate the optimal pixel value without elastic net regularization. The last step is to combine the proximal operator of the elastic net regularization (4.3) with the gradient calculation, and we arrive at the following update step:

$$pixel_{opt} = \frac{max(gradient_{location} - \lambda\alpha, 0)}{Lipschitz_{location} + (1-\alpha)\lambda} \tag{4.6}$$

This update rule now finds the optimal pixel value with elastic net regularization. If the PSFs do not overlap, we still only need one iteration per source.

4.2.3 Inefficient implementation pseudo-code

Now we put together our serial coordinate descent algorithm, and show where the bottleneck lies. In each iteration, the serial coordinate descent algorithm selects the pixel with the maximum gradient magnitude, and optimizes the selected pixel with the update rule (4.6).

```
1  dirty = IFFT(GridVisibilities(visibilities))
2  residuals = dirty
3
4  x = new Array
5  objectiveValue = 0.5* Sum(residuals * residuals) + ElasticNet(x)
6
7  diff = 0
8  do
9    oldObjectiveValue = objectiveValue
10
11   //Step 1: Search pixel
12   pixelLocation = GreedyStrategy(residuals, PSF)
13   oldValue = x[pixelLocation]
14   shiftedPSF = Shift(PSF, pixelLocation)
15
16   //Step 2: Optimize pixel
17   gradient = Sum(residuals * shiftedPSF)
18   lipschitz = Sum(shiftedPSF * shiftedPSF)
19   tmp = gradient + oldValue * lipschitz
20   optimalValue = Max(tmp - lambda*alpha) / (lipschitz + (1 - alpha)*lambda)
21   x[pixelLocation] = optimalValue
22
23   //housekeeping
24   diff = newValue - oldValue
25   residuals = residuals - shiftedPSF * (optimalValue - oldValue)
26 while epsilon < Abs(diff)
```

The actual update step (line 19) is cheap to compute. We are only dealing with 4 one dimensional variables. The expensive calculations are the inner products. The gradient calculation, the Lipschitz constant and the objective value. The residuals and PSF generally contain millions of pixels. Calculating the inner product of those becomes expensive.

Also note that the greedy strategy needs to calculate the gradient for each pixel. As it is, the greedy strategy has a quadratic runtime complexity.

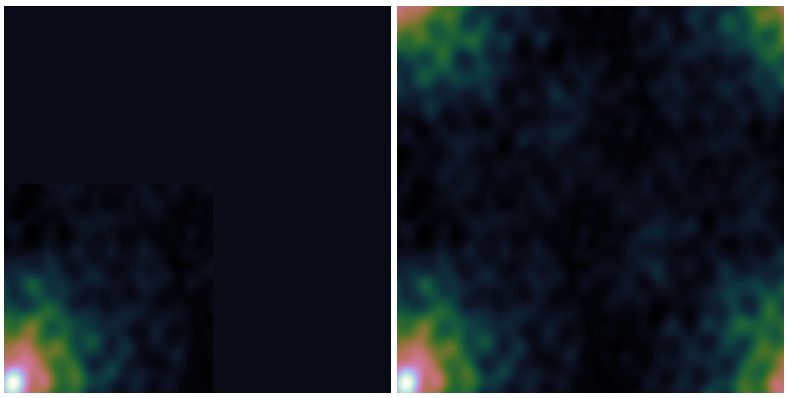
4.3 Efficient implementation

The bottleneck of the serial coordinate descent algorithm are all the inner products that need to be calculated in each iteration. In each iteration, we need to know the gradient for every pixel, and the Lipschitz constant of the current pixel. Luckily, we can cache a map of gradients, where we save the gradient for every pixel and skip all of the inner products associated with the gradient. Also, we can efficiently calculate and cache the Lipschitz constants. We can greatly reduce the runtime cost for each iteration.

This section shows the implementation details on how we can calculate the map of gradients and the Lipschitz constant efficiently. But first we need to define another implementation detail: How we handle the edges of the convolution.

4.3.1 Edge handling of the convolution

As the reader is probably aware, there are several ways to define the convolution in image processing, depending on how we handle the edges on the image. Two possibilities are relevant for radio interferometric image reconstruction: Circular and zero padded.



(a) Zero padded convolution. *(b) Circular convolution.*

Figure 11: Comparison of the two convolution schemes.

Circular convolution assumes the image ”wraps” around itself. If we travel over the right edge of the image, we arrive at the left edge. The convolution in Fourier space is circular. Remember: A convonlution in image space is a multiplication in Fourier space, and vice versa. When we convolve the reconstructed image x with the PSF using circular convolution, then non-zero pixels at the right edge of the image ”shine” over to the left edge. This is physically impossible.

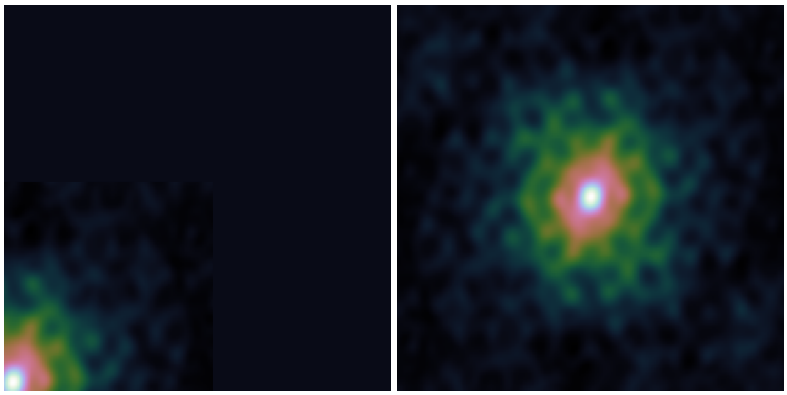
Zero padding assumes that after the edge, the image is zero. Non-zero pixels at the right edges of the image do not influence the left edge after convolution. This is the physically plausible solution. However, the zero padded convolution is more expensive to calculate. We either have to calculate the convolution in image space, which is too expensive for large kernels, or apply the FFT on a zero-padded image. Either way, it is more expensive than the circular convolution.

In designing a deconvolution algorithm, we have the choice between the circular and the zero-padded convolution scheme. Circular convolution is more efficient to calculate, while zero-padded convolution is closer to the reality. Both choices are possible. Some implementations leave this choice to the user [19]. We decide on using the zero-padded convolution. This choice influences how we calculate the Lipschitz and gradients efficiently.

4.3.2 Efficient calculation of the Lipschitz constants

In each iteration, we need the Lipschitz constant of the current pixel. I.e. we need the inner product $\langle PSF_{location}, PSF_{location} \rangle$ for every pixel. We can pre-calculate the Lipschitz constant before we run the serial coordinate descent algorithm. The naive way to calculate the Lipschitz constant for every pixel results in quadratic runtime(each inner product costs us $O(n)$ operations, and we do it for all n pixels). But this is not necessary. We can pre-calculate the Lipschitz constant for every pixel in linear time.

Figure 12a shows the PSF shifted to a pixel location. The Lipschitz constant is by squaring all values of Figure 12a and summing up the values. Or in another way: We sum up all the squared values of the PSF inside a specific rectangle. All that changes for a Lipschitz calculation between different pixels is the specific rectangle.



(a) Shifted PSF. *(b) Sum of squared values.*

Figure 12: Sum of squared values for the Lipschitz constant.

This can be exploited with a scan algorithm: We first calculate the result of every rectangle we can draw from the origin, up to some pixel value. We end up with an array we call $scan[,]$. It is the same size as the PSF, but contains the sum of squares inside a specific rectangle.

```
var scan = new double[,];
for (i in (0, PSF.Length(0))
  for (j in (0, PSF.Length(1))
    var iBefore = scan[i - 1, j];
    var jBefore = scan[i, j - 1];
    var ijBefore = scan[i - 1, j - 1];
    var current = PSF[i, j] * PSF[i, j];
    scan[i, j] = current + iBefore + jBefore - ijBefore;
```

Every Lipschitz constant can be now calculated by combining the sums of different rectangles. Our example is shown in Figure 12b. We start with the total sum of all values, and subtract two rectangles. Because the subtractions overlap, we need to add the third rectangle again. we take the total value. In short, we can calculate each Lipschitz constant by at most 4 lookups in the $scan[,]$ array.

4.3.3 Using a map of gradients

For an efficient greedy strategy, we need to know the gradient for each pixel. We show how to calculate the initial map of gradients in linearithmic time($O(nlogn)$ and how to update it directly after a change in the reconstructed image x. As we will see, we can use a map of gradients and drop the residual calculation from the algorithm. The gradient map implicitly contains the information of the residuals.

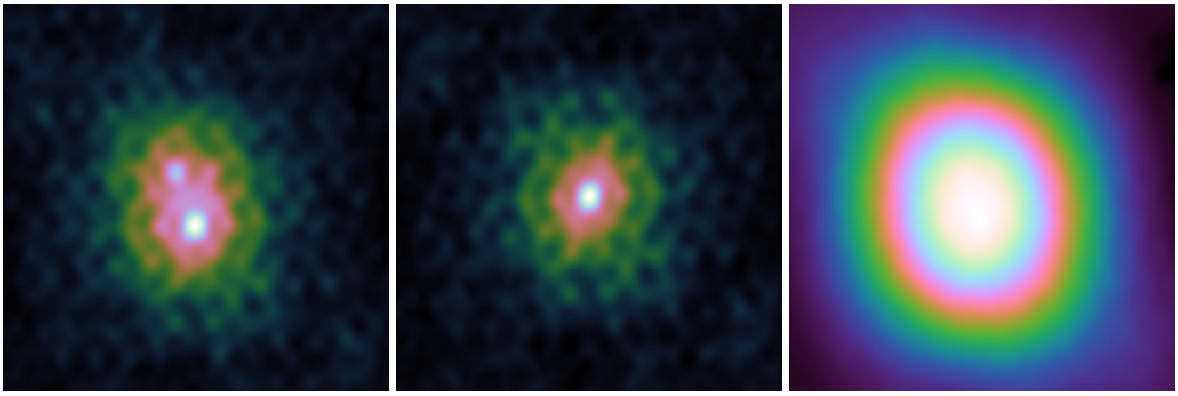
Efficient calculation

Calculating the gradient for each pixel results again in a quadratic runtime. We need to calculate $\langle residuals, PSF_{location} \rangle$ for every pixel (Similarly to the Lipschitz constants, each inner product costs us $O(n)$ operations, and we do it for all n pixels). But we can use the FFT to calculate the map of gradients in linearithmic time.

Note that the inner product is actually a correlation: We correlate the PSF with the residuals. The correlation and the convolution are related. The convolution is simply a correlation with a flipped kernel. This means we can use the FFT to efficiently calculate the correlation of the residuals and the PSF:

$$\begin{aligned} psfFlipped &= FlipUD(FlipLR(PSF)) \\ gradients &= iFFT(FFT(residuals) * FFT(psfFlipped)) \end{aligned} \tag{4.7}$$

A convolution in image space is a multiplication in Fourier space. This fact can also be exploited for the correlation by flipping the PSF. Since the FFT takes linearithmic time $O(nlogn)$ to compute, the overall operation also takes us linearithmic time. The operation is shown in Figure 13.



(a) Dirty Image. *(b) Point Spread Function.* *(c) Gradient map.*

Figure 13: Example of the gradient calculation.

Direct update of gradient map

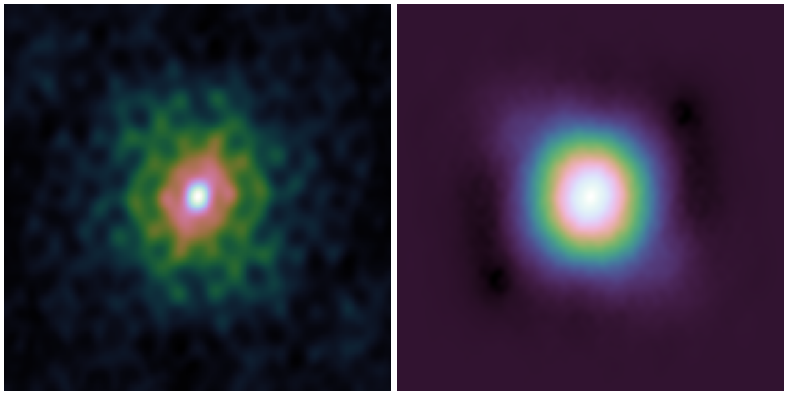
Thanks to the FFT, we can efficiently calculate the map of gradients at the start of the serial coordinate descent algorithm. After we update a pixel in the reconstruction x, the map changes. We could repeat the correlation in the Fourier space from equation (4.7) at each iteration. But this is wasteful. We can update the map of gradients directly.

Note that if we add pixel in the reconstruction x, we subtract the PSF (multiplied with the pixel value) from the specific location in the residuals. The gradient map is then calculated by again correlating the PSF with the residuals. We update the residuals by subtracting the PSF at the correct location. And we update the gradient map by subtracting the PSF correlated with itself at the correct position ($PSF \star PSF$). In our simulated example, the PSF and the gradient update map is shown in Figure 14b.

This means we can update the gradient map directly with the product of $(PSF \star PSF)$. We can simply shift $(PSF \star PSF)$ at the correct pixel location and subtract it from the gradient map directly.

There is one issue though: We use zero padded convolution. The PSF at the edges of the image is masked. This means that the product $(PSF \star PSF)$ actually changes with the pixel location. If we update a pixel in the corner of the image, the actual update $(PSF \star PSF)$ at that location looks different than what was shown in Figure 14b.

The exact update is again expensive to calculate. We need to correlate the PSF with itself for every pixel location. This is as repeating equation (4.7) in each iteration (re-calculating the PSF correlation with the



(a) Point Spread Function. *(b) Gradient update:* $(PSF \star PSF)$.

Figure 14: Example problem with two point sources.

residuals in each iteration). However, the exact gradient update only changes significantly at the edges of the image, when large parts of the PSF are masked by the edges. Otherwise the difference between the exact update and simply shifting $(PSF \star PSF)$ at the pixel location is small.

This is the reason why we chose to accept that the gradient update is only an approximation. The approximation only becomes inaccurate at the edges, and we use the algorithm in the major cycle framework: Every major cycle removes any inaccuracies we introduced during the previous cycle. This may potentially lead to more major cycles until the algorithm converge to the solution. But in practice the serial coordinate descent algorithm did not need more major cycles than CLEAN. We compare CLEAN and the serial coordinate descent algorithm on a real-world observation in Section 6.

4.4 Efficient implementation pseudo-code

Now we put all the run time improvements discussed before into the new implementation of the serial coordinate descent algorithm. We pre-calculate the Lipschitz constants and the gradient map. Then during iterations, we update the gradient map directly.

This leads to the following algorithm:

```
dirty = IFFT(GridVisibilities(visibilities))
residualsPadded = ZeroPadding(dirty)

psfPadded = ZeroPadding(PSF)
psfPadded = FlipUD(FlipLR(psfPadded))
gradientUpdate = iFFT(FFT(ZeroPadding(PSF)) * FFT(psfPadded))

x = new Array[,]
gradientsMap = iFFT(FFT(residualsPadded) * FFT(psfPadded))
lipschitzMap = CalcLipschitz(PSF)

objectiveValue = 0.5* Sum(residuals * residuals) + ElasticNet(x)
maxAbsDiff = 0
do
  oldObjectiveValue = objectiveValue

  //Step 1: Search pixel
  maxAbsDiff = 0
  maxDiff = 0
  pixelLocation = (-1, -1)
```

```
  for(i in Range(0, dirty.Length(0))
    for(j in Range(0, dirty.Length(1))
      oldValue = x[i, j]
      tmp = gradientsMap[i, j] + oldValue * lipschitzMap[i, j]
      optimalValue = Max(tmp - lambda*alpha) / (lipschitz[i, j] + (1 - alpha)*lambda
          )
      diff = optimalValue - oldValue

      if(maxAbsDiff < Abs(diff))
        maxAbsDiff = Abs(diff)
        maxDiff = diff
        pixelLocation = (i, j)

  //Step 2: Optimize pixel. Now all that is left is to add the maximum value at the
      correct location
  x[pixelLocation] += maxDiff

  //housekeeping
  shiftedUpdate = Shift(gradientUpdate, pixelLocation)
  gradientMap = gradientMap - shiftedUpdate * maxDiff
while epsilon < maxAbsDiff
```

In step 1, we replaced the gradient and Lipschitz calculation with lookups, reducing the runtime costs of the step. In this implementation, step 2 is trivial. We only need to update the reconstruction at the correct location. All the important work has already been completed in step 1.

The two most time consuming parts of this implementation are step 1, and updating the gradient map. Our implementation in .Net Core uses parallel computing for both parts. We search for the maximum pixel in parallel, and update the gradient map in parallel.

