# Project Final Report – Distributed Tileable Image Conversion with Natural Cubic Splines

Andrew Hartz, NAU, CS 599, Flagstaff, USA, AH568@nau.edu

*Abstract- This project develops an MPI program to convert non-tileable images into tileable images. The program utilizes a modified natural cubic spline method to edit near edge pixel data and a closest replacement method to ensure that new pixel colors are selected from colors that previously existed in the image. The natural cubic spline is used to adjust the trend rather than the actual values and is forced to behave less erratically with a modification to the data used in the curve. A moving average is used perpendicularly to spline calculations to diminish banding in the results. Where the trend change results in undesirable colors for pixels, trend editing is skipped, and the pixel's original color is used. Image results are good, giving a tileable image with acceptable distortion, a slightly crumpled look near the edges. Image results still have some room for improvement. Performance results compute quickly but show mediocre to poor parallel scaling with parallel efficiency of 69% down to 06%.*

*Index Terms* – MPI, distributed computing, natural cubic splines, tileable images

## I. Introduction

This project aims to convert a non-tileable image into a tileable image. A *tileable image* is an image that can be repeated without a noticeable discontinuity at the edge.

The program utilizes MPI to spread the computation among multiple processing ranks. The program can be run on a compute cluster in parallel.

The program uses a modified natural cubic spline method to edit pixels near the right and bottom edges of the image. To edit the right edge, data is repeated for each row. A curve is fit using data on either side of the region being edited, excluding the pixels within the region. This curve is then used to edit the region of pixels. The same process is repeated for columns to edit the bottom edge.

PNG images are made up of pixels with four color channels. Three of these are being edited by this program, red, green, and blue. Channels have integer values between 0 and 255.

The splines fit to the data consider location as the independent variable and one of these color channels as the dependent variable. Each row has three splines, one for each color channel. Calculations from the three splines are used together to make up the new color value at a pixel within the near-edge region.

## II. Background

*A. Smoothing method*

Natural cubic splines can be used to independently fit a smooth curve to rows and columns of the data. If there is a big jump in the data, these splines can be used to smooth between given points without jagged changes in slope. Below is the conceptual representation of how one would fit a spline to a single color channel.

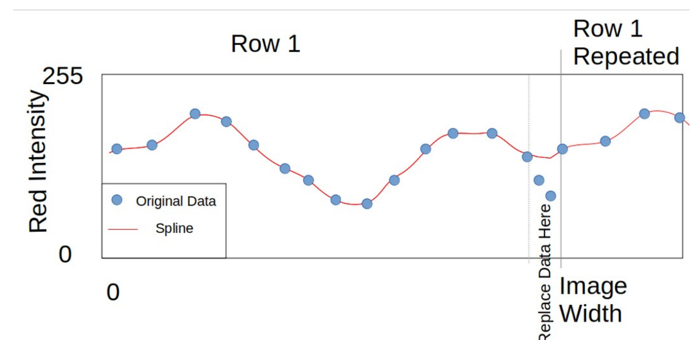


**Figure 1:** Example spline for a row, color red values as the dependent variable

If the points in the "Replace Data Here" region were included in the spline calculation, the red line would pass through them. By excluding those points from the spline calculation, one can calculate alternative values that will connect to the repeated data.

One should note that this is not a perfect representation of the problem. There would be a lot more blue points, and they would vary up and down much more erratically but would have an overall trend similar to what is shown.

The spline code produces four coefficients for each segment between two points. These coefficients can be used in an equation of the form shown below to estimate the color intensity for a single color channel.

$$Color_x = a+bx+cx^2+dx^3 \quad (1)$$

The spline code is based on the seven-step procedure for calculating splines described in the textbook Numerical Analysis by Burden and Faires.

*B. PNG read and Write*

For importing and exporting PNG files, this project uses stb. Stb is a public domain licensed, free set of c++ files for importing multiple types of images. The code is available in .h files containing functions. The files for importing and exporting PNG files into c++ were included.

