

Scientific Data Compression Through Wavelet Transformation

Chris Fleizach

1. Introduction

Scientific data gathered from simulation or real measurement usually requires 64 bit floating point numbers to retain accuracy. Unfortunately, these numbers do not lend themselves to traditional compression techniques, such as run-length or entropy encoding used in most general compression schemes. A different approach is to recognize that a reversible signal transformation can encode the information using less information. Perhaps the most useful technique for this domain is the wavelet, which goes beyond traditional Fourier transforms by effectively capturing time and frequency information. The result is that most of the pertinent data is concentrated into a smaller range of values. At this point the data can be thresholded and still retain a high percentage of the “energy” of the signal. Subsequent reconstruction introduces only minor errors depending on the level of thresholding and the wavelet used. Because the data has been thresholded, there exists a large number of zeros which can subsequently be compressed using traditional techniques.

1.1. Project Goals

The project was comprised of a number of components designed to explore the domain of data compression with wavelets (biased towards turbulence data generated from the Navier-Stokes equations). An initial examination looked at various wavelets and thresholding schemes in order to determine the appropriate combination that would provide an acceptable error level, yet still result in good compression. A Matlab implementation was then done for 1-D and 2-D data that allowed for the use of three wavelet/thresholding techniques that corresponded to high compression/high error, medium compression/medium error and low compression/low error.

The next step was to create an implementation in C++ that performed that same functions that Matlab offered (namely 1D and 2D wavelet deconstruction and reconstruction), but also allowed for the compression of 3-D data natively. Another goal with the C++ program was to create a method for processing parts of the wavelet transformation in parallel on multiple nodes. The parallel implementation was done with XML-RPC, which added a large overhead for sending data, but allowed for a relatively straightforward implementation.

2. Wavelet Analysis

There exists a wide variety of ways in which the many types of wavelets can be used. Understanding enough to make sense of what was the best option took a fair amount of reading and experimentation. The goal was to choose which combination of decomposition, wavelet filter and thresholding technique would result in the best accuracy and and the best compression.

The first step was to look at the difference between the continuous and discrete wavelet transformations. While continuous transformations provide exact reconstruction, they are slow and encode redundant data. Instead, it was found that using only a dyadic sampling of data is remarkably more efficient and just as accurate. This is the essence of the discrete transform, which produces an approximation and a detail signal for each decomposition.

The next step was to choose the best wavelet function for compression. The methodology here had no theoretical underpinnings. Rather all wavelet filters were examined by brute force. There are a large number of such filters that tend to be useful in specific domains. For compression purposes, there were two biorthogonal filters that provided the best compression and accuracy. Biorthogonal filters differ from other wavelet filters by requiring different sets of coefficients to perform deconstruction and reconstruction. There are usually two sets for each, one that creates the approximation using a low pass filter and one that creates the details with a high pass filter. Figure 1 shows a graphical representation of the two filters chosen. Note that the filters themselves are only samples of the actual wavelet functions, in what is called a finite impulse response filter. Figure 2 shows the actual wavelet function for bior5.5 wavelet.