4.5 GPU implementation

We implemented the serial coordinate descent algorithm on the GPU. It is implemented in .Net Core with ILGPU[20]. ILGPU is a Just-In-Time compiler for high performance GPU programs written in .Net Core.

GPU programs are split into kernels. Each kernel consists of a single routine optimized for executing on the GPU. Our serial coordinate descent algorithm consists of Step 1, find the best pixel to optimize, step 2, optimize pixel and then updating the gradient map. The GPU implementation of the Serial coordinate descent algorithm uses three kernels: Kernel 1 is equivalent to step 1. Kernel 2 updates the reconstruction x and kernel 3 updates the gradient map.

Kernel 2 and 3 are straight forward to implement. The implementation of kernel 1, searching for the best pixel, is more interesting. Essentially, this step is a max reduce operation: We want to find the pixel with the maximum absolute step. We implemented the step 1 kernel with an atomic-max instruction:

```
MaxPixelKernel(x, gradienstMap, lipschitz, location, maxPixel)
  oldValue = x[location]
  tmp = gradientsMap[location] + oldValue * lipschitzMap[location]
  optimalValue = Max(tmp - lambda*alpha) / (lipschitz[location] + (1 - alpha)*lambda
      )
  diff = optimalValue - oldValue
  currentPixel = (absDiff = Abs(diff), diff = diff, location = location)
  AtomicMax(maxPixel, currentPixel)
```

The kernel is executed on multiple processors in parallel on the GPU. Each processor is checking a single pixel. Communication is done with the atomic-max instruction. The atomic-max writes on a global variable, which keeps track of the current maximum pixel. This implementation turned out to be the fastest for the MaxPixelKernel. Warp shuffle[21] was also tested, but resulted in a slower kernel.

Synchronization

Synchronization between the kernels. It is a serial coordinate descent method. We need to wait for all kernels to finish before we can continue.

```
do
  //Step 1: Search pixel
  maxPixel = (absDiff = 0, diff = 0, location = (-1, -1))
  ExecuteMaxPixelKernel(x, gradienstMap, lipschitz, maxPixel)
  SynchronizeKernels()

  //Step 2: Optimize
  ExecuteUpdateXKernel(x, maxPixel.diff, maxPixel.location)
  ExecuteUpdateGradientsKernel(gradientsMap, gradientUpdate, maxPixel.diff, maxPixel
      .location)
  SynchronizeKernels()

while maxPixel.absDiff < epsilon
```

ILGPU implementation finished.

4.6 Distributed implementaion MPI

How we distribute it with MPI.

Message pasing interface (MPI)

We split up the image and the gradient map into patches.

Each simply updates its patch.

MPI Allreduce

4.7 Serial coordinate descent and similarities to the CLEAN algorithm

When we look back at the CLEAN algorithm, we start to see similarities to the serial coordinate descent algorithm.

CLEAN finds the maximum in the residual image. Serial Coordinate descent finds the maximum in the gradient map.

CLEAN then removes a fixed fraction of the PSF at that point. Serial coordinate descent uses the Lipschitz constant, and subtracts the PSF correlated with itself at that point.

The main difference is that serial coordinate descent calculates the gradient, and CLEAN does not. CLEAN is more a matching pursuit algorithm. But the structure is the same. We can use the serial coordinate descent GPU and MPI implementation. With a few tweaks, we would arrive at the CLEAN algorithm.

By accident, we also found a GPU and MPI implementation for standard CLEAN.

Multi-scale CLEAN is what is used. Difficulty with MPI implementation because of a distributed convolution.

5 PSF approximation for parallel and distributed deconvolution

Because of the Major cycle, we already use an approximate PSF.

For a distributed deconvolution, we would like to deconvolve the image with as little communication as possible. This largely depends on the size of the PSF when compared to the overall image. If the PSF is for example $\frac{1}{16}$ of the total image size, we have patches of the image which are completely independent of each other. Sadly, this is not true for radio interferometers. The PSF is generally the same size as the image. We cannot deconvolve any part of the image independently of each other.

However, we have two effects of modern radio interferometers, that produce an "approximately" smaller PSF: First, we have an increasing number of visibilities. They create a PSF that increasingly resembles a Gaussian function in the center, and the rest approaches zero. And secondly, we reconstruct images with a wide field-of-view. Although the PSF is not zero the further away we move from the center, its values approach zero.

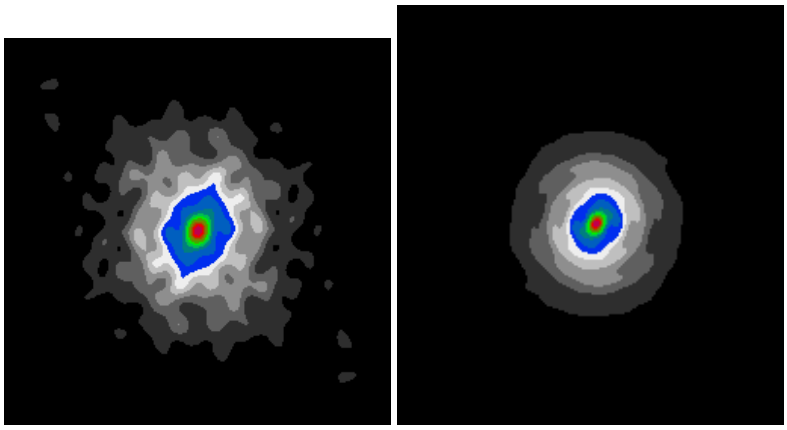


Figure 15: PSF arising from an increasing number of visibilities.

In short, with an ever increasing number of visibilities and field-of-view, the influence of far-away image sections become negligible. We can approximate the deconvolution with a fraction of the true PSF. To our knowledge, we are the first to propose such approximation methods. In this Section, we present our approximation methods. In Section 6.3, we empirically demonstrate the validity of our approximations on a real-world MeerKAT observation. In Section 7, we show more sophisticated coordinate descent methods that can exploit the smaller PSF.

5.1 Intuition for approximating the PSF

Our basic coordinate descent algorithm chooses a pixel to minimize, calculates its gradient and descends in that direction. The gradient calculation reduces itself to a correlation of the residuals with the PSF at the pixel location. In other words, we need the PSF to calculate a gradient. If we only use parts of the PSF for the calculation, we essentially approximate the gradient for the pixel. Because the PSF only has significant non-zero values in the center, we should be able to ignore most of the values and still have an adequate gradient approximation.

Furthermore, our basic coordinate descent algorithm reconstructs inside the Major/Minor cycle framework. The framework is designed to handle only an approximate PSF in the deconvolution (Remember: the w-term changes the PSF depending on the position in the image). The framework may be able to deal with further PSF approximations, namely with a PSF that is reduced in size, which makes the distributed deconvolution simpler.

The approximation methods essentially work by "cutting off" the less significant PSF values and only use a rectangle around the center. If we cut off the PSF by a factor of $\frac{1}{4}$ (If the PSF is 1024^2 pixels in size,

we only use a rectangle of 256^2 of the center), we get pixels in the reconstructions that are not influenced by each other. Starting from a cut-off factor of $\frac{1}{16}$, we can split the image into eight patches, four of which are independent of each other. This would allow us to run up to four basic coordinate descent algorithms in parallel, simplifying the distribution.

Indeed, it is possible to approximate the PSF with only a fraction of its true size, as we will demonstrate empirically in Section 6.3. But an approximation of the PSF may lead to other problems in the reconstruction:

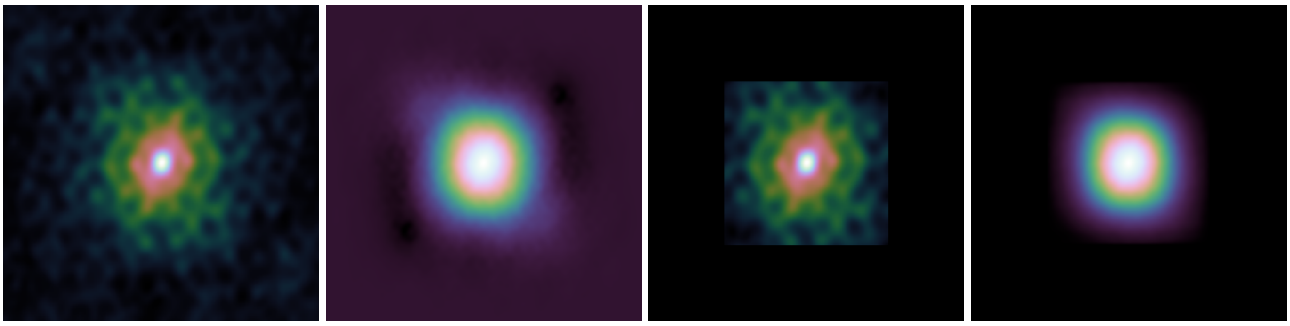
- Needs additional Major cycles to converge.
- Slow down convergence speed of coordinate descent.
- Not guaranteed to converge to the same image.

The Major cycle corrects the errors the approximate PSF introduces. The more inaccurate the PSF is, the more Major cycles we may need to converge. As we already discussed, a Major cycle is an expensive operation. Our PSF approximation should lead to as few (if any) additional Major cycles.

For the basic coordinate descent algorithm, an approximate PSF may slow down the convergence speed. In each iteration, the basic algorithm finds the optimal value for the current pixel. With an approximate PSF, we may need several iterations on the same pixel (over several major cycles) until we arrive at the same value. In short, an approximate PSF can slow down the convergence speed of coordinate descent.

Depending on how we approximate the PSF, we may not have any guarantee that we arrive at the same result. We developed two approximation methods: Method 1 uses an approximation in the gradient update step of coordinate descent. Method 2 solves an approximate deconvolution problem with only a fraction of the PSF. Only method 1 is guaranteed to converge to the same solution (with enough major cycles), but is slower to converge than method 2. Depending on the method we use, we can remedy some of the problems that approximating the PSF introduces. But there seems to be a trade-off to be made. We have not found a method that works best in every aspect.

5.2 Method 1: Approximate gradient update



(a) PSF *(b) Gradient update* *(c) $cut(PSF)$* *(d) Approx. update*

Figure 16: Approximation of gradient update.

The basic coordinate descent method first correlates the residuals with the PSF. It pre-calculates the gradient for each pixel. Then, in each iteration, it directly updates the map of gradients with the product of $PSF \star PSF$ (the PSF correlated with itself). In this approximation method, we start from the same pre-calculated map of gradients, but use an approximate update. The first coordinate descent iteration of this approximation method is identical to coordinate descent with the full PSF. With each coordinate descent iterations, the gradient map becomes more inaccurate. But with enough major cycles, this method converges to the same result as when the full PSF is used.

The question is, how do we approximate the product of $PSF \star PSF$. As we have seen before, the product of $PSF \star PSF$ also approaches zero away from the center (an example was shown in Figure 14 in Section **??**). A naive way to approximate the update step is to cut off the insignificant value and only use a rectangle of the center, which is a fraction of the total image size. For example: An image of size 1024^2 also has a PSF the size of 1024^2. The product $PSF \star PSF$ actually has the size of 2048^2 pixels due to the correlation. We can try to approximate the product by only using the center rectangle $\frac{1}{8}$ of the total size, (256^2 pixels). This approximation works, but leads to artifacts during deconvolution: The image will be reconstructed in visible "blocks" which are the size of the center fraction we use.

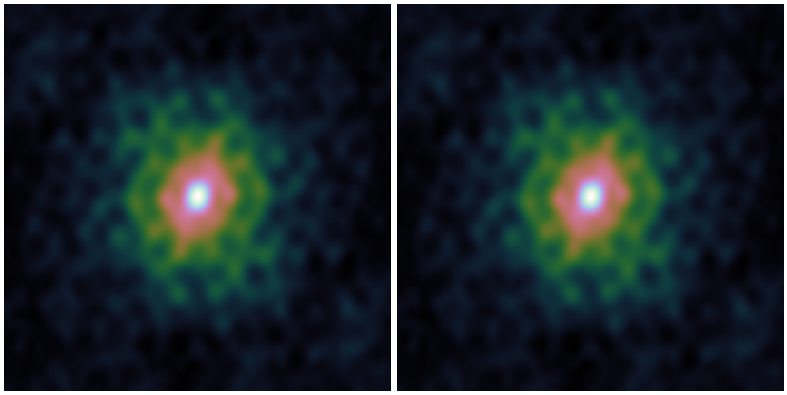
We use the approximation shown in equation (5.1), where $cut()$ is the function that cuts away everything but the center rectangle of the PSF. This is a better approximation than cutting the product of $PSF \star PSF$ directly, and leads to faster convergence.

$$PSF \star PSF \approx cut(PSF) \star cut(PSF) \tag{5.1}$$

The reason why (5.1) is a better approximation lies in the reason why we update the gradients with the product of $PSF \star PSF$ in the first place: It is the combination of two separate operations, removing the PSF from the residuals at the current position, and recalculating the correlation with the PSF. If we cut away parts of the product $PSF \star PSF$ directly, we implicitly update the residuals with a different PSF. But when we approximate the product by (5.1), we ensure that the implicit removal of the PSF from the residuals is equal to $cut(PSF)$.

In our implementation, we use one more trick to improve the approximation: we scale the product of $cut(PSF) \star cut(PSF)$ to have the same maximum as the original product $PSF \star PSF$. The approximation has a lower maximum value than the original. Over several coordinate descent iterations, we run into the danger of overestimating the pixel values. By scaling the product of $cut(PSF) \star cut(PSF)$ to the same maximum, we end up with a better approximation of the true gradient update.

5.3 Method 2:Approximate deconvolution



(a) PSF on log scale *(b) PSF Approximation*

Figure 17: Approximate deconvolution with a fraction of the PSF.

The main problem with Method 1 is that the map of gradients becomes less accurate with more coordinate descent iterations. This method sovles this problem by using an approximate deconvolution instead, but it loses the guarantee to converge to the same result as the full PSF.

This method cuts off the insignificant part of the PSF, and only uses the center rectangle of it for the whole deconvolution problem. As such, the coordinate descent method solves an approximate deconvolution problem

shown in (5.2).

$$\underset{x}{minimize}\ \frac{1}{2}\left\|I_{dirty} - x * cut(PSF)\right\|_2^2 + \lambda ElasticNet(x) \tag{5.2}$$

In essence, it uses the same basic coordinate descent method, but ignore large parts of the PSF. It pre-calculates the gradient map by correlating the residual image with $cut(PSF)$, and update the gradient map with the product of $cut(PSF) \star cut(PSF)$. The main difference between approximate gradient update and approximate deconvolution is: In approximate gradient update starts with the same gradient map as the original. The approximate deconvolution does not. With this method, the gradient map does not become more inaccurate with more coordinate descent iterations.

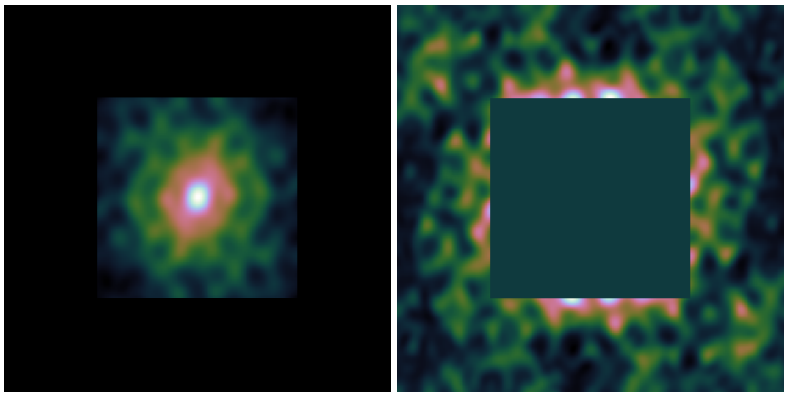
The approximate deconvolution objective (5.2) is not guaranteed to "point" to the same solution as the original. It may in reality point to a very different solution. But thanks to the Major cycle, the image retrieved by optimizing (5.2) is always "close" to the original solution.

Nevertheless, this approximation method introduces an error in the final reconstructed image. The obvious error it introduces is it under-estimates the true pixel values. Pre-calculating the gradient map with $cut(PSF)$ under-estimates gradient magnitudes, and by extend the pixel values.

To combat the under-estimation of pixel values, we reduce the regularization parameter λ for the approximate deconvolution problem. Since we cut off parts of the PSF, we also reduce the Lipschitz constant (sum of the squared PSF values) used in the approximate deconvolution. We reduce the λ parameter by the same factor that the Lipschitz constant gets reduced. This ensures that the approximate deconvolution and the original deconvolution arrive at the same pixel value for a point source. But it does not completely remove the issue for extended emissions.