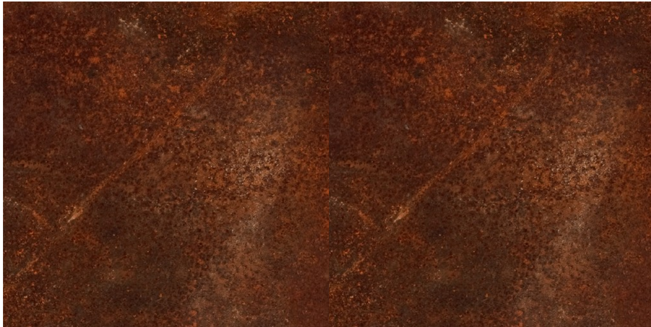

**Figure 2:** Original Non-tileable image repeated

## III. Description of the approach to the problem studied

*A. Milestones*

The milestones that make up the initial project plan are shown below.

- Read and export png data.
- Implement a smoothing method such as Natural cubic splines.
- Create a sequential program for calculating Natural cubic spline results and replacing near edge pixels
- Implement closest values selection from original pixel colors sequentially.
- Convert spline related code to distributed MPI computing
- Convert closest values code to distributed MPI computing
- Run tests tiling images

*(a) Read and export png data:* An image of rust was chosen for testing for two reasons. One, it is not a tillable image. It is a photographed image cut square. Two, the image is primarily shades of red. This coloration made it easier to decipher the input from stb and figure out the function inputs for the stbi_load function. It was easy to see when the color channels were organized correctly.

*(b) Implement a smoothing method: A modification on natural cubic splines was used for smoothing. The original intention was to use natural cubic splines based on an algorithm described in the text Numerical Analysis.*

*(c) Closest value selection:* In this portion of the project, a 3D array 256 x 256 x 256 is created and initialized to all 0s. Cycling through each point in the original data, the bin[r][g][b] is set to 1 for each point. With discrete data limited to a small range, that seemed to be a good way to handle the problem rather than any more complex bin solution. The bins are created on each rank.

The bin array was later converted to a 1D array and a function was written to convert red green and blue indices to a single index.

*(d) Sequential Program:* The sequential program calculated spline values for near edge pixels. It served no greater purpose than as a step toward a parallel program.

*(e) Parallel Program:* The MPI code utilizes MPI_Bcast, MPI_Allgather, MPI_Allreduce, and MPI_Reduce. MPI_Bcast is used to pass the imported image data to each rank. MPI_Allgather shares distributed results of row calculations. For sharing results of column calculations, MPI_Gather is used instead since only one rank is needed to write to file. MPI_AllReduce is used to combine the bin information on all ranks. A maximum operation is used since 1 is the flag for the color being present in the original image, and 0 for the color not present in the original image. MPI_Reduce is used to collect performance time data. Below is an overview of the basic procedure.

- Image data read into rank 0
- Image data broadcast to all ranks
- Image data is separated into 2D arrays of red, green, and blue color channels
- Bins are filled in parallel
- Bins are synchronized across all ranks
- Parallel calculations are performed for rows
- Results are synchronized across all ranks
- Parallel calculations are performed for columns
- Results are synchronized on rank 0
- Rank 0 data are converted to the output format and are written to file.

*(f) Convert Spline code to MPI:* The spline calculations and bin replacement were split between ranks. The results were then converted to the closest values.

*(g) Convert Closest Values to MPI:* Instead of writing directly into the 2D output array, it was necessary to crate intermediary 1D arrays for local results and global results. These 1D arrays only contain the data that is being changed. It is easy to use

MPI_Allgather or MPI_Gather to combine the local data into the global data. The 2D output array is updated with the values from the 1D global results.
It is necessary to use fixed intervals size for the number of rows or columns assigned to each rank. Rather than assigning extra rows to the last rank, the extra rows are split evenly, and dummy rows are assigned to the last rank to even out the array sizes. These dummy rows are passed around but not calculated.

*B. Adaptations*

A list below shows subsequent adaptations to the project that were not originally planned for but were required to attain a better result.

- Tamping down spline fluctuation by copying edge values
- Cubic spline trend adjustment rather than replacement
- Outlier skip
- Perpendicular smoothing

*(a)Tamping down spline fluctuation by copying edge values: T*here was an apparent bug in the spline replacement of pixel colors. One can see where the near edge column spline data overrides the row data since they look terrible in different patterns.