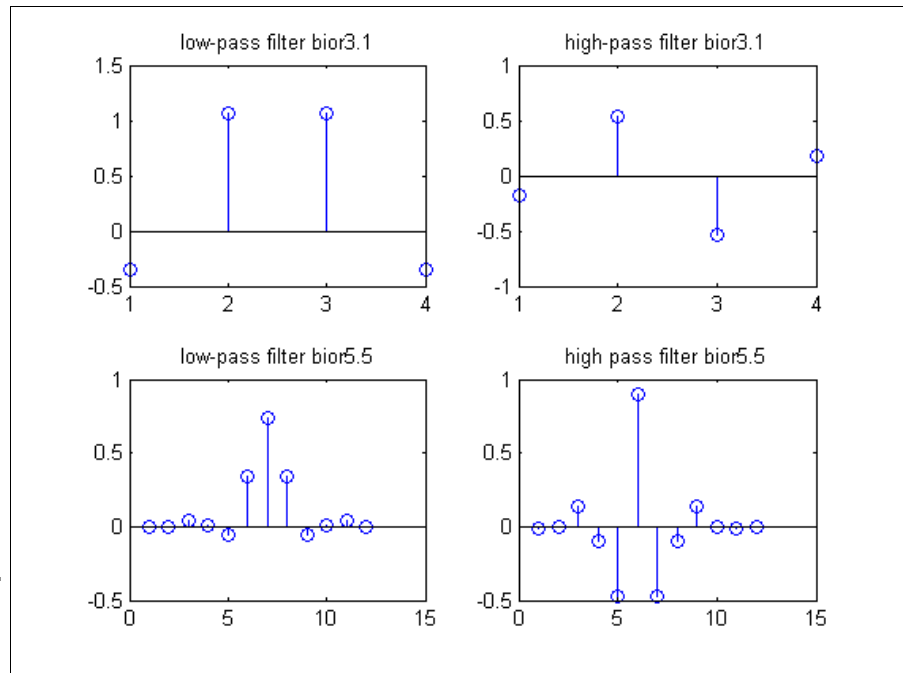


Figure 1: Biorthogonal Filters (3.1 and 5.5) were the most effective

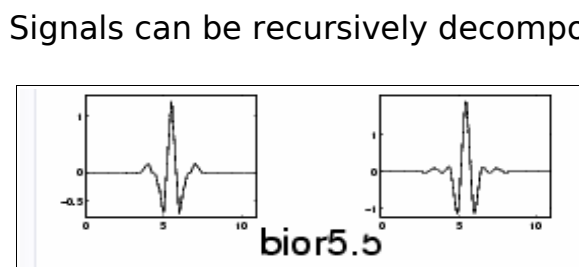


Figure 2: Bior5.5 Wavelet functions

Signals can be recursively decomposed with wavelets a number of times, which results in finer details and more general approximations. Doing so has the advantage of separating out more detail from the approximation vectors created. This in turn can lead to greater thresholding. Each iteration reduces the approximation vector by about half and introduces another detail vector. In the experiments, decomposing the signal, one, three times and five times was

examined. Typically, a signal can be decomposed $\log(\text{length})$ times. For this project, the level of decomposition did not vary based on size.

The actual testing of each compression methodology involved decomposing a 1000x1000 grid of vorticity data, thresholding it to a certain value, and then storing and compressing it using gzip. The data was then reconstructed and compared to the original to obtain the max error between the two. Thresholding was examined in a few ways. First, various detail coefficient vectors were removed. Next, reconstructing the signal with only the approximation vector was tried. Lastly, removing numbers below some small threshold was tested for limits between 1×10^{-4} and 1×10^{-7} . The latter method proved to offer the greatest accuracy.

3. Matlab Implementation

Three methods exhibiting different qualities were chosen for implementation. These are listed in Table 1. Low compression aimed offered a reasonable 2500% improvement in the data used, while maintaining a low error rate. Medium compression offered 3 times that of low, but the error was slightly more. Finally high compression gave a 128 time improvement over the original data. It's error was not significant. For all reconstructions, they were visually identical. It should be noted that these results are highly dependent on the data used. For other types of data, sizes or frame, the results will vary. In terms of velocity and vorticity data generated from the Navier Stokes equations, these methods worked very well. Note the relative error ignored values smaller than 10^{-6} in its calculations.

Type	Wavelet	Max Absolute Error	Max Relative Error	Compression Ratio
Low Comp.	Bior5.5 – Level 5 (1e-7 threshold)	7.35×10^{-8}	.026	25.36
Medium	Bior3.1 – Level 5 (1e-5 threshold)	3.54×10^{-6}	.941	59.91
High	Bior3.1 – Level 5 (1e-4 threshold)	3.03×10^{-5}	6.33	128.92

Data was 1000x1000 vorticity grid. Compression ratio is the ratio of the gzipped compressed transformation data compared to the gzipped compressed original data

To demonstrate the visual fidelity of the data, Figure 3 shows the representation of original data, the reconstruction and the error between the two for a 75x75 grid of velocity data using the “High” type of compression listed in Table 1. These results were obtained with the Matlab functions. The Matlab functions wavecompress, wavecompress2, wavedecompress and wavedecompress2 implemented compression and decompression for one and two dimensional data sets. These are described in more detail in the Appendix. (Special note: all programs assume the input data includes the size of the grid at the start of the file. The Navier Stokes program was modified for this purpose).