5.4 Major Cycle convergence and implicit path regularization

In the two presented methods, we use a fraction of the PSF to approximate the deconvolution problem. Both methods rely on the Major Cycle to periodically reset the gradient map. By using only a fraction of the PSF, the approximate deconvolution leaves parts of the PSF in the gradient map. Figure 18 shows the fraction of the PSF that get included in the deconvolution, and "sidelobes" which get ignored.



(a) PSF $\frac{1}{2}$ cutoff *(b) Sidelobes*

Figure 18: Max sidelobe PSF.

After a number of serial coordinate descent iterations, we run into the danger of deconvolving the leftovers of the PSF approximation. In that case, the serial coordinate descent algorithm adds spurious point sources to the image. After a major cycle, we can detect them as spurious and remove them again. Degree to how bad this is depends on the maximum absolute value we cut off in the approximation, the maximum absolute value

in Figure 18b. With an aggressive approximation, we may end up oscillating from major cycle to major cycle: Adding spurious pixels and removing them again in the next, just to add the same spurious pixels in the one after.

But we can estimate at what point we are likely to add spurious pixels. This lets us estimate a minimum λ parameter for the elastic net regularization. At this major cycle iteration, we cannot go below the minimum lambda. The next major cycle, the maximum residual is lower, and so is the minimum λ

This is known as a path regularization. We start with a stronger regularization and continually reduce λ until we reached the desired value. We have intermediate results in each major cycle. Coordinate descent methods tend to be faster from a warm start, when we start from an intermediate result [22].

In this work we use the path regularization mainly for convergence of the major cycle.

How we estimate the minimum λ parameter. Imagine the interferometer only observed a point source in the center of the image. In that case, after the first serial coordinate descent iteration, our residuals image looks exactly like the Figure 18b. The serial coordinate descent algorithm should not take another step, or it adds spurious pixels. In other words, the regularization has to suppress all the values in Figure 18b

Remember the elastic net regularization: It is a mixture of the L1 and L2 norm. The L1 norm shrinks the pixel values, while the L2 norm divides them. We need the L1 norm to shrink away the values in Figure 18b.

$$\begin{aligned} gradients &= residuals \star PSF \\ maxSidelobe &= Max(gradients) * Max(Abs(Sidelobe(PSF)))) \\ \lambda_{cycle} &= \frac{maxSidelobe}{\alpha} \end{aligned} \tag{5.3}$$

This estimate is a minimum. Meaning it is possible that we still add spurious pixels. For example when the sidelobes of two point sources overlap.

The estimate however only considers point sources. For extended emissions, the estimate is too low. Extended emissions are a group of non-zero pixels. Beause they are next to each other, their sidelobes overlap and get amplified.

An estimate considering extended emissions.

$$\begin{aligned} psfSidelobe &= Max(PSF - cut(PSF)) \\ gradients &= residuals \star PSF \\ correction &= Max(1, \frac{Max(gradients)}{Max(residuals) * lipschitz}) \\ maxSidelobe &= Max(gradients) * Max(psfSidelobe) * correction \\ \lambda_{cycle} &= \frac{maxSidelobe}{\alpha} \end{aligned} \tag{5.4}$$

Only considers the ”maximum” extended emission. Extended emissions tend to contain the largest pixel value in the residual image. Because of the convolution. The correction factor gets reduced the closer the maximum pixel in the residuals. Over several major cycles, the correction factor and the $lambda_{cycle}$ become smaller. Helps convergence.

But this is not true for extended emissions.

he serial deconvolution algorithm calculates the gradient map (by correlating the PSF with the residuals).

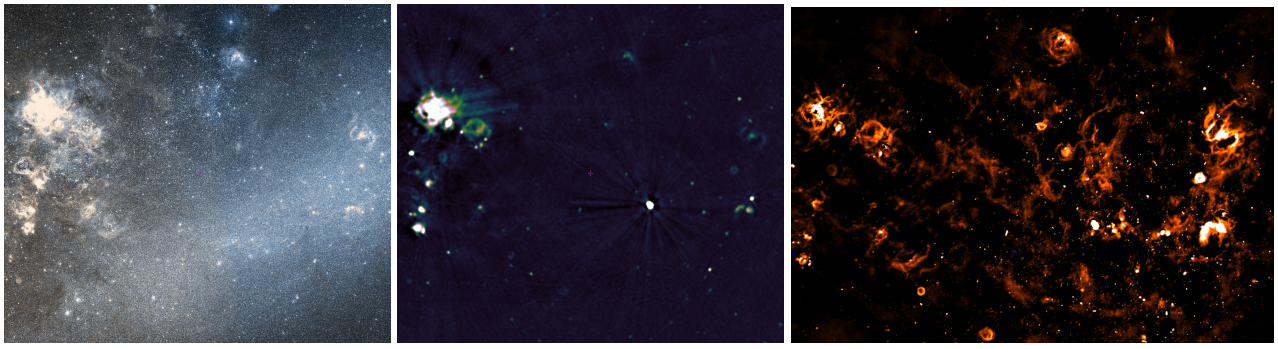
Residuals will be left.

6 Tests on MeerKAT LMC observation

The Large Magellanic Cloud (LMC) is a galaxy is the second or third closest galaxy to the Milky Way. Figure 19 shows the LMC in both optical and radio wavelenghts. The radio wavelengths was observed by the VLA radio interferometer[23] at 843MHz. In the optical wavelengths, the abundance of stars are clearly visible. The LMC is close enough to earth for individual stars are visible. But it also contains a large number of supernova remnants, gas clouds, and other extended emissions, which shine bright in the radio wavelengths.

The LMC is a region with a large number of sources at different brightness. In the lower-right quadrant of the radio-image 19b, we see the bright emission of the supernova remnant N132D, the brightest radio source in the LMC. But around the N132D are faint emissions from gas-clouds. This means faint emissions may get lost next to N132D. We need a deconvolution algorithm to uncover these faint emissions.

We received a MeerKAT observation of the LMC from SARAO for the purpose of algorithm testing. At the time of writing, the MeerKAT instrument is still being tested. The observation is only representative in the data volume. The observation is calibrated, and averaged down in both frequency and time. The averaging reduces both the disk space and the runtime costs of the gridding step. Nevertheless, the observation takes up over 80 GB of disk space (roughly $\frac{1}{30}$ of the original data). A CLEAN reconstruction of the calibrated observation is shown in Figure 19c.



(a) Optical wavelength *(b) Radio wavelength at 843MHz.* *(c) Wide band radio image by MeerKAT.*

Figure 19: Section of the Large Magellanic Cloud (LMC)

The MeerKAT observation covers a wide band of radio frequencies. The lowest frequency in the MeerKAT observation is 894 MHz, and the highest frequency is 1658 Mhz. Imaging the whole frequency band requires a wide band deconvolution algorithm. In wide band imaging, several images at different frequencies get deconvolved as an image cube. Wide band imaging again multiplies the amount of work that has to be done for reconstruction, as now we cannot deconvolve a single image, but have to deal with a whole image cube.

Wide band imaging is not possible within the time frame of this project. We take a narrow band subset of 5 channels from the original data (ranging from 1084 to 1088 MHz, about 1 Gb in size) for reconstruction. We also reduce the field-of-view to a more manageable section. Figure 20 shows the LMC image section we are using together with a CLEAN reconstruction of the narrow band data.

At the center of our image section 20 we see the N132D supernova remnant. We partially see the faint extended emissions, although they are close to the noise level. This is known as a high-dynamic range reconstruction. We have strong radio sources mixed together with faint emissions, which are only marginally above the noise level of the image.

The total field-of-view of our image section is roughly 1.3 degrees(or 4600 arc seconds). Our reconstruction has 3072^2 pixel with a resolution of 1.5 arc seconds per pixel. this is still a wide field-of-view reconstruction problem. We have to account for the effects of the w-term to achieve a high-dynamic range reconstruction.

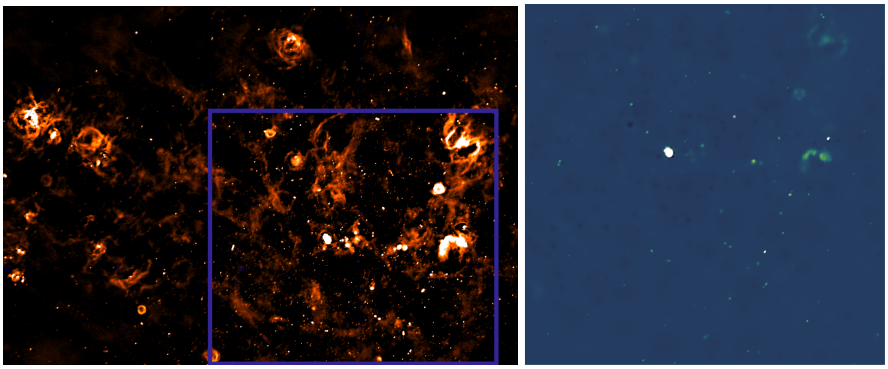


Figure 20: Narrow band image section used.

In our test reconstruction, we need to account for w-term correction and high-dynamic range. We have excluded wide-band imaging as not feasible within the time frame of this project. In Section 6.1 we compare the reconstructions of CLEAN with our coordinate descent based algorithm on the LMC observation. The next Section 6.2 presents the speedup we achieve with coordinate descent by using our distributed or GPU-accelerated implementations.

In Section 6.3 we show the core result of this project. Namely what effect has an approximate PSF on the deconvolution problem and whether we can use it to further distribute the problem. The answer to that question is affirmative: We can approximate the PSF, and we can exploit it to further distribute the deconvolution. But we need more sophisticated coordinate descent algorithms to fully benefit from it.

6.1 Comparison with CLEAN reconstructions

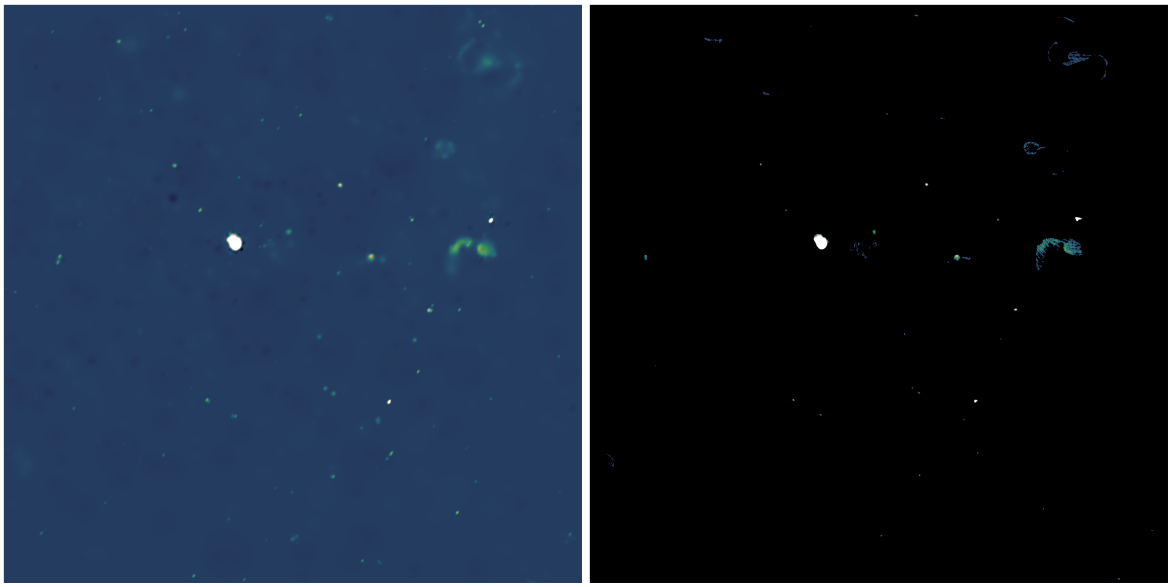
We use the WSCLEAN [24] implementation of multi-scale CLEAN. We compare our coordinate descent reconstruction with two CLEAN reconstructions, oe with naturally weighted visibilities and one with briggs weighted visibilities.

There are three main visibility weighting scheme for the gridder that lead to different PSFs from the same measurements: Natural, uniform, and Briggs[25]. Natural weighting scheme leads to an image with a lower noise level, but a wider PSF. Uniform weighting leads to a higher noise level, but to a PSF whgich is more concentrated around a single pixel. Briggs weighting is a scheme combines the best from both worlds, receiving an image with acceptable noise level while getting a more concentrated PSF. As such it is widely used in radio astronomy image reconstruction. Our gridder implements the natural weighting scheme only. Nevertheless our coordinate descent algorithm is able to retrieve structures similar to the briggs-weighted multi-scale CLEAN reconstruction, even though coordinate descent has to work with a wider PSF.

Figure 21 shows the reconstruction of both briggs-weighted multi-scale CLEAN and the naturally weighted coordinate descent reconstruction. CLEAN used 6 major cycles and 14 thousand minor cycle iterations. Our coordinate descent implementation converged after 5 major cycles and needed 100 thousand iterations to converge.

Coordinate descent needs a large number of iterations to converge when compared to multi-scale CLEAN. Note that a coordinate descent iteration is cheaper to compute than one iteration of multi-scale CLEAN. Also note that because we are searching for structures close to the noise level of the image, coordinate descent often adds pixels belonging to the noise in one major cycle, just to remove them in the next one. Path regularization[22] can combat this problem, and gets further investigated in the following Section 6.3.

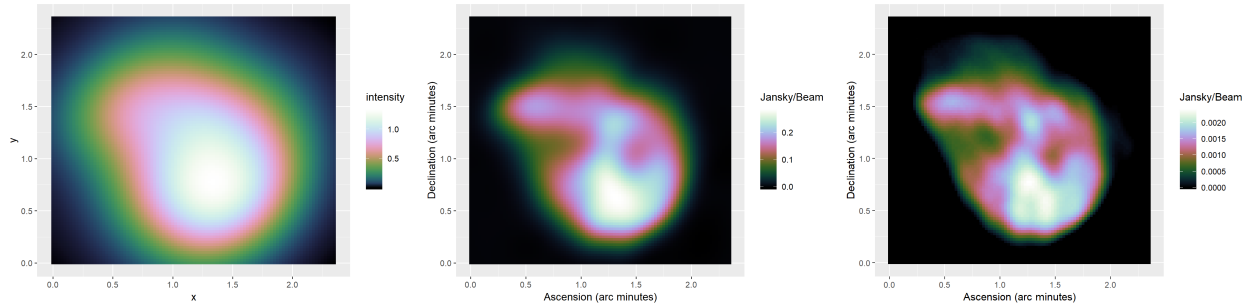
Both algorithms detect the three extended emissions at the right side of the image. They detect various point sources at the same location. Coordinate descent and multi-scale CLEAN arrive at a roughly similar



(a) Briggs weighted multi-scale CLEAN. *(b) Naturally weighted coordinate descent.*

Figure 21: Comparison of the whole image

result. Coordinate descent detects similar structures in the N132 supernova remnant, as the briggs-weighted CLEAN, but also includes calibration errors in its reconstruction of the faint extended emissions.



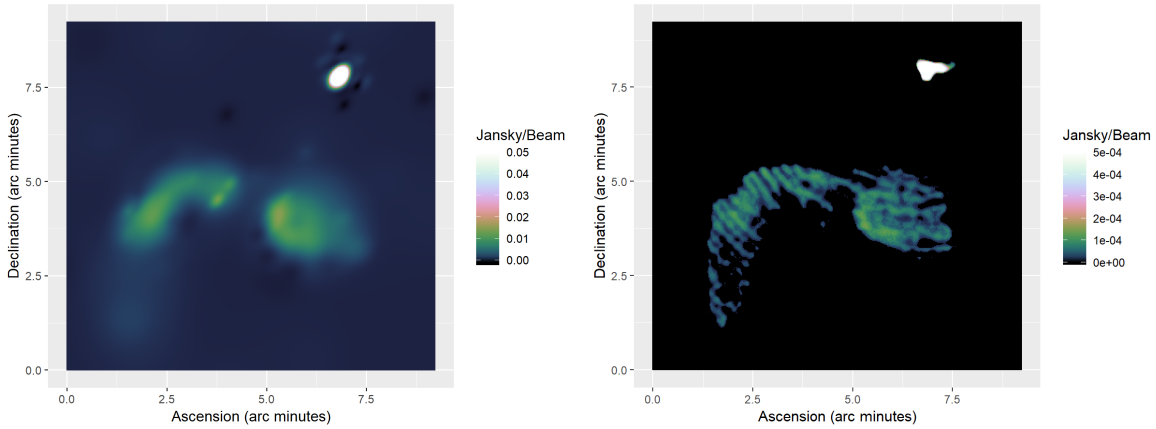
(a) CLEAN Natural weighting. *(b) CLEAN Briggs weighting.* *(c) CD Natural weighting.*

Figure 22: N132 comparison

Figure 22 compares the naturally-weighted CLEAN, briggs CLEAN and coordinate descent on the N132 supernova remnant. The naturally-weighted CLEAN and coordinate descent use the same PSF for the deconvolution. But coordinate descent finds structures in N132 similar to the briggs-weighted CLEAN. Coordinate descent arrived at a plausible higher-resolved reconstruction of N132.