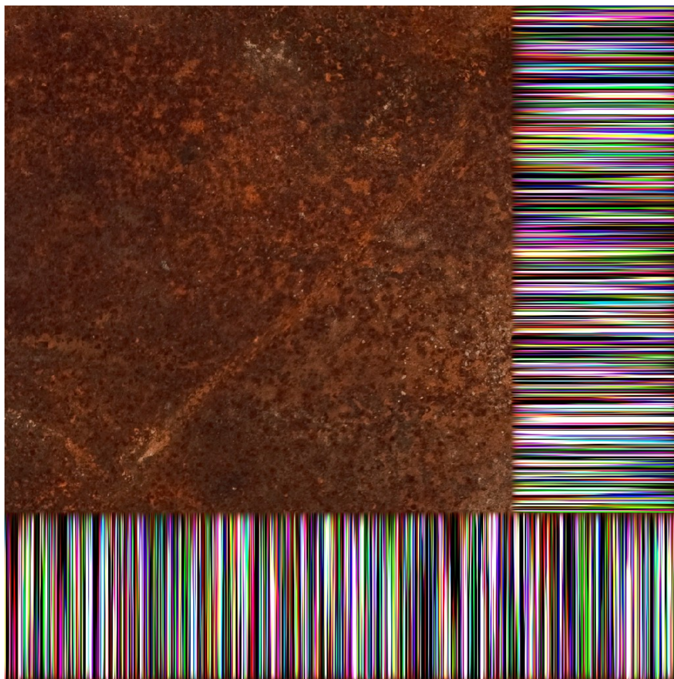

**Figure 3:** Buggy Image result of Spline calculation

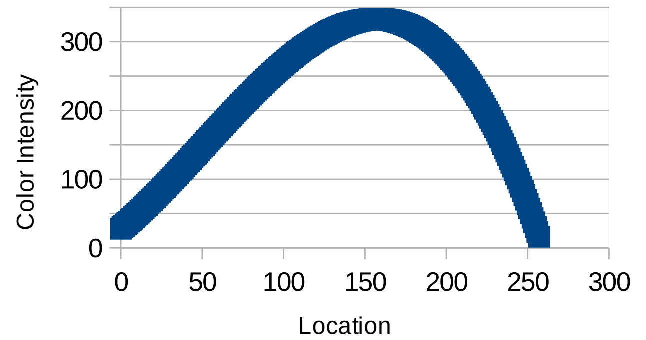


**Figure 4:** Erratic Spline

Figure 4 shows color intensities for a single row. Notice that the curve peaks above 255, the maximum color intensity allowed for a pixel color channel. What is seen above is the effect of erratically changing data on a spline. This erratic change in consecutive color intensities causes curves with massive arches.

Cubic splines preserve continuous third-order relationships at each point. Two curves end at each point, one to the left and one to the right. For the point's x value, both curves evaluate the same y, dy/dx (slope), and d2y/dx2 (instantaneous rate of change).

A straightforward use of a cubic spline with our data does not interpolate values between the data points well because a row of color data does not change smoothly from point to point. Also, a cubic spline fit to the data is not bounded by the desired output range. This is a problem that could not be overcome without simplifying the spine.

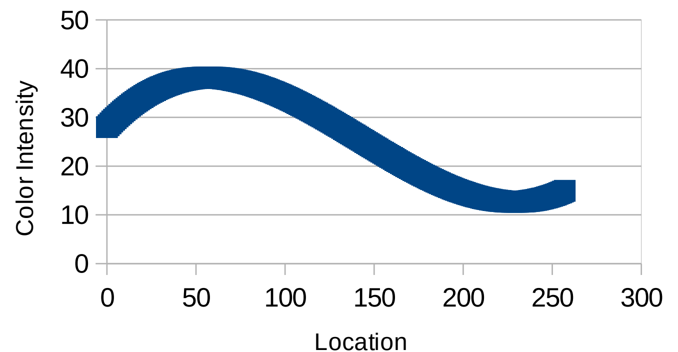


**Figure 5:** Tamped down spline

To solve this bug the color values of the points bounding the region of points being replaced are copied outward away from the region. The result shown in Figure 5 is a curve that is

much more likely to remain within the range of acceptable color values.

Unfortunately, this curve is not a true spline and is not influenced by the rest of the data in the row. This pseudo-spline was meant to be a temporary solution until a way was found to reincorporate the remaining row data in a manner that was not erratic.