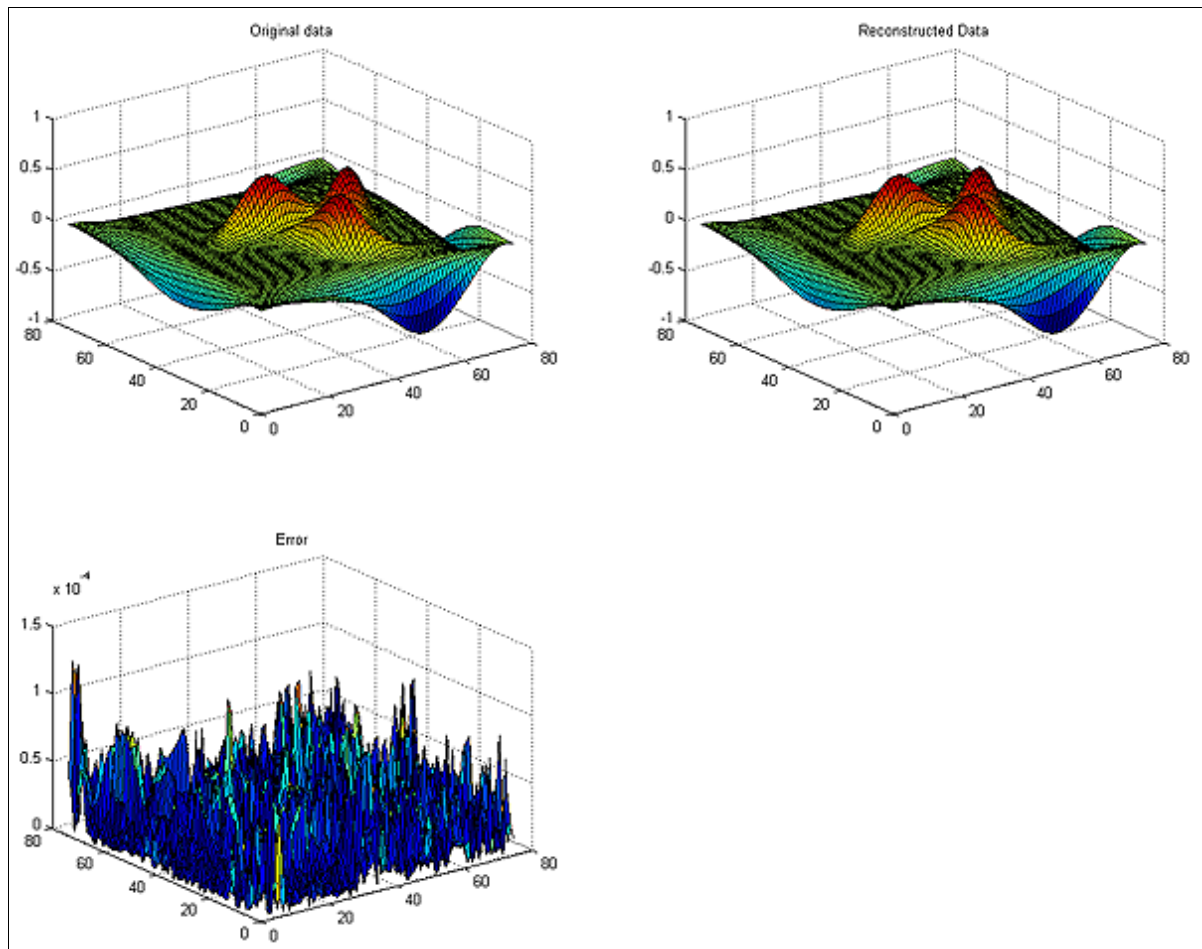


Figure 3: Matlab Processing of 75x75 grid with High Compression

4. C++ Implementation

A C++ implementation was desired for a number of reasons. Chief among them was to gain a thorough understanding of the discrete wavelet transform. Other goals were to support three dimensional data, which Matlab does not support natively, and to implement a basic parallel processing mechanism. The actual implementation of the transform turned out to be more difficult than expected. Although there was a wealth of information on the abstract formulas of wavelet decomposition, the subtleties involved make it fairly difficult to get right and process efficiently. The main challenge was handling how to extend the data depending on the wavelet filter and the type of data. There are different methods for handling the corners of the data whether it is symmetric, periodic, asymmetric and so on. The most appropriate method was zero-padding. The length of the wavelet filters also affects the amount of extension needed in the data. Fortunately, a library was found which had correct implementations of this extension and could be modified.

4.1. Decomposition algorithm

The basic algorithm in a wavelet transformation is to convolve a signal with the low pass wavelet filter (to obtain an approximation information) and to convolve it with the high pass wavelet filter (to obtain the detail information). During the convolution, only every other sample is recorded, which results in new signals that are about half the size of the original. The process can be repeated iteratively to obtain finer details and more general approximations. Thus, the second level of decomposition starts with the approximation of the level before and generates new approximation and detail information. Figure 4 illustrates the process.