Calibration errors on the other hand negatively influence the coordinate descent reconstruction. Figure 23 shows a cutout of the right hand section of the reconstruction, where a faint extended emission is next to a point source with calibration errors. Multi-scale CLEAN is able to differentiate between the ”ripples” from the calibration error, and the signal from the extended emission. Coordinate descent with the elastic net regularization includes the ripples into the reconstructed image.

The only way to exclude the ripples from the reconstruction is to increase the regularization parameter λ., such as no pixel gets included which is not above the noise level + calibration error in the image. However, that would lead to other sources being ”regularized away” in other regions of the image, which do not have a severe calibration error close by.



(a) Briggs CLEAN *(b) Coordinate Descent*

Figure 23: Influence of calibration errors

6.2 Coordinate descent acceleration with MPI or GPU

Describe hardware

Distributed with MPI

GPU implementation

Measurement of the speedup.

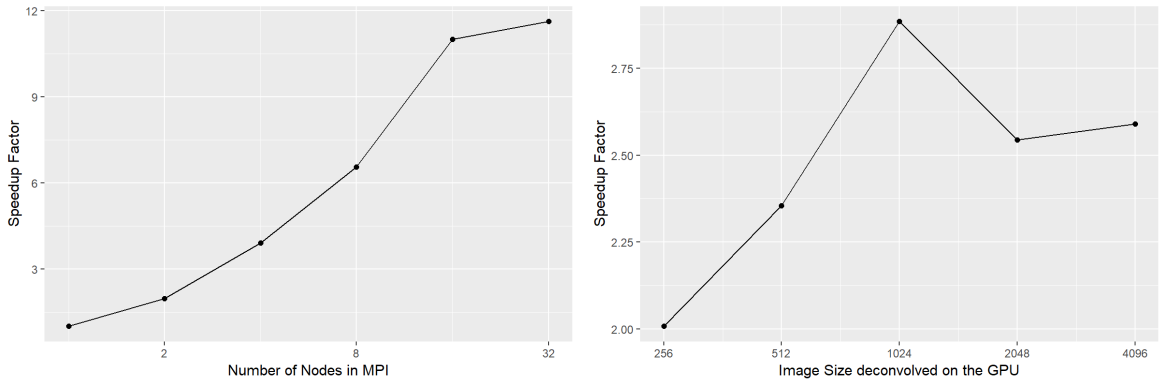


Figure 24: Speedup by using MPI or GPU acceleration

We cannot use MPI combined with the GPU. The MPI implementation uses a communication step in each coordinate descent iteration (communicating which pixel to optimize with MPI Allreduce).

6.3 Effect of approximating the PSF

As we described in Section 5, the PSF for deconvolution is as big as the image. For wide field-of-view observations of MeerKAT, the PSF is approximately a gaussian with decreasing pixel values the further we move from the center. Most of the values in the PSF are close to zero. The question is, what effect has an approximate deconvolution with a smaller PSF? If we can approximate the deconvolution with a small enough PSF, we can solve patches of the image independently of each other. However, the PSF approximation may need more major cycles to converge.

The effect of approximating the PSF are not clear. We know that thanks to the w-term in the visibilities,

the PSF is not constant over the image. We already need several major cycles to converge. With a good approximation of the PSF, we may speed up the individual iterations of coordinate descent without needing more major cycle.

We presented two methods to approximating the PSF for the deconvolution in Section 5. Method 1 updates only a fraction of the gradients, and Method 2 uses a fraction of the PSF for deconvolution. We test both methods on the LMC data and explore what effects the approximations have on the reconstruction.

6.3.1 Method 1: Approximate gradient update

Our coordinate descent method updates the map of gradients after each iteration. Method 1 starts with the same map of gradients as the original, but then only updates a fraction of the gradients in each iteration. It updates a rectangle of the most significant gradients.With each coordinate descent iteration, the map of gradients gets less accurate. Because we do an approximate update of gradients, this method should converge to the same result as the original with enough major cycles.

At the beginning of each major cycle, we calculate the objective value of the current solution. We compare the objective value and the wall-clock time of the original and the approximate gradient update. This is a minimization problem, meaning the lowest objective is the most accurate reconstruction (according to the elastic net regularization).

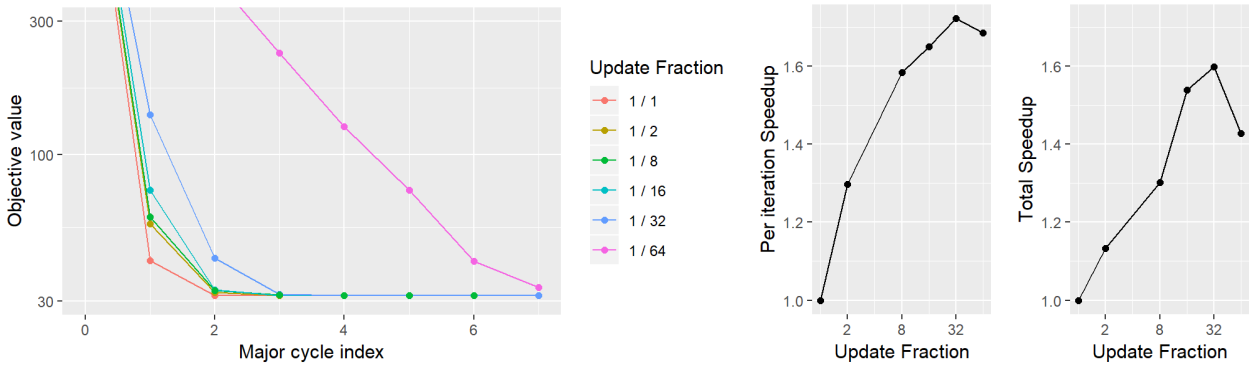


Figure 25: Effect of only updating a fraction of the gradients.

The smallest fraction of gradient updater, for which coordinate still converges is $\frac{1}{64}$ of the total update. Meaning, we can update only a rectangle of 48^2 pixels, or an image patch of 72 arc-seconds in size. However, it is obvious from the Figure 25 that coordinate descent needs more major cycles to converge with such an extreme approximation.

With less extreme approximations, we also reduce the number of necessary major cycles. The approximation of $\frac{1}{32}$ needs two more major cycle, while all other need the one more as the original method[3]. The objective values of the approximations end within 0.03% of the original(the objective value of the approximations are higher by a factor of 0.0003).

Figure 25 also shows two speedup comparisons. The speedup from the approximation is harder to quantify. For one, each iteration of coordinate descent becomes cheaper, because we only update a small part of the gradient map. This is the per iteration speedup. The second speedup Figure compares the overall time spent in deconvolution. Although each iteration becomes cheaper with the approximation, we may need more iterations to converge. Also, we have an implicit path regularization in the approximation. Therefore, a speedup per coordinate descent iteration does not necessarily lead to an overall speedup.

[3] 6 cycles to converge, and one extra cycle which is here to measure the final objective

As we discussed in section In Section 5.4, we have a limit in each major cycle to how far we can trust our approximation. That is why in the first major cycle coordinate descent gets started with a higher λ regularization than specified. In each successive major cycle, we reduce the λ parameter until we reach the specified value. This is an implicit form of path regularization, and may help speeding up the convergence rate of coordinate descent in general [22]. In our case, we need path regularization to ensure convergence over several major cycles (Aggressive approximations like $\frac{1}{64}$ start to oscillate over several major cycle). Notice that the overall speedup in Figure 25 is lower than the speedup we get per iteration. This suggests path regularization as implemented here does not speed up convergence rate.

Overall with gradient update approximations give us a speedup factor of around 1.5. Depending on how aggressive we approximate, We need one or two more major cycles than the original coordinate descent deconvolutions. This puts gradient update approximations on par with multi-scale CLEAN in terms of major cycles.

6.3.2 Method 2: Approximate deconvolution

In this method, we use a fraction of the total PSF for deconvolution. This method solves a different deconvolution problem, where the PSF is for example only $\frac{1}{8}$ the size. The downside is that method 2 is not guaranteed to converge to the same optimum. Nevertheless, we solve the approximate deconvolution problem with several different fractions of the PSF and compare how close the approximate solution is to the original.

As before, we measure the true objective value for the approximate solution at the beginning of each major cycle iteration. The Figure 26 compares the approximate deconvolution to the original.

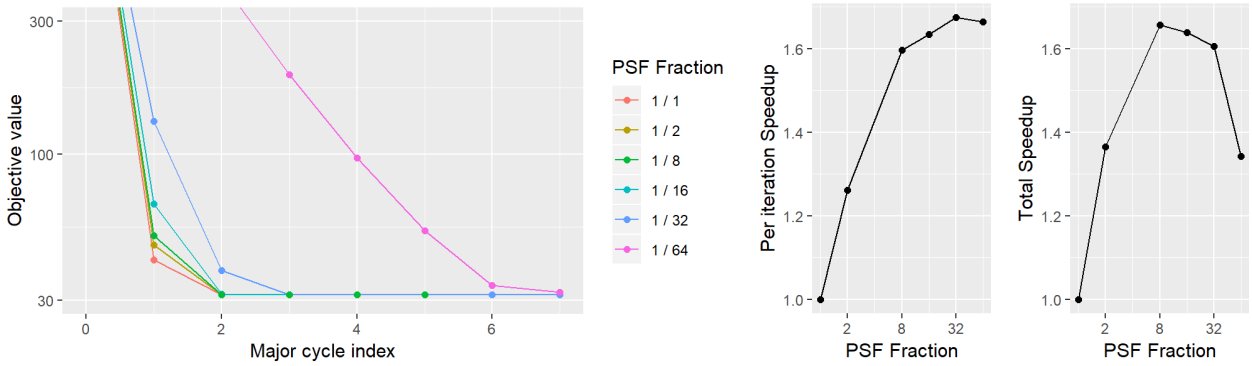


Figure 26: Effect of the L1 and L2 Norm separately.

The performance is similar to the first method at first glance. Both methods converge in a similar manner, with a similar factor of speedup. For the PSF fraction $\frac{1}{64}$ this is largely due to the same path regularization used in this method. For larger PSF's, starting from $\frac{1}{16}$, the path regularization becomes less important with this method, as it only effects the regularization parameter λ of major cycle index 0. Indeed, this method is less likely to oscillate and seems to have a more stable convergence.

The approximate deconvolution converges faster than the first method. At the start of the same major cycle, this method always has a lower objective. However, it is not guaranteed to converge to the same result. This can be seen in the fact that it never reaches the same objective value as the original. The objective value of the approximation is within 0.07% (Factor of 0.0007, roughly twice the factor of the first method). The difference gets more significant, the more extreme we chose the deconvolution approximation.

Nevertheless, the approximate deconvolution converges surprisingly close to the original solution. The question is, is this difference of 0.07% in the objective value significant? The answer to this question depends on what kind of error the approximation introduces in the image. The obvious error is that the approximation chronically under-estimates the pixel values: The maximum pixel value in N132 of the original is 0.0024

Jansky/beam, while the maximum of the $\frac{1}{16}$ approximation is 0.00235 Jansky/beam. The pixel magnitude is important in the self-calibration regime [7] (When we take the result of the reconstruction and try to improve the calibration).

The under-approximation of pixel values is an error we cannot ignore. One naive remedy is to start with the deconvolution approximation, but switch to the original PSF after a certain number of major cycle. This would give us the guarantee to converge to the same result, but let us use an approximation at the start. We propose another solution: Combining the two approximation methods.

6.3.3 Combination of Method 1 and 2

The two approximation methods have two different shortcomings: Approximate gradient update (method 1) converges more slowly, and converges to the same result as the original. Approximate deconvolution (method 2) converges faster than method 1, but does not converge to the same result. Here, we combine the two methods to remedy the shortcomings: For the first couple of major cycles, we use approximate deconvolution, and then switch to approximate gradient update.

We compare the original, approximate gradient update, approximate deconvolution and our combination in. We use the factor $\frac{1}{16}$ for approximations, meaning we update $\frac{1}{16}$ of the gradients, and do the approximate deconvolution with $\frac{1}{16}$ of the PSF.

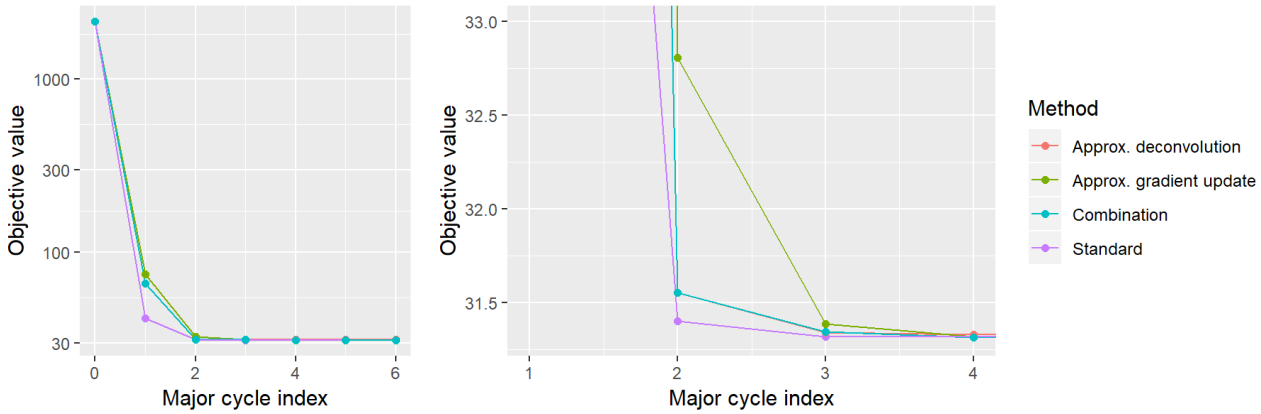


Figure 27: Comparison of the two methods using the fraction $\frac{1}{16}$ of the PSF.

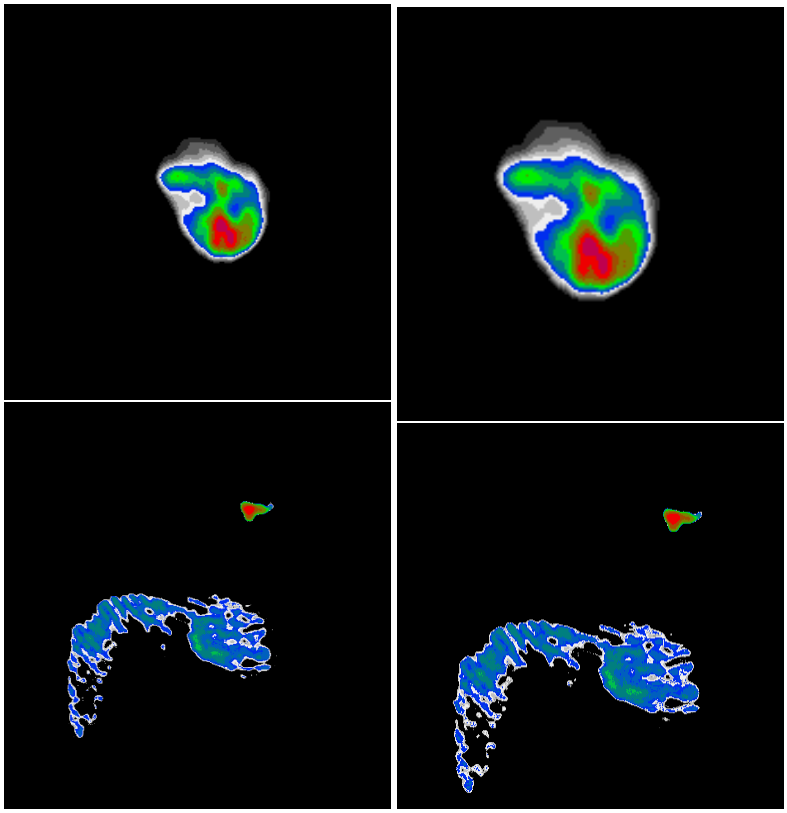
Figure 27

We switch from approximate deconvolution to gradient update after the path regularization has reached the target λ for the first time. In the case of the LMC data, the implementation uses approximate deconvolution for major cycle index 0 and 1, and then switches to approximate gradient update.