An argument is included when the program is run. This argument specifies how far to copy color data outward; it is named NUMSETEQ. For example, if the value is set to three, then the pixel just before the region has its colors copied to the three pixels before it. Also, the pixel just after the region has its colors copied to the three pixels after it. Since we are using cubic splines, value of NUMSETEQ >= 3 causes only one point's color data on either side of the region to actually affect the curve. It gives a tame result at the cost of of being less true to the data.

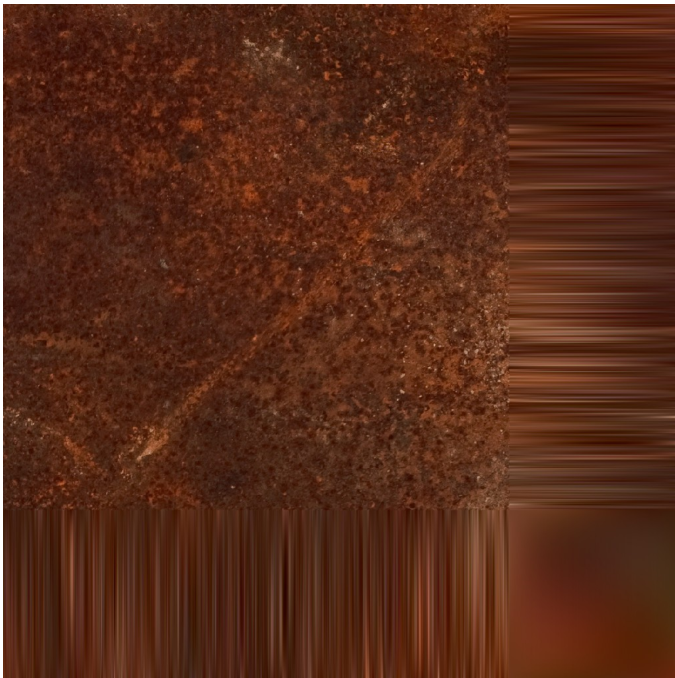

**Figure 6:** Tamped down spline resulting image too smooth

The result of using a spline to replace color information near the edges of the image was overly smoothed. Everywhere else in the image color changes erratically, and the smooth data curves look out of place.

There is a similarity in the color tone, but the spline results are missing the essential variability of the data.

Also, it should be noted that the bin selection function passes through any pixel colors below 0 or above 255 without changing them. This counterintuitive way of handling erroneous data was beneficial when developing adjustments, since errors were more visible. The bin selection function has not been changed.

*(b) Cubic spline trend adjustment rather than replacement:* Trend adjustment is used to solve the over smoothing problem. Changing the use of the curve from calculating color data to trend adjustment involves multiple steps:

- copying original data into the local vector
- creating the curve for the new trend
- creating the curve from the old trend
- adjusting the original data with the difference in the trends

Two spline curves are created the "new," which captures how you want the data to trend so that it will tile, and the "old," which is the trend of the data already has without tiling.

The "new" curve uses the same curve that was used to replace data before. It includes data up to the edge of the region we are replacing, leaving out the data from the region being replaced. Then it repeats the row data after the gap adjusting location information. The colors repeat, but the locations continue increasing with a large gap in their values for the missing data. The curve fills the gap.

The "old curve works the same way except that the gap stops one point earlier. Before the data repeats, the last point from the line (row or column) is included. A curve is fit to the data before the data is repeated.

