P9 Distributed Image Reconstruction for the new Radio Interferometers

Jonas Schwammberger

December 5, 2019

1 Introduction

asdfadsfjdaklkjdfklafjklasdfjdöklafkjaklvmadfklvnak
adlkfnadölöfdsklafjdklfasdlfkdjsflakjsdfkldfjadklsfjadsklfjsdfkladjsflkdfjklsdafjsdklfj

2 Methods

3 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 **??** 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:

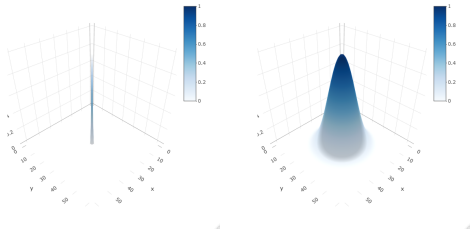
$$\underset{x}{minimize} \frac{1}{2} \|I_{dirty} - x * PSF\|_2^2 + \lambda ElasticNet(x) \tag{3.1}$$

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 3.2. We then go over the efficient CPU, GPU and distributed implementations of the serial coordinate descent algorithm.

3.1 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 1 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.



(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 1: Effect of the L1 and L2 Norm separately.

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[1], 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, be-

coming ”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:

$$ElasticNet(x, \alpha) = \alpha \|x\|_1 + \frac{1-\alpha}{2} \|x\|_2 \quad (3.2)$$

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 (3.2) for each pixel and sum up the results, or if we evaluate (3.2) 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 (3.2) 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:

$$ElasticNetProximal(x, \lambda, \alpha) = \frac{max(x - \lambda\alpha, 0)}{1 + \lambda(1-\alpha)} \quad (3.3)$$

In our deconvolution problem, we can apply the proximal operator (3.3) 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.

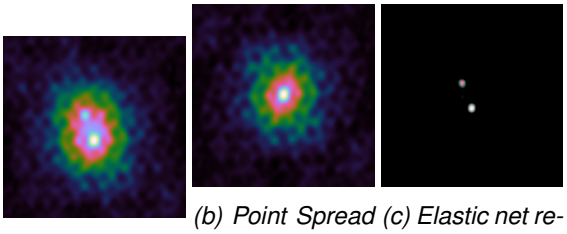
The elastic net regularization is separable. We can calculate its penalty for each pixel independently of its neighbors. As such, its proximal operator is also independent of the neighbors. It is the only regularization we use in this project. We now derive a coordinate descent based deconvolution algorithm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

A side note on the proximal operator used in this project (3.3): 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 [2].

3.2 Serial coordinate descent algorithm

Our serial coordinate descent deconvolution algorithm minimizes the deconvolution objective (3.1). 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 2a shows the dirty image of two point sources, and Figure 2b the PSF. The deconvolved image with elastic net regularization is shown in Figure 2c. I.e. Figure 2c is the optimum x of the objective function (3.1).



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

Figure 2: 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[1]. Also,

[1] There are block coordinate descent methods that optimize

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[3, 4]. We discuss the efficient implementation in Section 3.3. First, we explain each step in the serial coordinate descent algorithm in more detail.

3.2.1 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.

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.

a block of coordinates at each iteration. They are also discussed together with parallel coordinate descent methods in Section **??**

3.2.2 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 (3.1) is reduced to a one dimensional problem:

$$\underset{x}{minimize}\frac{1}{2}\left\|I_{dirty}-x_{location}*PSF\right\|_2^2+\lambda ElasticNet(x_{locati} \tag{3.4}$$

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 (3.4) 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 (3.4) ($\left\|I_{dirty}-x_{location}*PSF\right\|_2^2$). After simplifying the partial derivative, we arrive at the calculation:

$$\begin{aligned} residuals &= I_{dirty}-x*PSF \\ gradient_{location} &= \langle residuals, PSF_{location}\rangle \\ Lipschitz_{location} &= \langle PSF_{location}, PSF_{location}\rangle \\ pixel_{opt} &= \frac{gradient_{location}}{Lipschitz_{location}} \end{aligned} \tag{3.5}$$

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 (3.4) 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 (3.5). 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 (3.5) 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 2, 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 (3.3) with the gradient calculation, and we arrive at the following update step:

$$pixel_{opt} = \frac{max(gradient_{location} - \lambda\alpha, 0)}{Lipschitz_{location} + (1-\alpha)\lambda} \quad (3.6)$$

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.