Method	Iteration Speedup	Total Speedup
Original	1.00	1.00
(Method 1) Approx. gradient update	1.66	1.55
(Method 2) Approx. deconvolution	1.67	1.68
Combination	1.69	1.68

Speedup. Combination is the fastest overall and leads to the lowest objective function. Actually beats the original by a small margin. Approximation errors are not correlated. We can use Method 1 to remove the approximation errors of Method 2.

Figure 28 compares the original result with the result of the combined approximation. Virtually identical. Same problem with calibration. Only visual difference in the extended emission with calibration errors.



(a) Original CD *(b) Combined approximation*

Figure 28: Image comparison to approximation

Difference is negligible. We can use the approximation and arrive at the same result. Speedup of a factor 1.6
How many major cycles: One more than the original. But same as multi-scale CLEAN.

7 Parallel coordinate descent methods

We demonstrated that the true PSF of the deconvolution problem can be approximated. A smaller PSF can be used, which is only a fraction of the true size. The serial coordinate descent methods, which was used so during this project, achieves a moderate speedup with the PSF approximation. Parallel coordinate descent methods may benefit more from the PSF approximation. For the rest of this work, we develop a parallel coordinate descent deconvolution algorithm. We explain why parallel methods should benefit more from the PSF approximation, and show that they indeed speed up the deconvolution problem.

Introduce the principle of parallel coordinate descent methods, and derrive a parallel coordinate descent deconvolution algorithm.

7.1 From serial to parallel

Parallel coordinate descent methods take several steps at different coordinates before they update the gradient map. The difficulty in parallel coordinate descent methods lies in dealing with correlated pixel values¿ In our deconvolution problem, two pixels which are close to each other in the image are correlated. The more they overlap, the more we over-estimate their pixel values with parallel coordinate descent methods[4]. This over-estimation leads to slow convergence. Or if we increase the number of parallel updates, can lead to a divergence. This means we cannot simply modify our serial coordinate descent algorithm to take a number of parallel steps. We need a way to deal with the over-estimation.

Our serial coordinate descent algorithm uses a greedy pixel selection strategy. Parallel coordinate descent methods on the other hand often select their pixels uniformly at random. The random selection strategy lets us estimate how much the parallel algorithm over-estimates the pixel values[26]. However, a random strategy is not efficient for our deconvolution problem.

We first explain the parallel coordinate descent method in the next section 7.2. We then introduce the modifications we developed for an efficient parallel deconvolution algorithm.

7.2 Parallel (Block) Coordinate Descent Method (PCDM)

The PCDM algorithm can be seen as a generalization of our serial coordinate descent algorithm from Section 4. The serial algorithm optimizes a single pixel at a time. PCDM can update one, or a whole block of pixels at each iteration. And as the name implies, it can update multiple blocks of pixels in parallel. In this project, we use the accelerated variant of the PCDM algorithm, named APPROX [27]. We first introduce a PCDM deconvolution algorithm and then describe the accelerated variant.

7.2.1 Block coordinate descent deconvolution

Here we introduce the serial block coordinate descent algorithm. Instead of optimizing a single pixel in each iteration, the serial block coordinate descent algorithm can update a block of pixels. The block size is left for the user to define. It can be any number between a single pixel, in which case the algorithm is identical to the serial coordinate descent algorithm from Section 4, or the whole image.

Remember the single pixel update from the serial coordinate descent algorithm:

$$pixel_{opt} = \frac{max(gradient_{location} - \lambda\alpha, 0)}{Lipschitz_{location} + (1-\alpha)\lambda} \tag{7.1}$$

[4] Imagine we optimize two pixels next to each other in serial coordinate descent: The serial coordinate descent algorithm optimizes he first pixel, updates the gradient, and then optimizes the second pixel. The value of the second pixel will be magnitudes lower than the first, because most of the emission in that area was already explained by the first pixel. If however we update in parallel, both pixels will end up with a similar value, and both pixels try to explain the same emission.

We optimize the pixel at the current location by taking the gradient and dividing it by the Lipschitz constant. For the serial block coordinate descent algorithm we vectorize the update rule. That means $gradient_{location}$ and $Lipschitz_{location}$ and the output $pixel_{opt}$ become vectors:

$$pixels_{opt} = \frac{max(gradients_{locations} - \lambda\alpha, 0)}{Sum(Lipschitz_{locations}) + (1-\alpha)\lambda} \tag{7.2}$$

This is the serial block coordinate descent update rule. Note that we divide the gradient for each pixel by the the block Lipschitz constant (which is the sum of every pixel Lipschitz constant in the block). The single pixel update rule can take a larger step, but only for a single pixel.

The reader might be familiar with the (F)ISTA method[28]. The block update shown in equation (7.2) is related to the (F)ISTA update step. When the block size equal to the image size (we update all pixels in the image in each iteration), then the serial block coordinate descent is equivalent to (F)ISTA.

7.2.2 Parallel block coordinate descent deconvolution

We present the parallel block coordinate descent algorithm. It is based on the PCD method[26]. In this section, we show how it can be used to solve the deconvolution problem. Our parallel coordinate descent algorithm updates t random blocks of pixels in parallel in each iteration. Each iteration is split into three steps: Step 1 is to select t unique blocks of pixels uniformly at random (we cannot select the same block multiple times). In step 2 we update in parallel each selected block in the reconstructed image x. And finally in step 3 we update the gradient map.

```
dirty = IFFT(GridVisibilities(visibilities))
residualsPadded = ZeroPadding(dirty)

psfPadded = ZeroPadding(PSF)
psfPadded = FlipUD(FlipLR(psfPadded))
gradientUpdate = iFFT(FFT(ZeroPadding(PSF)) * FFT(psfPadded))

x = new Array[,]
gradientsMap = iFFT(FFT(residualsPadded) * FFT(psfPadded))
lipschitzMap = CalcLipschitz(PSF)

objectiveValue = 0.5* Sum(residuals * residuals) + ElasticNet(x)
eso = ESO(CountNonZero(PSF), t, x.Length / blockSize)

do
  oldObjectiveValue = objectiveValue

  //Step 1: select t blocks uniformly at random
  blocks = sample(t)

  //Step 2: update reconstruction
  diffBlocks = new Array
  parallel for each block in blocks
    blockLipschitz = Sum(GetBlock(LipschitzMap, block))

    //increase blockLipschitz according to the ESO
    blockLipschitz = blockLipschitz * eso
    gradientsBlock = GetBlock(gradientMap, block)
    oldBlock = GetBLock(x, block)
    tmp = gradientsBlock + oldBlock * blockLipschitz
    optimalBlock = Max(tmp - lambda*alpha) / (blockLipschitz + (1 - alpha)*lambda)
```

```
32      diffBlock = optimalBlock - oldBlock
33
34      x[block] += diffBlock
35      diffBlocks[block] = diffBlock
36
37    //Step 3: Update gradients
38    for each block in blocks
39      diffBlock = diffBlocks[block]
40      for each pixel in block
41        diff = diffBlock[pixel]
42        shiftedUpdate = Shift(gradientUpdate, pixelLocation)
43        gradientMap = gradientMap - shiftedUpdate * diff
44
45  while maxAbsDiff < epsilon
```

The parameter t can be thought of as the number of processors. We select a block to optimize in parallel for each available processor. The parallel algorithm presented here is a synchronous implementation. Each processor waits for the others to finish in each step. The implementation used later in this section is asynchronous, where each processor separately selects a block, updates the reconstruction and updates the gradient map independent of the other processors.

The core of the parallel coordinate descent algorithm is the Estimated Seperability Overapproximation (ESO). In essence, the ESO represents how much we over-estimate the pixels values when we update t random blocks in parallel. To guarantee convergence, we decrease the step size by the factor of the ESO. With more processors t involved, we have to take smaller and smaller steps to guarantee convergence. Here is a trade-off between the degree of parallelism and the overall convergence speed.

The ESO is derived from three components: The uniform sampling strategy used, the number of parallel updates, and the number of non-zero components in the PSF. We use a t-nice uniform sampling. The ESO that arises from the t-nice sampling (take t blocks uniformly at random) according to[26]:

$$ESO(\omega, t, n) = 1 + \frac{(\omega - 1)(t - 1)}{max(1, n - 1)} \tag{7.3}$$

Where ω is the number of non-zero entries in the PSF, t is the number of parallel updates, and n is the number of blocks in the problem. For example: The image is 256^2 pixels in size, we update blocks with a size of 4^2 pixels, the PSF has $\omega = 24$ non zero entries, and we use $t = 4$ processors, then the ESO is:

$$ESO(\omega = 24, t = 4, n = (256^2/4^2)) = 1 + \frac{(24 - 1)(4 - 1)}{max(1, 4096 - 1)} \approx 1.017 \tag{7.4}$$

The lowest possible value for the ESO is 1. The fewer processors t we use and the fewer non-zero components ω the PSF has, the closer the ESO is to 1. We want a small ESO with the highest number of processors possible. As we see from equation (7.3), the ESO gets smaller with fewer non-zero values in the PSF.

Remember that the PSF in radio astronomy is typically dense. Although most of its values are close to zero, it generally does not have any zero values. The PSF approximation methods we developed in Section 5 effectively reduce the number of non-zero values. For the parallel coordinate descent algorithm, this leads to an ESO closer to 1, even and we can take larger steps without diverging.

7.3 Accelerated parallel block coordinate descent method

We introduced the parallel coordinate descent algorithm in the previous section. In this section we extend the previous algorithm with gradient acceleration, similar to the APPROX method [27].

Instead of using a single gradient map and a single reconstructed image x variable, we an 'explore' and 'correction' variable of both. The algorithm uses an $xExplore$, $xCorrection$ and $gradientMapExplore$, $gradientMapCorrection$. Intuitively, the 'correction' variables contain the accelerated part of the algorithm, and the 'explore' variables the standard part.

We introduce the acceleration variable $theta$, and arrive at the following accelerated, parallel block coordinate descent algorithm:

```
dirty = IFFT(GridVisibilities(visibilities))
residualsPadded = ZeroPadding(dirty)

psfPadded = ZeroPadding(PSF)
psfPadded = FlipUD(FlipLR(psfPadded))
gradientUpdate = iFFT(FFT(ZeroPadding(PSF)) * FFT(psfPadded))

xExplore = new Array[,]
xCorrection = new Array[,]
gradientsMapExplore = iFFT(FFT(residualsPadded) * FFT(psfPadded))
gradientMapCorrection = new Array[,]
lipschitzMap = CalcLipschitz(PSF)

objectiveValue = 0.5* Sum(residuals * residuals) + ElasticNet(x)
eso = ESO(CountNonZero(PSF), t, x.Length / blockSize)
theta0 = t / (x.Length / blockSize)
theta = theta0

do
  oldObjectiveValue = objectiveValue

  //Step 1: select t blocks uniformly at random
  blocks = sample(t)

  //Step 2: update reconstruction
  diffBlocks = new Array
  parallel for each block in blocks
    blockLipschitz = Sum(GetBlock(LipschitzMap, block))

    //increase blockLipschitz according to the ESO
    blockLipschitz = blockLipschitz * eso
    oldBlock =
    tmp = theta^2 * GetBlock(gradientsMapCorrection, block) + GetBlock(
        gradientsMapExplore, block) + GetBLock(xExplore, block) * blockLipschitz
    optimalBlock = Max(tmp - lambda*alpha) / (blockLipschitz + (1 - alpha)*lambda)
    diffBlock = optimalBlock - oldBlock

    xExplore[block] += diffBlock
    xCorrection[block] += diffBlock * (-(1.0f - theta / theta0) / theta^2)
    diffBlocks[block] = diffBlock

  //Step 3: Update gradients
  for each block in blocks
    diffBlock = diffBlocks[block]
    for each pixel in block
```

```
45          diff = diffBlock[pixel]
46          shiftedUpdate = Shift(gradientUpdate, pixelLocation)
47
48          gradientsMapExplore = gradientsMapExplore - shiftedUpdate * diff
49          gradientsMapCorrection = gradientsMapCorrection - shiftedUpdate * diff *
                (-(1.0f - theta / theta0) / theta^2);
50
51      theta = (Sqrt((theta^2 * theta^2) + 4 * (theta^2)) - theta^2) / 2.0f;
52
53  while maxAbsDiff < epsilon
54
55  output = new float[,]
56  for(i in in Range(0, dirty.Length(0))
57    for(j in in Range(0, dirty.Length(0))
58    output[i, j] = theta * xCorrection[i, j] + xExplore[i, j];
```

The accelerated variant takes larger steps towards the optimum during deconvolution. As such, it should need fewer iterations to converge than the non-accelerated variant. Note that the accelerated deconvolution algorithm reduces to the normal parallel deconvolution algorithm if we skip the $theta$ update. Then, the 'correction' variables stay zero over all iterations.

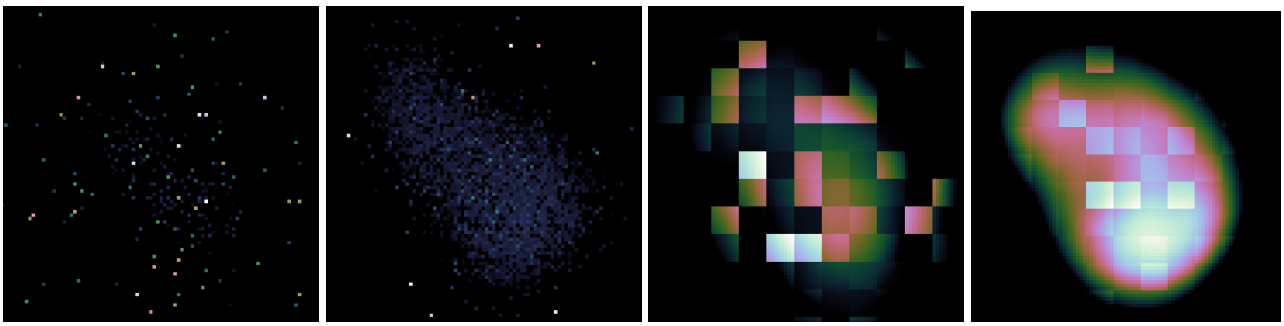
The acceleration comes with a cost attached: Instead of a single gradient map and a single reconstructed image, the acceleration algorithm updates two of each. It is not clear whether the added costs of maintaining the 'explore' and 'correction' variables outweigh the benefit we receive with acceleration. As we will see later, the non-accelerated variant is actually faster on the LMC observation than the accelerated variant.

7.4 Asynchronous implementation

Asynchronous implementation with compare exchange. gradient map gets updated asynchronously. All processors do their deconvolution asynchronously, independent of the other processors. This needs synchronization.

7.5 The problem with random selection for deconvolution

Both algorithms, the accelerated and non-accelerated variant as presented here do not perform well on the LMC dataset. The reason lies in the random selection strategy: In the first few iterations, the deconvolution algorithm selects blocks at random, and tries to explain the whole emission in that area. In short, the first iterations always over-estimate the block-values.



(a) 8k iterations *(b) 80k iterations* *(c) 8k iterations, 8^2 block* *(d) 80k iterations, 8^2 block*

Figure 29: Random parallel deconvolutions on the LMC N132D supernova remnant.

The Figure 29 shows the behaviour on the LMC observation. The reconstructions receive obvious artifacts

from the random selection strategy. The blocks, which get selected in the first few iterations, keep their over-estimated values until we randomly select them again in later iterations. Other blocks in the neighborhood cannot be changed to a reasonable value until the algorithm has randomly selected the over-estimated blocks. That is why even after 80k iterations, the N132D supernova remnant gets only hinted at in Figure 29b. Until the over-estimated blocks get selected again, the algorithm cannot do useful updates in that region.

This behavior is pronounced when we choose a block size of one pixel (i.e. we do not group pixels into blocks). Increasing the block size also increases our changes to select one of the over-estimated block again. But as we see in Figure 29d, the same problem exists with larger block sizes, although less pronounced. After 80k iterations the N132D supernova remnant is visible, but a few random blocks still contain too much of the emission in that area.

The order in which we select blocks seems to be relevant in the deconvolution problem. A random selection strategy needs a prohibitive large number of iterations to converge. But we cannot simply switch out the selection strategy. The random selection strategy is at the core of the Parallel coordinate descent methods. Remember the ESO arises from the fact that we select tau pixels uniformly at random. When we select τ-pixels with a greedy strategy, we might break the ESO and may not converge.

To solve this behavior, we introduce the pseudo-random selection strategy: We select a block at random, but greedily search in the neighborhood for the optimal block to optimize. The size of the neighborhood can be defined by the user. It is essentially a mix between a greedy and a random selection strategy. If we choose the neighborhood to be the whole image, we arrive at a greedy strategy. If we choose the neighborhood to be just one block, we are back at a random strategy. The mixture of the greedy and random strategy allows us to fix the problems with the pure random

We also introduced three heuristics that speed up the parallel deconvolution algorithm in practice: An active set heuristic, Restarting heuristic and a 'Minor' cycle.