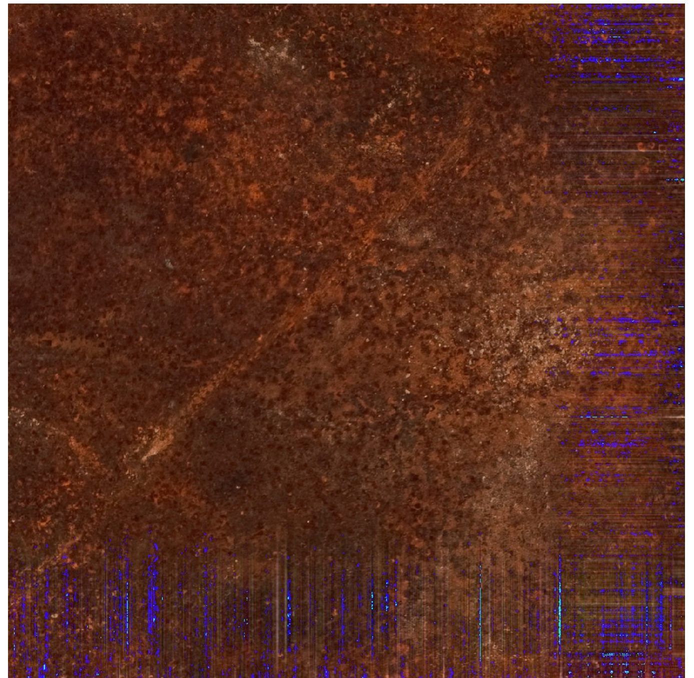

**Figure 7:** Spline adjustment results outliers

The program then takes the original data, subtracts the old trend, and adds the new trend. The result is that the data tiles, and also includes the erratic fluctuations from the original data. New problems arise, but this is a step in the right direction.

The new problems are outliers and what looks like crumpled edges or bands of lighter or darker color. The problem of outliers is caused by local maxima or minima excessively exaggerated by trend adjustment.

*(c) Outlier skip:* Outlier skip is a straightforward solution. If adjusting the trend of a point gives a bad result, then do not adjust the trend of that point. Skipping a trend adjustment of problem points here and there does not seem to create any noticeable artifacts. It is an effective solution to the outlier problem.

Determining the distance from the edge where spline results should replace pixels is left to the user.

The images in this paper show testing with a much larger distance than would be desirable, 256 pixels. The intent is for the changes to the image to be more noticeable while testing. With the final method, even this range did not look too bad. There is an effect that looks like the image is banded or crumpled near the edge.

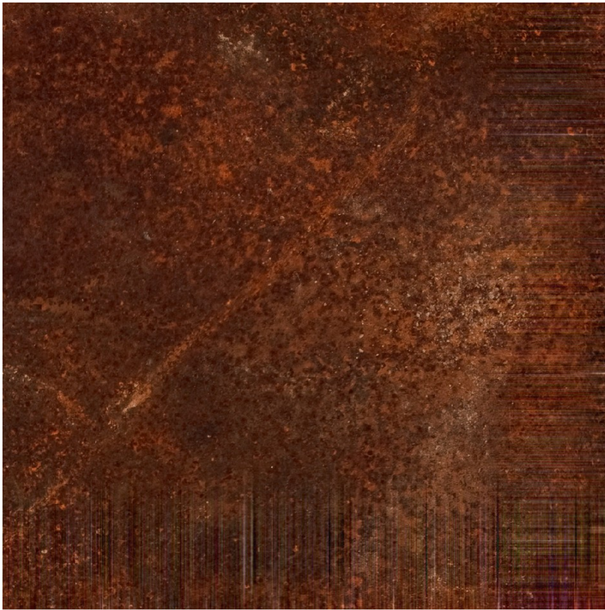

**Figure 8: Ou**tlier skip resulting image banding

*(d) Perpendicular smoothing:* A moving average is calculated perpendicular to the spline calculations to mitigate the edge crumple problem.

When considering the 1D array of local color data, this smoothing is not performed on the original data initially filling the array. It affects the new and old spline data. It is, in essence, a moving average of trend lines, so adjacent trends are similar. This trick eliminates banding and reduces crumple. However, the crumpled effect seems not to be entirely avoidable. Reducing the length of the edge region makes it less noticeable.