The method is similar in the two dimensional case. First, all the rows are convolved with the low and high pass filters resulting in the approximation and detail vectors respectively. The rows are downsampled so that only the even indexed columns are kept. Next, the same filters are used on the downsampled data, except this time each column is decomposed. Only the even indexed rows are kept. At this point, there is one approximation and three details vectors. The process can again be recursively transformed, where the approximation information is taken as the original data. Figure 5 illustrates the 2-Dimensional decomposition.

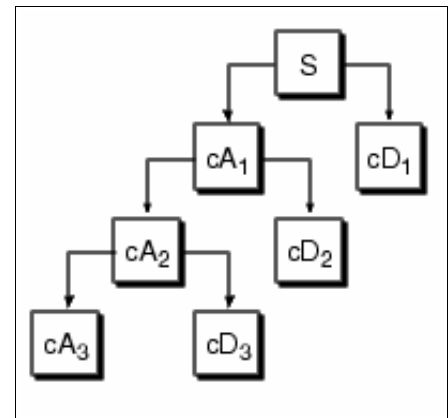


Figure 4: 1D Multilevel Decomposition

A wavelet transformation can be applied to any number of dimensions, but doing so becomes difficult because the number of detail signals generated at each level grows to $2^d - 1$ where d is the dimension. For the three dimensional case, each z axis is convolved and downsampled, then each y axis is convolved and downsampled and finally each x axis is processed. This is not a strict ordering though, as the convolution operates on a one dimensional signal only.

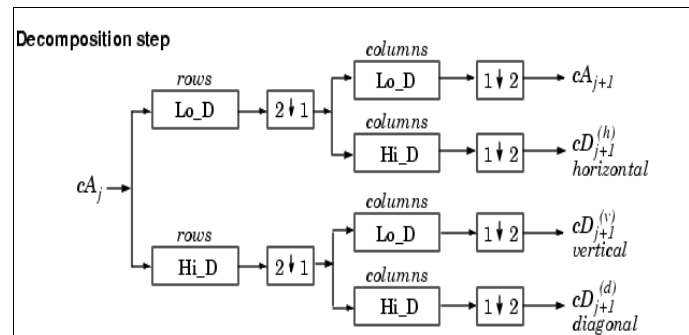


Figure 5: 2D Decomposition

4.2. Reconstruction

Reconstruction is a mostly straightforward process and operates in reverse of the decomposition. The detail and approximation signals from the last level of decomposition are taken and upsampled first. This means that zeros are inserted at every other index, which creates a signal of the original size. Then both are convolved with their respective reconstruction filters and finally they are combined with addition. In a multilevel process, the reconstruction will create the approximation vector to use in the next step. In higher dimensional cases, the approximation and detail vectors just need to be combined with the ones that they were created with in the decomposition phase.

unnecessary overhead. Ideally, the entire DataGrid should be sent to each node at the start of the level of decomposition. Currently, only one array is sent at a time, and then the results are sent back. This means two threads are created each time, and the operating system must bind to different ports each time. As the parallel processing was more proof of concept, the results were acceptable for this project, but deserve greater attention.