7.5.1 Active set heuristic

The active set heuristic is typically used in cyclic coordinate descent: It chooses a subset of blocks, and optimizes the set until it converges. Then it chooses a new set. We use the active set heuristic together with our pseudo-random selection strategy. A large portion of the blocks in the image will be zero. If we select blocks at pseudo-random, we are likely to select a block that will never contain non-zero values. The active set heuristic increases the likelihood that the pseudo-random strategy selects a relevant block.

At the start of the parallel deconvolution algorithm, we iterate over all blocks. We add all blocks whose value can be changed to non-zero. During parallel iterations, the algorithm only selects blocks from the active set.

Note that the algorithm only adds blocks to the active set, which can be changed to a non-zero value at the start of deconvolution. Over several iterations, there may be blocks that are not in the active set, but are part of the optimal solution. This is remedied with a restarting heuristic.

7.5.2 Restarting heuristic

In accelerated gradient methods, like APPROX or (F)ISTA, restarting the acceleration can lead to a significant speedup[29]. In our accelerated variant, restarting means we reset the acceleration variable $theta$ restart with the deconvolution, using the intermediate solution from the last iteration. Our accelerated deconvolution algorithm uses an active set. If we detect that important blocks are missing from the active set, we want to restart the deconvolution with a new active set.

We implemented two restarting heuristics: One heuristic is based on Glasmachers et al.[30] and restarts the algorithm when the acceleration likely benefits from it. The other heuristic was developed by us and restarts

the when the active set is likely to be missing blocks. In our tests, we always needed to restart due to the active set, and never due to the heuristic by Glasmachers.

Our own restarting heuristic is based on the following idea: When the parallel deconvolutions start to converge before the currently best greedy step, we restart the algorithm with a new active set.

We introduce a new loop for the active set iteration. For each active set iteration, we execute several parallel deconvolution steps. For example, a single active set iteration may include 1000 parallel deconvolutions for each processor, running for $\tau * 1000$ iterations in total. Each processor updates the gradient maps and the reconstructed image asynchronously. We assume that after each active set iteration, we have landed at least once on the block that the greedy algorithm would have chosen in a single iteration.

At the end of each active set iteration, we check whether the parallel accelerated iterations converge faster than the currently best greedy step. If it does, we restart the parallel algorithm with a new active set.

```
...
do
  lastMaxDiff = GetGreedyMaxBlockDiff(gradientsMapExplore, xExplore)
  parallelDiffFactor = 0
  for activeSetIteration in activeSetIterations

    //concurrent iterations
    maxParallelDiff = 0
    parallel for
      ...
      diffBlock = optimalBlock - oldBlock
      ...

      if(maxParallelDiff < diffBlock)
        maxParallelDiff = diffBlock

    //restarting heuristic
    if(parallelDiffFactor = 0)
      parallelDiffFactor = lastMaxDiff / maxParallelDiff

    currentMaxDiff = GetGreedyMaxBlockDiff(gradientsMapExplore, xExplore)
    activeSetInvalid = lastAbsMax / maxParallelDiff > parallelDiffFactor * 2
    activeSetInvalid = activeSetInvalid | currentMaxDiff > lastAbsMax & lastAbsMax /
        parallelDiffFactor > concurrentFactor
    if activeSetInvalid
      Restart()
      parallelDiffFactor = 0
    lastMaxDiff = currentMaxDiff
  ..
while maxAbsDiff < epsilon
```

After the first active set iteration, we check how close the maximum parallel update is to the best greedy step. The ratio of maximum greedy update and maximum parallel update should stay similar over the course of the algorithm, if the active set is valid. If the active set invalid, if it is missing important blocks, the algorithm will encounter ever smaller values for $maxParallelDiff$, while $lastMaxDiff$ does not decrease significantly over the active set iterations. In that case, the ratio between $lastMaxDiff/maxParallelDiff$ sharply increases, and we restart the algorithm.

We have two similar conditions on which we flag the active set as invalid. If the ratio of $lastMaxDiff/maxParallelDiff$ is twice as big as the ratio of the first active set iteration, we restart the algorithm. The second condition is based on the observation that the $currentMaxDiff$ may increase from active set iteration to active set iter-

ation. In this case, we are likely missing important blocks in the active set and we restart more aggressively: If the ratio $lastMaxDiff/maxParallelDiff$ bigger than the ratio of the first active set iteration.

7.5.3 Re-introduction of a 'Minor' cycles

As we will demonstrate in the next section, the parallel coordinate descent deconvolution algorithm benefits significantly from our PSF approximation. The drawback of our PSF approximation is that it needs more major cycles to converge. We re-introduce a similar minor cycle to the Clark CLEAN algorithm [13], and reduce the number of necessary major cycles.

The CLEAN algorithm developed by Clark also uses only a fraction of the PSF during CLEAN deconvolutions. After a number of iterations, the residuals of the Clark algorithm are inaccurate, and it resets the residuals with the full PSF.

We use a similar idea: We run our parallel coordinate descent deconvolution algorithm, and retrieve the intermediate solution. We then decide whether we reset the residuals using the full PSF (the 'minor' cycle), or we use the major cycle.

Resetting the residuals with the full PSF is done as follows:

```
residuals = iFFT(Gridding(visibilities))    //Major cycle
x = deconvolve(residuals, Cut(PSF))     //Deconvolve with approximate PSF
residuals_minor = residuals - iFFT(FFT(x) * FFT(PSF))
```

We convolve the intermediate solution x with the PSF in Fourier space, and subtract the result from the original residuals from the major cycle. This allows us to use fewer major cycles when we use an aggressive PSF approximation.

The question only question that remains is, when to use a major cylce or a 'minor' cycle. Remember from Section 5, we introduced a heuristic based on the PSF sidelobe: When we deconvolve using only a fraction of the full PSF, we leave sidelobes in the residual image. In each major cycle, we can only run the deconvolution algorithm up to a certain point, before we include PSF sidelobes in the reconstructed image.

In Section 5 solved this by estimating a minimum regularization parameter λ for each major cycle. Now with the addition of a 'minor' cycle, we use the same heuristic twice: We have two minimum regularization parameters, λ_{minor} and λ_{major}. We use the minor cycle, as long as λ_{minor} is larger than λ_{major}. Otherwise, we start a new major cycle. In total, the major and 'minor' cycle for our parallel coordinate descent algorithm is implemented as follows:

```
residualVis = visibilities
x = new Array[,]

for each cycle in Range(0, maxMajorCycles)
  residuals = iFFT(Gridding(residualVis))
  residualsMinor = residuas

  lambdaMajor = Estimate(residuals, PSF, 2)
  lambdaMinor = 0

  do
    lambdaMinor = Estimate(residualsMinor, PSF, psfFraction)
    lambdaMinor = Max(lambdaMinor, lambdaMajor)

    x_current = deconvolve(residuals, Cut(PSF, psfFraction))
    x += x_current
    residuals_minor = residuals - iFFT(FFT(x) * FFT(PSF))
```

```
while(lambdaMajor < lambdaMinor)

modelVis = DeGridding(FFT(x))
residualVis = visibilities - modelVis
```

Estimating λ_{minor} is identical to the heuristic developed in Section 5. We use the maximum sidelobe of the PSF which is not contained in the cutout used for deconvolution. Estimating λ_{major} is more difficult: In theory, we use the full PSF, and do not have a sidelobe outside the cutout. However, we also know that the full PSF is only an approximation (remember: the w-term changes the PSF slightly for all pixels in the image). For the parameter λ_{major}, we simply use the maximum sidelobe outside of the $\frac{1}{2}$ cutout.

7.6 Tests on the LMC Observation

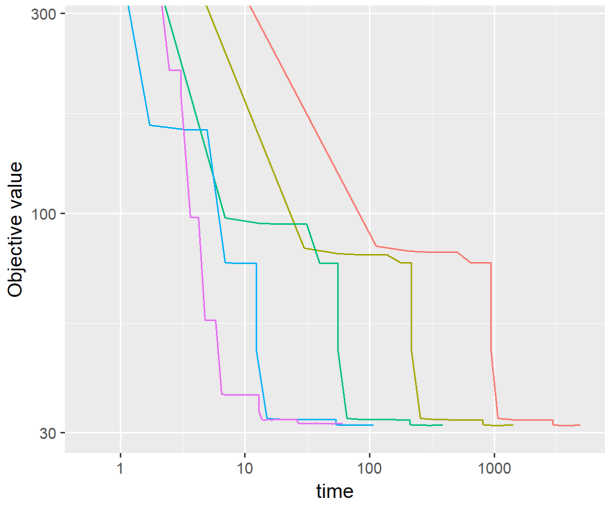
We have shown the parallel coordinate descent algorithm in two variants: With gradient acceleration and without. We developed several heuristics, which includes a set of tuning parameters. In this section, we test our parallel coordinate descent algorithm on the MeerKAT LMC observation. We test the tuning parameters, and show how much our algorithm benefits from the PSF approximation method.

In this test, we measure the wall-clock time of only the deconvolution. The wall-clock time of the Major- and 'Minor' cycle are excluded. Again, we calculate the value of the objective function. We compare different parameters of our parallel coordinate descent algorithm, trying to get the shortest convergence time.

7.6.1 PSF approximation

First, we test the effect of our PSF approximation method. We test different PSF cutouts of our parallel coordinate descent algorithm with gradient acceleration. We use $\tau = 8$ processors with our asynchronous implementation.

Figure 30 shows the result of our parallel coordinate descent algorithm with different PSF fractions, combined with the ESO and the total seconds needed to converge. The y-axis shows the objective value, and the x-axis shows the elapsed seconds. Note that the axis of the figure are logarithmic. The 'kinks' in the lines are due to either the Major or the 'Minor' cycle resetting the residuals (remember: we do not include the wall-clock time of the Major or 'Minor' cycle in this test).



PSF	ESO	Total seconds
1 / 4	1.437	4941
1 / 8	1.109	1433
1 / 16	1.027	390
1 / 32	1.007	108
1 / 64	1.002	61

Figure 30: Convergence times with PSF approximation

Clearly, the parallel coordinate descent algorithm benefits from our PSF approximation method. If we use $\frac{1}{32}$ of the full PSF, the parallel coordinate descent algorithm spends a total of 108 seconds in to deconvolve the image. For comparison, the serial coordinate descent algorithm with PSF approximation takes roughly 1400 seconds, or 23 minutes.

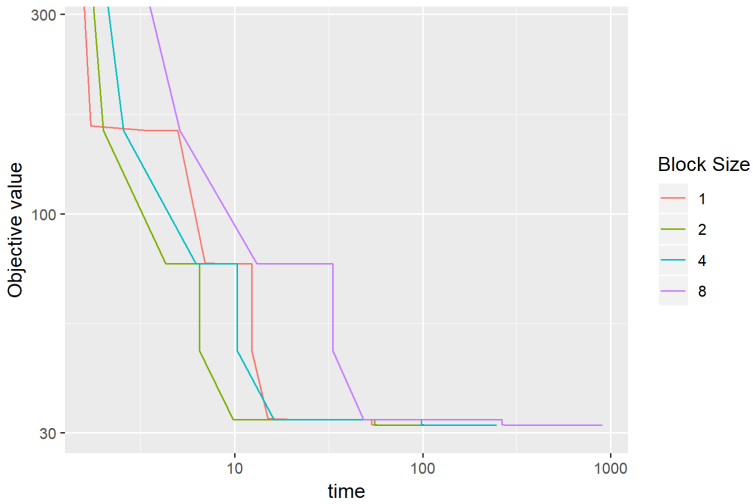
As expected, with increasing PSF approximation, we get an ESO which is ever closer to 1. Remember: An ESO of 1 means that each processor in the parallel coordinate descent algorithm can has the same step size as the serial coordinate descent algorithm. With an approximation of $\frac{1}{32}$ and 8 parallel processors, the ESO is only marginally larger than 1. This suggests that the parallel coordinate descent algorithm may be sped up further with additional processors.

Note that the speedup from fraction $\frac{1}{4}$ to $\frac{1}{8}$ is roughly a factor of 3.5. The same holds true for the speedup of $\frac{1}{8}$ to $\frac{1}{16}$, and $\frac{1}{16}$ to $\frac{1}{32}$. The PSF cutout from $\frac{1}{8}$ to $\frac{1}{16}$ four times fewer pixels. This suggest the speedup may be due to the reduced conflicts in the asynchronous update of the gradient map. With a smaller PSF we reduce the change of several threads updating the same location in the gradient map, and we spend more time in the deconvolution itself.

For the rest of this project, we will use a PSF approximation of $\frac{1}{32}$ for the parallel coordinate descent algorithm. The $\frac{1}{64}$ approximation is faster in this tests, but needs more 'Minor' cycles (Which were excluded from the wall-clock time). Including the time spent in the 'Minor' cycle, the $\frac{1}{32}$ approximation is the fastest overall.

7.6.2 Block size

The parallel coordinate descent algorithm can group several pixels in blocks, and optimize the blocks of pixels in parallel. It is not clear if grouping the pixels in blocks results in shorter convergence times. We test different block sizes in Figure 31, all using the same PSF approximation of $\frac{1}{32}$. We tested out block size of 1^2 (every block contains a single pixel) up to a block size of 8^2:



Block Size	Total seconds
1^2	108
2^2	103
4^2	248
8^2	905

Figure 31: Convergence times with different block sizes

The larger block sizes of 4^2 and 8^2 are significantly slower to converge than a block size of 1^2. Only a block size of 2^2 is slightly faster. Interestingly though, the parallel coordinate descent algorithm is faster to arrive at intermediate results with the block sizes 2^2 and 4^2. This suggests that the parallel coordinate descent algorithm may benefit starting out from larger block sizes, and gradually reducing the block size over several major cycle iterations.

However, two factors lead to the decision to simply use a block size of 1^2 for all major cycle iterations: First, the block size is likely connected to the image resolution. An effective heuristic that starts with larger blocks may

become difficult to develop for general observations. Secondly, the parallel coordinate descent implementation becomes more complicated when it has to account for different block sizes. An implementation with a block size of only 1^2 is shorter and simpler.

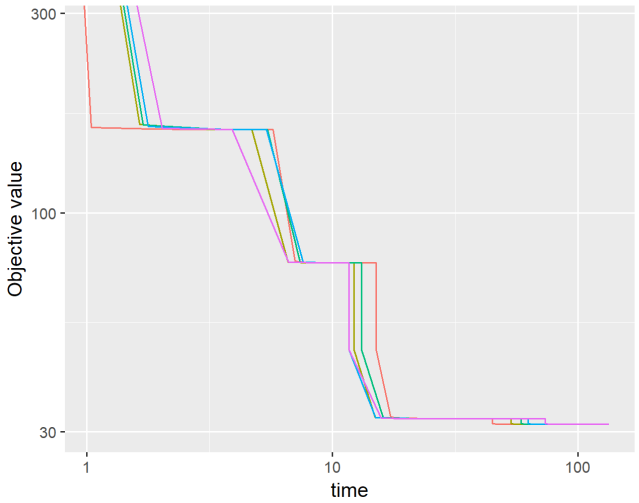
For the rest of this project, we use a block size of 1, meaning every thread is minimizing a single pixel.

7.6.3 Pseudo-random strategy

We discussed in Section 7.5, a random block selection strategy seems to perform badly on the deconvolution problem. We created the pseudo-random strategy, which selects a block at random, but searches in it's neighborhood for the optimal block to optimize. It is a mixture between a random and a greedy selection strategy.

But the pseudo-random strategy introduces a new tuning parameter, which we call the 'search percentage'. A search percentage of 0 says that after a random block has been selected, we search 0% of the neighboring blocks in the active set. It is identical to a pure random selection strategy. A search percentage of 1.0 says that after a random block has been selected, we search 100% of its neighbors in the active set. For example, if we have 256 blocks with 8 threads, a search percentage of 1.0 selects a random block and looks at 32 blocks in its neighborhood. A search percentage of 100% is identical to a greedy strategy, where each thread searches its part of the active set.

Our parallel coordinate descent algorithm needs a search percentage larger than 0. We compare different search percentages in Figure 32.



Search Percentage	Total seconds
0.01	110
0.1	106
0.2	104
0.4	112
0.8	134

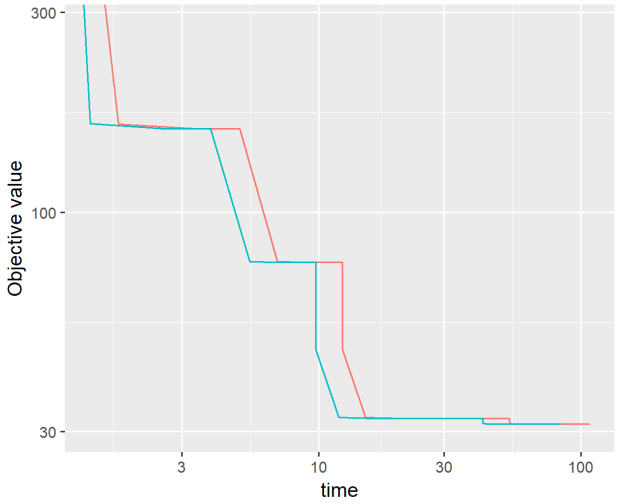
Figure 32: Convergence times with different search percentages.