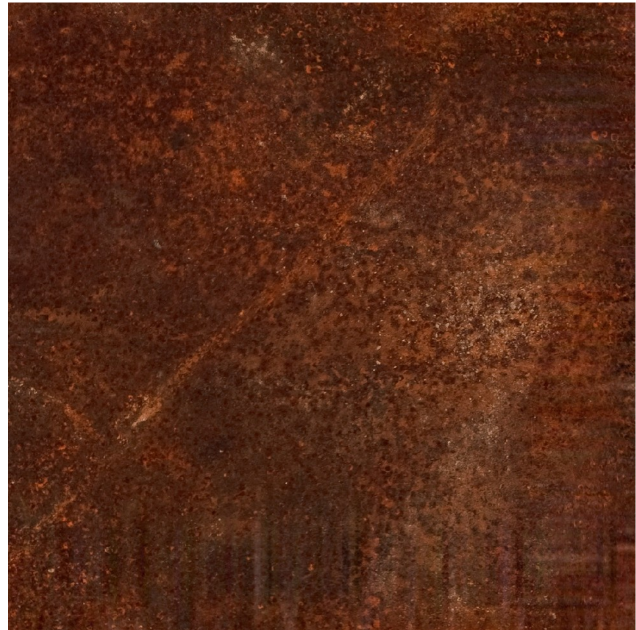

**Figure 9:** Final result after perpendicular smoothing

*C. Possible Improvements*

The following changes have not been implemented but may be beneficial.

- Smoothing along the spline direction
- Increased Interval spacing for spline calculation

*(a) Smoothing along the spline direction:* This method is intended to be an extension of the moving average of perpendicular smoothing to include a moving average in the direction of the spline as well.

*(b) Increased interval spacing for spline calculation:* It is hypothesized that a spline with more widely spaced nodes may behave in a less erratic manner more useful to this calculation.

Not every pixel in the direction of the spline would be included in the spline calculation. Pixels would be skipped at a fixed interval defined by the user.

This interval spacing may be more effective in combination with smoothing along the spline direction. Since smoothing along the spline direction does not use the interval spacing in

the moving average calculation, data being skipped over still affects the moving average calculations.

## IV. Results

The current results are shown below. The 1024 by 1024 (1K) rust image was tested to see how well it performed in parallel. Between 1 and 20 ranks were used. The times measured were the total time of execution, the time excluding reading and writing of the image to disk, the time performing communication, and the computation time (Exclude IO – Communication).

Table 1
Time 1K Texture

| # of Ranks | Total | Exclude IO | Computation | Communication |
|---|---|---|---|---|
| 1 | 0.725 | 0.366 | 0.345 | 0.021 |
| 2 | 0.601 | 0.244 | 0.193 | 0.055 |
| 4 | 0.539 | 0.188 | 0.110 | 0.079 |
| 8 | 0.522 | 0.172 | 0.070 | 0.102 |
| 12 | 0.561 | 0.211 | 0.060 | 0.177 |
| 16 | 0.518 | 0.166 | 0.050 | 0.115 |
| 20 | 0.61 | 0.24 | 0.05 | 0.206 |

From these time measurements, speedup and parallel efficiency could be calculated. By comparing the total column to the Exclude IO column in the Time, Speedup, and Parallel Efficiency tables, one can see that IO dominates the calculation time. IO especially dominates with a higher number of ranks. Parallel efficiency is poor, between 0.32 and 0.08. IO uses serial computation and brings down speedup and parallel efficiency.

Table 2
Speedup

| # of Ranks | Total | Exclude IO | Computation | Communication |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1.21 | 1.50 | 1.79 | 0.38 |
| 4 | 1.35 | 1.94 | 3.14 | 0.27 |
| 8 | 1.39 | 2.13 | 4.95 | 0.21 |
| 12 | 1.29 | 1.73 | 5.79 | 0.12 |
| 16 | 1.40 | 2.21 | 6.84 | 0.18 |
| 20 | 1.19 | 1.53 | 6.74 | 0.10 |

Table 3
Parallel Efficiency

| # of Ranks | Total | Exclude IO | Computation | Communication |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0.60 | 0.75 | 0.90 | 0.192 |
| 4 | 0.34 | 0.49 | 0.79 | 0.067 |
| 8 | 0.17 | 0.27 | 0.62 | 0.026 |
| 12 | 0.11 | 0.14 | 0.48 | 0.010 |
| 16 | 0.09 | 0.14 | 0.43 | 0.011 |
| 20 | 0.06 | 0.08 | 0.34 | 0.005 |

Excluding IO improves parallel efficiency, especially at higher ranks. Even so, parallel efficiency, as seen in Table 3, is only good for a low number of ranks. Without IO considered parallel efficiency is between 0.75 to 0.08.