5. Results

The C++ implementation was tested using vorticity data of different sizes. The results from the High and Low compression modes are shown in Tables 3 and 4. For the high compression mode, it becomes apparent that real savings are only reached for a 512x512 grid, which is normally about 2MB. For larger sizes, there is a huge amount of data that can be thresholded, as evidenced by the 2500x2500 grid. Even more surprising is that accuracy is better for the larger grids, which is difficult to explain. With the high compression, since the data is thresholded to 1×10^{-4} , one would expect to see a few errors within that magnitude, but with large grids it seems to do better.

The relative error shows a different picture from the absolute error. Although most values have very little relative error, there are a few portions of the grid that exhibit a high relative error. Small values tend to make these errors blow up, so only values greater than 10^{-6} were used in the relative error calculation. Still the values may be greater than desired for some purposes.

As a comparison to the adaptive subsampling technique presented in the thesis of Tallat, the wavelet method using the 1024x1024 grid achieves better compression. When Tallat used 10^{-3} thresholding, he was only able to compress to a 16.41 ratio [p47]. To compare, when 10^{-3} thresholding was used with the bior3.1 wavelet, the compression ratio was 82.009.

2-D data vorticity data sets using "high" compression <i>(uses bior3.1 wavelet. 1×10^{-4} threshold)</i>				
Data Size	Compression Time	Compression Ratio	Max Absolute Error	Max Relative Error (limit > 1×10^{-6})
64x64 grid	.066s	2.038	1.414×10^{-4}	71.30
128x128	.090s	3.359	1.43×10^{-4}	24.633
256x256	.161s	6.775	1.359×10^{-4}	12.42
512x512	.390s	16.901	2.218×10^{-4}	8.52
1024x1024	1.474s	41.624	7.965×10^{-5}	6.33
2500x2500	21.21s	295.376 (.33% of original)	4.67×10^{-5}	8.27

Low compression clearly doesn't do as well as high compression, but maintains an impressive maximum error rate.

2-D data vorticity data sets using “low” compression

(uses *bior5.5* wavelet (more coefficients than *bior3.1*). 1×10^{-7} threshold)

Data Size	Compression Time	Compression Ratio	Max Absolute Error	Relative Error (limit > 1×10^{-6})
64x64 grid	.071s	.6978	9.2×10^{-8}	.0379
128x128	.114s	1.350	8.95×10^{-8}	.0368
256x256	.222s	2.393	9.27×10^{-8}	.0503
512x512	.601s	5.678	9.23×10^{-8}	.0499
1024x1024	2.133s	16.11	8.013×10^{-8}	.0260
2500x2500	20.33s	23.03	1.389×10^{-7}	.013

Upon visual inspection, it becomes clear that the wavelet compression has not qualitatively affected the data. Figure 7 shows the reconstructed and original