And it is a tiny fraction. A lot less communication cost. Close to random, but not quite.

7.6.4 Acceleration

Lastly, we test whether the parallel coordinate descent algorithm is faster with or without gradient acceleration. The accelerated variant needs two versions of the gradient map, which get updated asynchronously with compare-exchange operations. Figure 33 compares the accelerated and non-accelerated parallel coordinate descent algorithm.

The accelerated variant is significantly slower in every part of the algorithm. Our hypothesis is that the two gradient maps necessary for the acceleration also increase the cost of synchronization.



Method	Total seconds
With Acceleration	108
Without Acceleration	84

Figure 33: Convergence time with or without gradient acceleration.

Furthermore the accelerated variant has additional run time costs which were not measured in this test: It has to create a copy of the gradient map and the reconstructed image for the acceleration. Meaning this test shown in Figure 33 is biased in favor of the accelerated variant, yet it is still slower.

7.7 Comparison to the serial coordinate descent algorithm

In the previous section, we tested various tuning parameters of the parallel coordinate descent algorithm. In this section, we compare the serial coordinate descent algorithm with the final, simplified parallel coordinate descent algorithm.

We took the lessons from the tests and implemented a simplified parallel coordinate descent algorithm. It does not use gradient acceleration, and cannot group pixels into blocks. Each thread can only minimize a single pixel in parallel in each iteration. As we will see shortly, the simplified parallel coordinate descent implementation is significantly faster than the previous implementation.

In this test, we compare the wall-clock time spent on the deconvolution algorithm. The time we measure here does not include the Major cycle. For the parallel coordinate descent algorithm, this means we also measure the time spent in the 'Minor' cycle for the parallel coordinate descent algorithm (which was excluded in the previous tests).

We use the same hardware as in the tests of Section 19. The table 1 shows the comparison between the serial and parallel coordinate descent algorithm. The parallel coordinate descent algorithm achieved a speedup factor of roughly 20. While the serial coordinate descent algorithm takes over 20 minutes to deconvolve the image, the parallel coordinate descent algorithm takes less than two minutes in total.

Algorithm	PSF Fraction	Major Cycles	Total Seconds in Deconvolution	Speedup factor
Serial CD	$\frac{1}{16}$	4	1486	–
Parallel CD	$\frac{1}{32}$	5	75	≈ 20

Table 1: Speedup comparison of the serial and parallel coordinate descent algorithm. Both algorithms were compared on an Intel Xeon E3-1505M with 8 logical cores.

This parallel coordinate descent implementation is even faster than the implementation tested in the previous Section 7.6. This implementation is simpler because it does not account for different block sizes. Each

processor deconvolves a single pixel in parallel. The simplification leads to another decrease in wall-clock time.

Comparison to CLEAN.

Note however that the serial coordinate descent requires one less major cycle.

Faster than even the previous parallel coordinate descent implementation. Simplified because we do not need to account for blocks of pixels. But needs a major cycle more than the serial coordinate descent algorithm. What was measured: Including the 'Minor' cycle.

Fastest variants of both. Serial coordinate descent

Compare the images from serial coordinate descent to the parallel coordinate descent, shown in Figure 34.

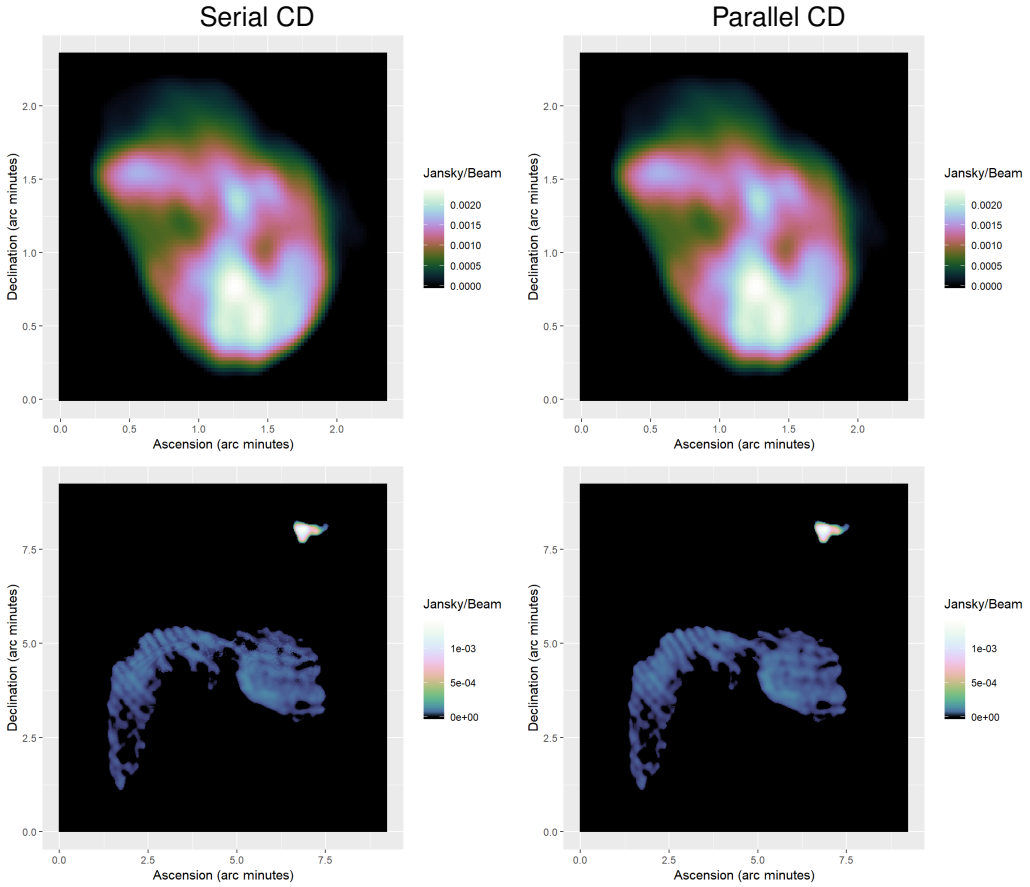


Figure 34: Comparison of the serial and parallel coordinate descent reconstruction of the LMC observation.

Acceleration factor

images

7.8 Scalability of the parallel coordinate descent algorithm

How many processors can we realistically use. At what point does the ESO get too small.

8 Discussion

Main thing: parallel coordinate descent works. Times comparable to standard CLEAN. Reconstruction quality similar, or superior to serial coordinate descent Super-resolution.. One more major cycle. Comparable number of major cycles to CLEAN. Exact comparison is difficult

Due to our gradient approximation scheme. We exploit the fact that the PSF of interferometers like MeerKAT is fairly concentrated around the center. Exploited by the parallel coordinate descent algorithm. Serial coordinate descent was difficult to speedup, even with GPU and distributed reconstruction.

Easy to extend to a distributed setting, hydra.

Expect it to scale better with larger field of views. The larger the field of view, the more concentrated the PSF is in relation to the whole image. It is not clear, whether we introduce a systematic error with the approximation. We couldn't find one.

Serial coordinate descent Similar to the standard CLEAN. In our implementation, it did not benefit a lot from GPU acceleration. Only by a factor of 2. Unclear if this generalizes to multi-scale CLEAN.

CLEAN CLEAN is striclty better with calibration error. Better residuals. Compared to coordinate descent methods, it manages small iteration counts. Our coordinate descent methods achieved super-resolution. Artificial example, but still.

Question about regularization Used the elasticNet regularization. Cheap and easy. More sophisticated,

Multiy frequency.

Works well for MeerKAT. Its PSF is located many not work well for LOFAR.

Difficult to achieve speedup with a dense PSF we approximated

8.1 Multi frequency extension

Difficult.

Regularized inverse problem [31]. Objective function How it works, adding a new term to the objective function

$$\underset{x}{minimize}\ \frac{1}{2}\left\|I_{dirty} - X * PSF\right\|_2^2 + \lambda ElasticNet(X) + \lambda_v \left\|DX\right\|_1 \tag{8.1}$$

Where D is the Discrete cosine transform.

Does not have a proximal operator for each pixel. problem for Coordinate descent method.

Question if each iteration can be cheap.

But may be separated with respect to frequency with Lagrangian multipliers. Question if cd methods are faster.

9 Conclusion

First based on convex optimization algorithm, that is comparable to CLEAN speed.

Whether we can do an approximation and simplify the problem for distribution Yes we can, but not all deconvolution algorithms benefit from it.

We developed our own algorithm that can benefit greatyl from the approximation. It already benefits from single machine processing. From our results, the developed algorithm is comparable to the runtime of CLEAN. CLEAN is known as one of the fastest deconvolution algorithms.

Exploiting sparsity in the problem. But not everything benefits from it. We need optimization algorihtm that exploits sparsity. Also, different regularizations can break sparsity, making the approximation not as effective. Regularization affects the reconstruction quality. As of the time of writing, it is unclear what is the optimal regularization for radio interferometers As of the time of writing, it is currently unclear what regularization has the

Unclear how the algorithm generalizes to other reconstruction problems. We just tested it on the LMC observation. A bunch of heuristics.

Generalization on different instruments. Our approximation scheme exploits the fact that instruments are often nearly a gaussian function. This may not work for instruments with lower frequencies, like LOFAR.

References

[1] Emmanuel J Candès, Justin Romberg, and Terence Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. IEEE Transactions on information theory, 52(2):489–509, 2006.

[2] David L Donoho. Compressed sensing. IEEE Transactions on information theory, 52(4):1289–1306, 2006.

[3] Arwa Dabbech, Alexandru Onose, Abdullah Abdulaziz, Richard A Perley, Oleg M Smirnov, and Yves Wiaux. Cygnus a super-resolved via convex optimization from vla data. Monthly Notices of the Royal Astronomical Society, 476(3):2853–2866, 2018.

[4] Arwa Dabbech, Chiara Ferrari, David Mary, Eric Slezak, Oleg Smirnov, and Jonathan S Kenyon. Moresane: Model reconstruction by synthesis-analysis estimators-a sparse deconvolution algorithm for radio interferometric imaging. Astronomy & Astrophysics, 576:A7, 2015.

[5] JA Högbom. Aperture synthesis with a non-regular distribution of interferometer baselines. Astronomy and Astrophysics Supplement Series, 15:417, 1974.

[6] Urvashi Rau and Tim J Cornwell. A multi-scale multi-frequency deconvolution algorithm for synthesis imaging in radio interferometry. Astronomy & Astrophysics, 532:A71, 2011.

[7] AR Offringa and O Smirnov. An optimized algorithm for multiscale wideband deconvolution of radio astronomical images. Monthly Notices of the Royal Astronomical Society, 471(1):301–316, 2017.

[8] Charles A Bouman and Ken Sauer. A unified approach to statistical tomography using coordinate descent optimization. IEEE Transactions on image processing, 5(3):480–492, 1996.

[9] Simon Felix, Roman Bolzern, and Marina Battaglia. A compressed sensing-based image reconstruction algorithm for solar flare x-ray observations. The Astrophysical Journal, 849(1):10, 2017.

[10] Madison Gray McGaffin and Jeffrey A Fessler. Edge-preserving image denoising via group coordinate descent on the gpu. IEEE Transactions on Image Processing, 24(4):1273–1281, 2015.

[11] Olivier Fercoq, Zheng Qu, Peter Richtárik, and Martin Takáč. Fast distributed coordinate descent for non-strongly convex losses. In 2014 IEEE International Workshop on Machine Learning for Signal Processing (MLSP), pages 1–6. IEEE, 2014.

[12] Bram Veenboer, Matthias Petschow, and John W Romein. Image-domain gridding on graphics processors. In 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 545–554. IEEE, 2017.

[13] BG Clark. An efficient implementation of the algorithm'clean'. Astronomy and Astrophysics, 89:377, 1980.

[14] FR Schwab. Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry. The Astronomical Journal, 89:1076–1081, 1984.

[15] André Ferrari, David Mary, Rémi Flamary, and Cédric Richard. Distributed image reconstruction for very large arrays in radio astronomy. In 2014 IEEE 8th Sensor Array and Multichannel Signal Processing Workshop (SAM), pages 389–392. IEEE, 2014.

[16] Jason D McEwen and Yves Wiaux. Compressed sensing for wide-field radio interferometric imaging. Monthly Notices of the Royal Astronomical Society, 413(2):1318–1332, 2011.

[17] Yu Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. SIAM Journal on Optimization, 22(2):341–362, 2012.

[18] Yu Nesterov. Gradient methods for minimizing composite functions. Mathematical Programming, 140(1):125–161, 2013.

[19] Jonathan S Kenyon. Pymoresane: Python model reconstruction by synthesis-analysis estimators. Astrophysics Source Code Library, 2019.

[20] Marcel Koester. Ilgpu: A modern, lightweight and fast gpu compiler for high-performance .net programs, 2019.

[21] Justin Luitjens. Faster parallel reductions on kepler, 2014.

[22] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. Journal of statistical software, 33(1):1, 2010.

[23] DC-J Bock, MI Large, and Elaine M Sadler. Sumss: A wide-field radio imaging survey of the southern sky. i. science goals, survey design, and instrumentation. The Astronomical Journal, 117(3):1578, 1999.

[24] AR Offringa, Benjamin McKinley, Natasha Hurley-Walker, FH Briggs, RB Wayth, DL Kaplan, ME Bell, Lu Feng, AR Neben, JD Hughes, et al. Wsclean: an implementation of a fast, generic wide-field imager for radio astronomy. Monthly Notices of the Royal Astronomical Society, 444(1):606–619, 2014.

[25] Dan Briggs. High fidelity deconvolution of moderately resolved sources, 2019.

[26] Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. Mathematical Programming, 156(1-2):433–484, 2016.

[27] Olivier Fercoq and Peter Richtárik. Accelerated, parallel, and proximal coordinate descent. SIAM Journal on Optimization, 25(4):1997–2023, 2015.

[28] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM journal on imaging sciences, 2(1):183–202, 2009.

[29] Olivier Fercoq and Zheng Qu. Restarting accelerated gradient methods with a rough strong convexity estimate. arXiv preprint arXiv:1609.07358, 2016.

[30] Tobias Glasmachers and Ürün Dogan. Coordinate descent with online adaptation of coordinate frequencies. arXiv preprint arXiv:1401.3737, 2014.

[31] André Ferrari, Jérémy Deguignet, Chiara Ferrari, David Mary, Antony Schutz, and Oleg Smirnov. Multi-frequency image reconstruction for radio interferometry. a regularized inverse problem approach. arXiv preprint arXiv:1504.06847, 2015.

[32] Luke Pratley, Melanie Johnston-Hollitt, and Jason D McEwen. A fast and exact w-stacking and w-projection hybrid algorithm for wide-field interferometric imaging. arXiv preprint arXiv:1807.09239, 2018.

List of Figures

1 Center of the Milky Way galaxy at radio wavelengths. Observed by the MeerKAT interferometer. 1
2 Example of an image reconstruction problem of radio interferometers . . . 2
3 Radio interferometry system . . . 4
4 Sampling regime of the MeerKAT radio interferometer. . . . 6
5 Example deconvolution problem with two point sources. . . . 9
6 CLEAN deconvolution iterations. . . . 10
7 Blurring with the CLEAN-beam . . . 10
8 Major/Minor cycle . . . 11
9 Effect of the L1 and L2 Norm separately. . . . 14
10 Example problem with two point sources. . . . 15
11 Comparison of the two convolution schemes. . . . 18
12 Sum of squared values for the Lipschitz constant. . . . 19
13 Example of the gradient calculation. . . . 20
14 Example problem with two point sources. . . . 21
15 PSF arising from an increasing number of visibilities. . . . 24
16 Approximation of gradient update. . . . 25
17 Approximate deconvolution with a fraction of the PSF. . . . 26
18 Max sidelobe PSF. . . . 27
19 Section of the Large Magellanic Cloud (LMC) . . . 29
20 Narrow band image section used. . . . 30
21 Comparison of the whole image . . . 31
22 N132 comparison . . . 31
23 Influence of calibration errors . . . 32
24 Speedup by using MPI or GPU acceleration . . . 32
25 Effect of only updating a fraction of the gradients. . . . 33
26 Effect of the L1 and L2 Norm separately. . . . 34
27 Comparison of the two methods using the fraction $\frac{1}{16}$ of the PSF. . . . 35
28 Image comparison to approximation . . . 36
29 Random parallel deconvolutions on the LMC N132D supernova remnant. . . . 41
30 Convergence times with PSF approximation . . . 45
31 Convergence times with different block sizes . . . 46
32 Convergence times with different search percentages. . . . 47
33 Convergence time with or without gradient acceleration. . . . 48
34 Comparison of the serial and parallel coordinate descent reconstruction of the LMC observation. 49
35 The Major Cycle Architecture . . . 57
36 State-of-the-art Compressed Sensing Reconstruction Architecture . . . 57
37 The Major Cycle Architecture of image reconstruction algorithms . . . 59

List of Tables

1 Speedup comparison of the serial and parallel coordinate descent algorithm. Both algorithms were compared on an Intel Xeon E3-1505M with 8 logical cores. . . . 48

10 attachment

11 Larger runtime costs for Compressed Sensing Reconstructions

The MeerKAT instrument produces a new magnitude of data volume. An image with several million pixels gets reconstructed from billions of Visibility measurements. Although MeerKAT measures a large set of Visibilities, the measurements are still incomplete. We do not have all the information available to reconstruct an image. Essentially, this introduces ”fake” structures in the image, which a reconstruction algorithm has to remove. Additionally, the measurements are noisy.