Table 4
Time 8K Texture

| # of Ranks | Total | Exclude IO | Computation | Communication |
|---|---|---|---|---|
| 1 | 54.456 | 33.788 | 33.753 | 0.036 |
| 2 | 39.623 | 18.855 | 18.701 | 1.132 |
| 4 | 30.781 | 10.014 | 9.506 | 0.634 |
| 8 | 26.682 | 5.854 | 5.521 | 0.388 |
| 12 | 25.788 | 4.867 | 4.495 | 0.532 |
| 16 | 25.195 | 4.383 | 3.996 | 0.420 |
| 20 | 25.25 | 4.32 | 3.78 | 1.700 |

Table 4 through 6 repeat tables 1 through 3 for an 8K texture. As can be seen by comparing Table 6 to Table 3 parallel efficiency is slightly higher for larger textures but is still mediocre between 69% and 11%.

Table 5
8K Speedup

| # of Ranks | Total | Exclude IO | Computation | Communication |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1.37 | 1.79 | 1.80 | 0.03 |
| 4 | 1.77 | 3.37 | 3.55 | 0.06 |
| 8 | 2.04 | 5.77 | 6.11 | 0.09 |
| 12 | 2.11 | 6.94 | 7.51 | 0.07 |
| 16 | 2.16 | 7.71 | 8.45 | 0.08 |
| 20 | 2.16 | 7.82 | 8.92 | 0.02 |

Table 6
8K Parallel Efficiency

| # of Ranks | Total | Exclude IO | | Computation | Communication |
|---|---|---|---|---|---|
| 1 | 1 | 1 | | 1 | 1 |
| 2 | 0.69 | 0.90 | | 0.90 | 0.016 |
| 4 | 0.44 | 0.84 | | 0.89 | 0.014 |
| 8 | 0.26 | 0.72 | | 0.76 | 0.011 |
| 12 | 0.18 | 0.58 | | 0.63 | 0.006 |
| 16 | 0.14 | 0.48 | | 0.53 | 0.005 |
| 20 | 0.11 | 0.39 | | 0.45 | 0.001 |

The tests described above had NUMSETEQ set to three. This value signifies that the calculation does not utilize information from the full spline. Tests were performed with NUMSETEQ < 3 to reincorporate the rest of the spline, but image results were poor.

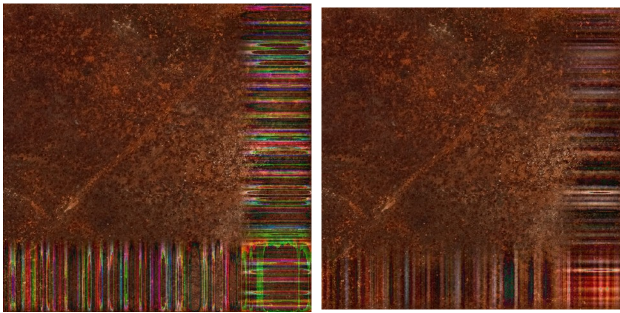

**Figure 10:** Removing Tamped down values Left NUMSETEQ = 0, Right NUMSETEQ = 2

**Figure 11:** Final image tiled

## V. Discussion and Conclusion

There is likely room to optimize this solution for parallel efficiency. Current parallel efficiency is *mediocre to poor*. The calculation is still fast with the small data sizes of textures compared to other types of data sets commonly used with MPI.

Good results for the output image were achieved. The program can create output images that tile well and look similar to the original image. The program has some problems with an effect that makes the edge of the image look somewhat crumpled. There is room for improvement.

The spline calculation had to be adjusted with moving averages and have color intensities tamping down near the region being edited so that the curve would not fluctuate erratically. The solution is acceptable but not ideal.

Ideally, the color intensities for the curve for the near edge region would be limited to the range of png color intensities from 0 to 255. The curve would incorporate the underlying data trend outside the region to influence the shape of the curve within the region. A natural cubic spine does not seem to be the appropriate tool for this job. Natural cubic splines may be effective with some transformation or pre and post-treatment of the data.

## VI. References


*R. Burden and J. Faires, Numerical Analysis, Ninth Edition, pg 149. Boston, MA, USA: Brooks/Cole, Cengage Learning, 2011.*

*S. Barrett, stb, GitHub.* [https://github.com/nothings/stb](https://github.com/nothings/stb) *(accessed April 27, 2022).*