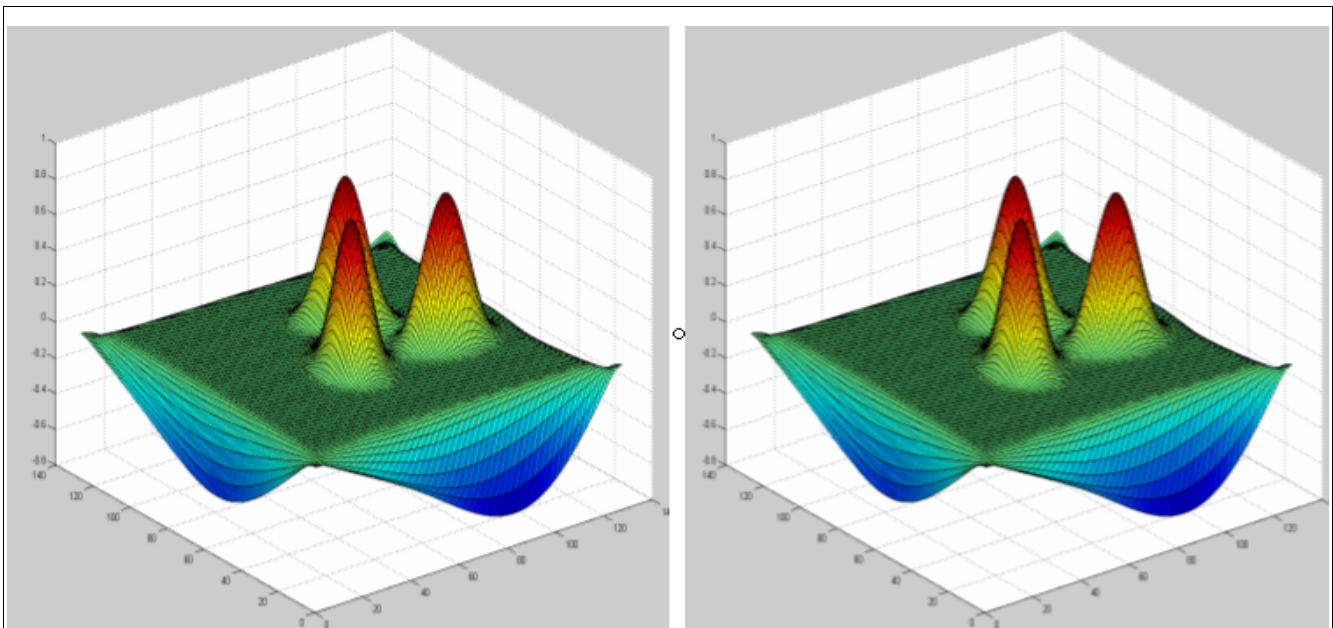


Figure 7: Original and Reconstructed (128x128 Grid)

128x128 vorticity grids for high compression, while Figure 8 shows the absolute error between them. Note that the scale on the error is 1×10^{-4} . Figure 9 shows the relative error.

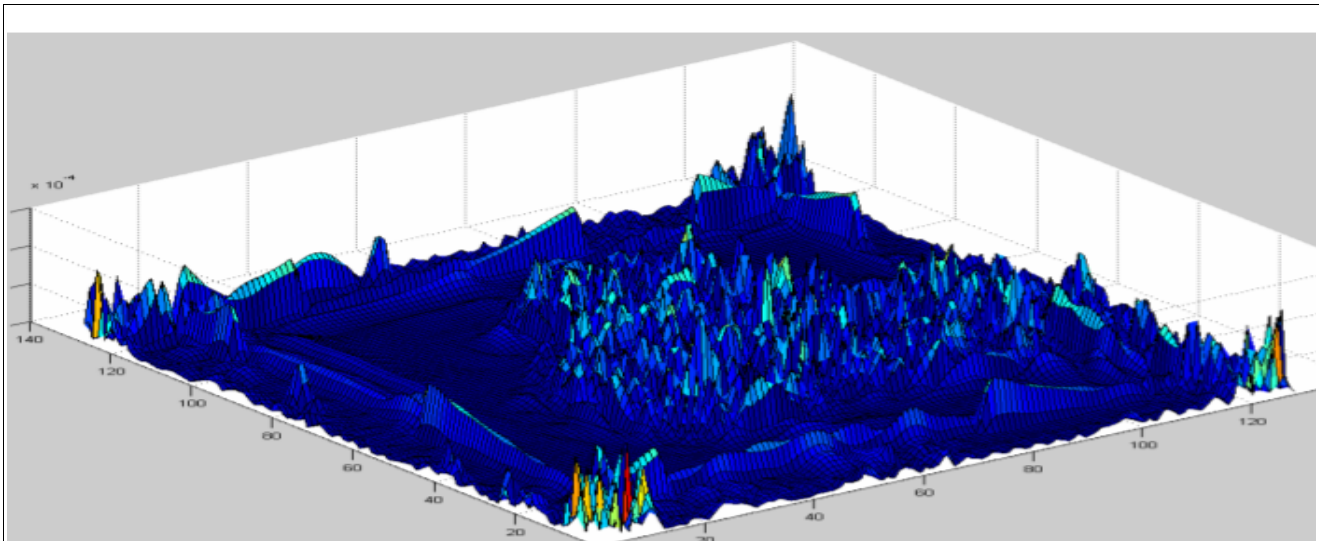


Figure 8: Absolute error between grids

The above results show 2-D data. 1-D and 3-D data faired similarly. Their results are omitted for brevity.