We require an image reconstruction algorithm which removes the ”fake” structures from the image, and removes the noise from the measurements. The large data volume of MeerKAT requires the algorithm to be both scalable and distributable. Over the years, several reconstruction algorithms were developed, which can be separated into two classes: Algorithms based on CLEAN, which are cheaper to compute and algorithms based on Compressed Sensing, which create higher quality reconstructions.

CLEAN based algorithms represent the reconstruction problem as a deconvolution. First, they calculate the ”dirty” image, which is corrupted by noise and fake image structures. The incomplete measurements essentially convolve the image with a Point Spread Function (PSF). CLEAN estimates the PSF and searches for a deconvolved version of the dirty image. In each CLEAN iteration, it searches for the highest pixel in the dirty image, subtracts a fraction PSF at the location. It adds the fraction to the same pixel location of a the ”cleaned” image. After several iterations, the cleaned image contains the deconvolved version of the dirty image. CLEAN accounts for noise by stopping early. It stops when the highest pixel value is smaller than a certain threshold. This results in a light-weight and robust reconstruction algorithm. CLEAN is comparatively cheap to compute, but does not produce the best reconstructions and is difficult to distribute on a large scale.

Compressed Sensing based algorithms represent the reconstruction as an optimization problem. They search for the optimal image which is as close to the Visibility measurements as possible, but also has the smallest regularization penalty. The regularization encodes our prior knowledge about the image. Image structures which were likely measured by the instrument result in a low regularization penalty. Image structures which were likely introduced by noise or the measurement instrument itself result in high penalty. Compressed Sensing based algorithms explicitly handle noise and create higher quality reconstructions than CLEAN. State-of-the-art Compressed Sensing algorithms show potential for distributed computing. However, they currently do not scale on MeerKATs data volume. They require too many computing resources compared to CLEAN based algorithms.

This project searches for a way to reduce the runtime costs of Compressed Sensing based algorithms. One reason for the higher costs is due to the non-uniform FFT Cycle. State-of-the-art CLEAN and Compressed Sensing based algorithms both use the non-uniform FFT approximation in a cycle during reconstruction. The interferometer measures the Visibilities in a continuous space in a non-uniform pattern. The image is divided in a regularly spaced, discrete pixels. The non-uniform FFT creates an approximate, uniformly sampled image from the non-uniform measurements. Both, CLEAN and Compressed Sensing based algorithms use the non-uniform FFT to cycle between non-uniform Visibilities and uniform image. However, a Compressed Sensing algorithm requires more non-uniform FFT cycles for reconstruction.

CLEAN and Compressed Sensing based algorithms use the non-uniform FFT in a similar manner. However, there are slight differences in the architecture. This project hypothesises that The previous project searched for an alternative to the non-uniform FFT cycle. Although there are alternatives, there is currently no replacement which leads to lower runtime costs for Compressed Sensing. Current research is focused on reducing the number of non-uniform FFT cycles for Compressed Sensing algorithms.

CLEAN based algorithms use the Major Cycle Architecture for reconstruction. Compressed Sensing based algorithms use a similar architecture, but with slight modifications. Our hypothesis is that we may reduce the number of non-uniform FFT cycles for Compressed Sensing by using CLEAN’s Major Cycle Architecture.

11.1 CLEAN: The Major Cycle Architecture

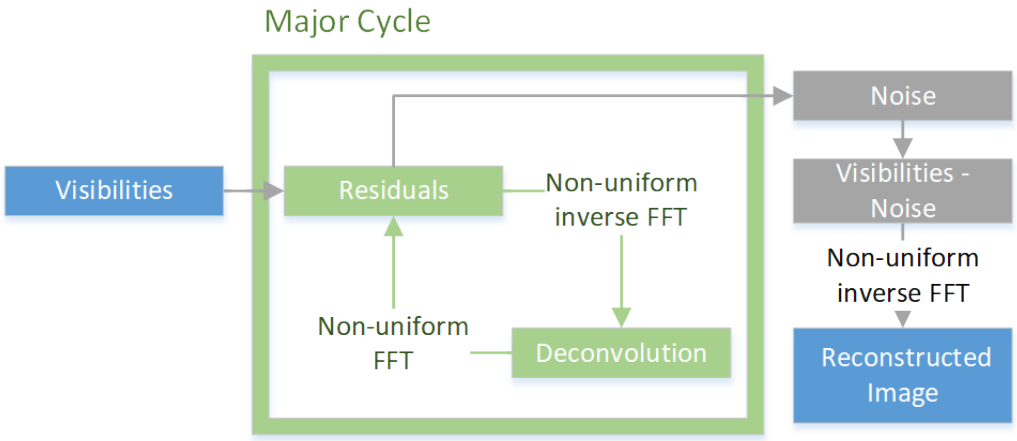


Figure 35: The Major Cycle Architecture

Figure 35 depicts the Major Cycle Architecture used by CLEAN algorithms. First, the Visibilities get transformed into an image with the non-uniform FFT. The resulting dirty image contains the corruptions of the measurement instrument and noise. A deconvolution algorithm, typically CLEAN, removes the corruption of the instrument with a deconvolution. When the deconvolution stops, it should have removed most of the observed structures from the dirty image. The rest, mostly noisy part of the dirty image gets transformed back into residual Visibilities and the cycle starts over.

In the Major Cycle Architecture, we need several deconvolution attempts before it has distinguished the noise from the measurements. Both the non-uniform FFT and the deconvolution are approximations. By using the non-uniform FFT in a cycle, it can reconstruct an image at a higher quality. For MeerKAT reconstruction with CLEAN, we need approximately 4-6 non-uniform FFT cycles for a reconstruction.

11.2 Compressed Sensing Architecture

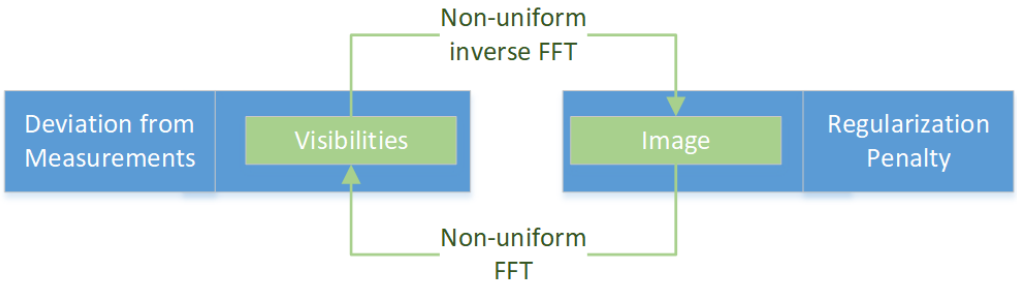


Figure 36: State-of-the-art Compressed Sensing Reconstruction Architecture

Figure 36 depicts the architecture used by Compressed Sensing reconstructions. The Visibilities get transformed into an image with the non-uniform FFT approximation. The algorithm then modifies the image so it reduces the regularization penalty. The modified image gets transformed back to Visibilities and the algorithm then minimizes the difference between measured and reconstructed Visibilities. This is repeated until the algorithm converges to an optimum.

In this architecture, state-of-the-art Compressed Sensing algorithms need approximately 10 or more non-uniform FFT cycles to converge. It is one source for the higher runtime costs. For MeerKAT reconstructions

the non-uniform FFT tends to dominate the runtime costs. A CLEAN reconstruction with the Major Cycle Architecture already spends a large part of its time in the non-uniform FFT. Compressed Sensing algorithms need even more non-uniform FFT cycle on top of the ”Image Regularization” step being generally more expensive than CLEAN deconvolution. There is one upside in this architecture: State-of-the-art algorithms managed to distribute the ”Image Regularization” operation.

11.3 Hypothesis for reducing costs of Compressed Sensing Algorithms

Compressed Sensing Algorithms are not bound to the Architecture presented in section 11.2. For example, we can design a Compressed Sensing based deconvolution algorithm and use the Major Cycle Architecture instead.

Our hypothesis is: We can create a Compressed Sensing based deconvolution algorithm which is both distributable and creates higher quality reconstructions than CLEAN. Because it also uses the Major Cycle architecture, we reckon that the Compressed Sensing deconvolution requires a comparable number of non-uniform FFT cycles to CLEAN. This would result in a Compressed Sensing based reconstruction algorithm with similar runtime costs to CLEAN, but higher reconstruction quality and higher potential for distributed computing.

11.4 State of the art: WSCLEAN Software Package

11.4.1 W-Stacking Major Cycle

11.4.2 Deconvolution Algorithms

CLEAN MORESANE

11.5 Distributing the Image Reconstruction

11.5.1 Distributing the Non-uniform FFT

11.5.2 Distributing the Deconvolution

12 Handling the Data Volume

The new data volume is a challenge to process for both algorithms and computing infrastructure. Push for parallel and distributed algorithms. For Radio Interferometer imaging, we require specialized algorithms. The two distinct operations, non-uniform FFT and Deconvolution, were difficult algorithms for parallel or distributed computing.

The non-uniform FFT was historically what dominated the runtime []. Performing an efficient non-uniform FFT for Radio Interferometers is an active field of research[24, 32], continually reducing the runtime costs of the operation. Recently, Veeneboer et al[12] developed a non-uniform FFT which can be fully executed on the GPU. It speeds up the most expensive operation.

In Radio Astronomy, CLEAN is the go-to deconvolution algorithm. It is light-weight and compared to the non-uniform FFT, a cheap algorithm. It is also highly iterative, which makes it difficult for effective parallel or distributed implementations. However, compressed sensing based deconvolution algorithms can be developed with distribution in mind.

12.1 Fully distributed imaging algorithm

Current imaging algorithms push towards parallel computing with GPU acceleration. But with Veeneboer et al’s non-uniform FFT and a compressed sensing based deconvolution, we can go a step further and create a distributed imaging algorithm.

13 Image Reconstruction for Radio Interferometers

In Astronomy, instruments with higher angular resolution allows us to measure ever smaller structures in the sky. For Radio frequencies, the angular resolution is bound to the antenna dish diameter, which puts practical and financial limitations on the highest possible angular resolution. Radio Interferometers get around this limitation by using several smaller antennas instead. Together, they act as a single large antenna with higher angular resolution at lower financial costs compared to single dish instruments.

Each antenna pair of an Interferometer measures a single Fourier component of the observed image. We can retrieve the image by calculating the Fourier Transform of the measurements. However, since the Interferometer only measures an incomplete set of Fourier components, the resulting image is ”dirty”, convolved with a Point Spread Function (PSF). Calculating the Fourier Transform is not enough. To reconstruct the from an Interferometer image, an algorithm has to find the observed image with only the dirty image and the PSF as input. It has to perform a deconvolution. The difficulty lies in the fact that there are potentially many valid deconvolutions for a single measurement, and the algorithm has to decide for the most likely one. How similar the truly observed image and the reconstructed images are depends largely on the deconvolution algorithm.

State-of-the-art image reconstructions use the Major Cycle architecture (shown in Figure 37), which contains three operations: Gridding, FFT and Deconvolution.

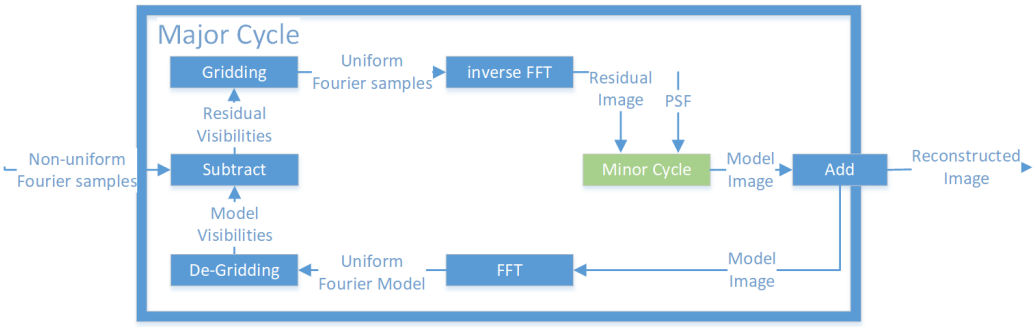


Figure 37: The Major Cycle Architecture of image reconstruction algorithms

The first operation in the Major Cycle, Gridding, takes the non-uniformly sampled Fourier measurements from the Interferometer and interpolates them on a uniformly spaced grid. The uniform grid lets us use FFT to calculate the inverse Fourier Transform and we arrive at the dirty image. A deconvolution algorithm takes the dirty image plus the PSF as input, producing the deconvolved ”model image”, and the residual image as output. At this point, the reverse operations get applied to the residual image. First the FFT and then De-gridding, arriving at the non-uniform Residuals. The next Major Cycle begins with the non-uniform Residuals as input. The cycles are necessary, because the Gridding and Deconvolution operations are only approximations. Over several cycles, we reduce the errors introduced by the approximate Gridding and Deconvolution. The final, reconstructed image is the addition of all the model images of each Major Cycle.

13.1 Distributed Image Reconstruction

New Interferometer produce an ever increasing number of measurements, creating ever larger reconstruction problems. A single image can contain several terabytes of Fourier measurements. Handling reconstruction problems of this size forces us to use distributed computing. However, state-of-the-art Gridding and Deconvolution algorithms only allow for limited distribution. How to scale the Gridding and Deconvolution algorithms to large problem sizes is still an open question.

Recent developments make a distributed Gridder and a distributed Deconvolution algorithm possible. Veeneboer et al[12] found an input partitioning scheme, which allowed them to perform the Gridding on the GPU. The

same partitioning scheme can potentially be used to distribute the Gridding onto multiple machines. For Deconvolution, there exist parallel implementations for certain algorithms like MORESANE[4]. These can be used as a basis for a fully distributed image reconstruction.

In this project, we want to make the first steps towards an image reconstruction algorithm, which is distributed from end-to-end, from Gridding up to and including deconvolution. We create our own distributed Gridding and Deconvolution algorithms, and analyse the bottlenecks that arise.

13.2 First steps towards a distributed Algorithm

In this project, we make the first steps towards a distributed Major Cycle architecture (shown in figure 37) implemented C#. We port Veeneboer et al's Gridder, which is written in C++, to C# and modify it for distributed computing. We implement a simple deconvolution algorithm based on the previous project and create a first, non-optimal distributed version of it.

In the next step, we create a more sophisticated deconvolution algorithm based on the shortcomings of the first implementation. We use simulated and real-world observations of the MeerKAT Radio Interferometer and measure its speed up. We identify the bottlenecks of the current implementation and explore further steps.

From the first lessons, we continually modify the distributed algorithm and focus on decreasing the need for communication between the nodes, and increase the overall speed up compared to single-machine implementations. Possible Further steps:

- Distributed FFT
- Replacing the Major Cycle Architecture
- GPU-accelerated Deconvolution algorithm.

A state-of-the-art reconstruction algorithm has to correct large number of measurement effects arising from the Radio Interferometer. Accounting for all effects is out of the scope for this project. We make simplifying assumptions, resulting in a proof-of-concept algorithm.

14 Ehrlichkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende schriftliche Arbeit selbstständig und nur unter Zuhilfenahme der in den Verzeichnissen oder in den Anmerkungen genannten Quellen angefertigt habe. Ich versichere zudem, diese Arbeit nicht bereits anderweitig als Leistungsnachweis verwendet zu haben. Eine Überprüfung der Arbeit auf Plagiate unter Einsatz entsprechender Software darf vorgenommen werden.
Windisch, December 9, 2019

Jonas Schwammberger