3.2.3 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 (3.6).

```
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.

3.3 Efficient implementation

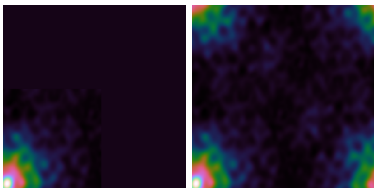
The bottleneck of the serial coordinate descent algorithm are all the inner products that need to be calculated in each iteration. In each iteration, we need to know the gradient for every pixel, and the

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.

3.3.1 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 3: 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.

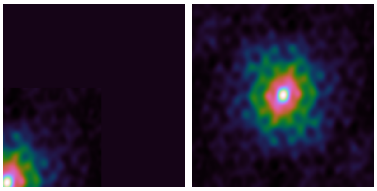
Zero padding assumes that after the edge, the image is zero. Non-zero pixels at the right edges of the image do not influence the left edge after convolution. This is the physically plausible solution. However, the zero padded convolution is more expensive to calculate. We either have to calculate the 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 [5]. We decide on using the zero-padded convolution. This choice influences how we calculate the Lipschitz and gradients efficiently.

3.3.2 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 4a shows the PSF shifted to a pixel location. The Lipschitz constant is by squaring all values of Figure 4a 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 4: 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 4b. 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.

3.3.3 Using a map of gradients

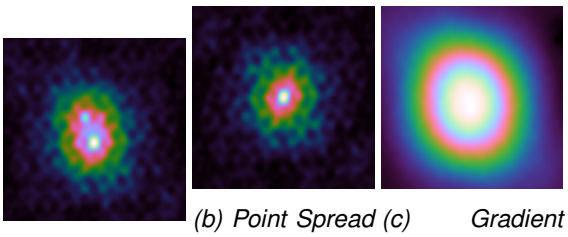
For an efficient greedy strategy, we need to know the gradient for each pixel. We show how to calculate the initial map of gradients in linearithmic time($O(nlogn)$ and how to update it directly after a change in the reconstructed image x. As we will see, we can use a map of gradients and drop the residual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

$$\begin{aligned} psfFlipped &= FlipUD(FlipLR(PSF)) \\ gradients &= iFFT(FFT(residuals) * FFT(psfFlipped)) \end{aligned} \tag{3.7}$$

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 5.



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

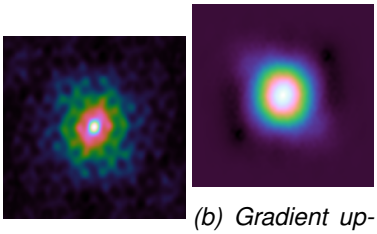
Figure 5: Example of the gradient calculation.

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 (3.7) 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 6b.

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



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

Figure 6: Example problem with two point sources.

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 6b.

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 (3.7) in each iteration (re-calculating the PSF correlation with the 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 **??**.

3.4 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 precalculate the Lipschitz constants and the gradient map. Then during iterations, we update the gradient map directly.

This leads to the following algorithm:

```
1  dirty = IFFT(GridVisibilities(
       visibilities))
2  residualsPadded = ZeroPadding(dirty)
3
4  psfPadded = ZeroPadding(PSF)
5  psfPadded = FlipUD(FlipLR(psfPadded))
6  gradientUpdate = iFFT(FFT(ZeroPadding(
       PSF)) * FFT(psfPadded))
7
8  x = new Array[,]
9  gradientsMap = iFFT(FFT(
       residualsPadded) * FFT(psfPadded))
10 lipschitzMap = CalcLipschitz(PSF)
11
12 objectiveValue = 0.5* Sum(residuals *
       residuals) + ElasticNet(x)
13 maxAbsDiff = 0
14 do
15   oldObjectiveValue = objectiveValue
16
17   //Step 1: Search pixel
18   maxAbsDiff = 0
19   maxDiff = 0
20   pixelLocation = (-1, -1)
21   for(i in Range(0, dirty.Length(0))
22     for(j in Range(0, dirty.Length(1))
23       oldValue = x[i, j]
24       tmp = gradientsMap[i, j] +
             oldValue * lipschitzMap[i, j
             ]
25       optimalValue = Max(tmp - lambda*
             alpha) / (lipschitz[i, j] +
             (1 - alpha)*lambda)
26       diff = optimalValue - oldValue
27
28       if(maxAbsDiff < Abs(diff))
29         maxAbsDiff = Abs(diff)
30         maxDiff = diff
31         pixelLocation = (i, j)
32
33   //Step 2: Optimize pixel. Now all
         that is left is to add the
         maximum value at the correct
         location
34   x[pixelLocation] += maxDiff
35
36   //housekeeping
37   shiftedUpdate = Shift(gradientUpdate
         , pixelLocation)
38   gradientMap = gradientMap -
         shiftedUpdate * maxDiff
39 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.

3.5 GPU implementation

We implemented the serial coordinate descent algorithm on the GPU. It is implemented in .Net Core with ILGPU[6]. 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:

```
1 MaxPixelKernel(x, gradienstMap,
      lipschitz, location, maxPixel)
2   oldValue = x[location]
3   tmp = gradientsMap[location] +
        oldValue * lipschitzMap[location
        ]
4   optimalValue = Max(tmp - lambda*
        alpha) / (lipschitz[location] +
        (1 - alpha)*lambda)
5   diff = optimalValue - oldValue
6   currentPixel = (absDiff = Abs(diff),
        diff = diff, location =
        location)
7   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[7] 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.

```
1 do
2   //Step 1: Search pixel
3   maxPixel = (absDiff = 0, diff = 0,
        location = (-1, -1))
4   ExecuteMaxPixelKernel(x,
        gradienstMap, lipschitz,
        maxPixel)
5   SynchronizeKernels()
6
7   //Step 2: Optimize
8   ExecuteUpdateXKernel(x, maxPixel.
        diff, maxPixel.location)
9   ExecuteUpdateGradientsKernel(
        gradientsMap, gradientUpdate,
        maxPixel.diff, maxPixel.location
        )
10  SynchronizeKernels()
11
12 while maxPixel.absDiff < epsilon
```

ILGPU implementation finished.

3.6 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

3.7 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.

References

[1] 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.

[2] 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.

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

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

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

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

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

bibencoding=utf8