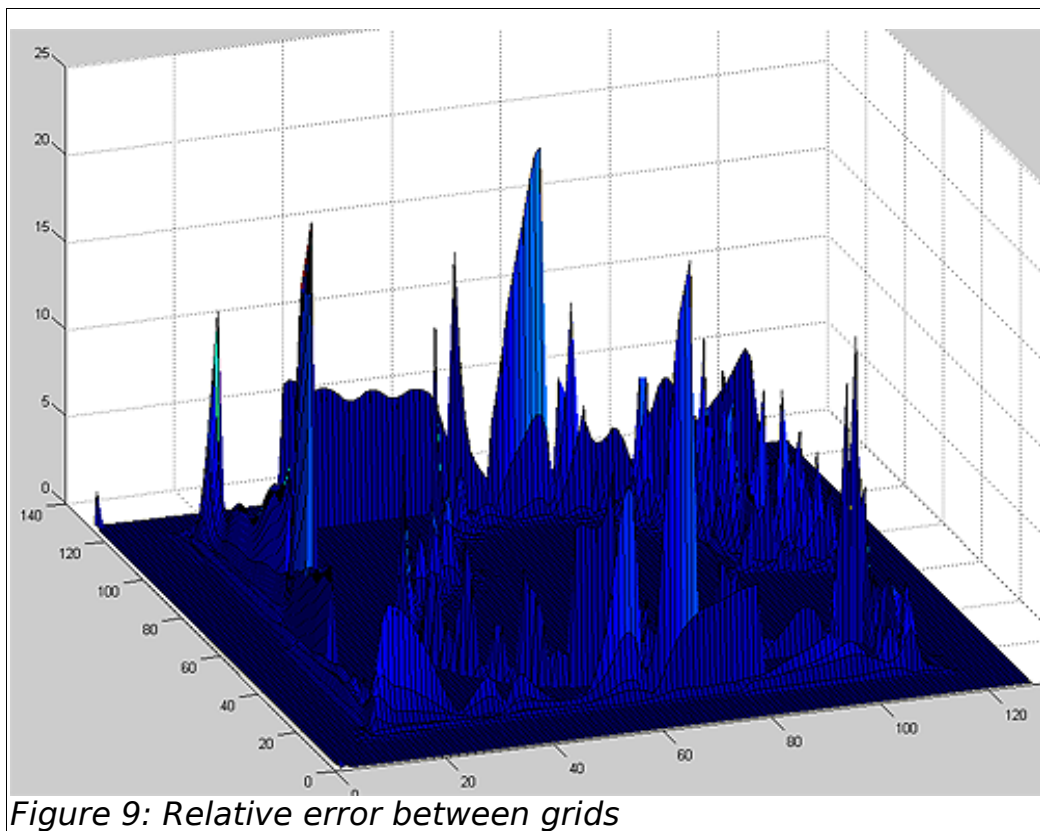


Figure 9: Relative error between grids

Although, it was known early on in the implementation that the parallel processing would be slower than the single node processing, measuring the processing time illustrated how badly it performed. Table 2 compares single node and parallel processing.

Data Size	Parallel Processing	Single Node
64x64 grid	.880s	.066s
128x128	2.64s	.090s
256x256	8.76s	.161s
512x512	33.49s	.390s

6. Conclusions

Lossy compression is applicable for many kinds of data, but it is still imperative the user has a basic understanding of the thresholding required. Wavelets are a good choice for doing such compression, as evidenced by other applications, such as image compression, and these results. The compression and decompression applications created are a set of capable and robust tools that would be useful for many scientific datasets.

7. References

1. Davis, Geoff. "Wavelet Construction Kit."
<http://www.geoffdavis.net/dartmouth/wavelet/wavelet.html>
2. Matlab Wavelet Toolbox.
<http://www.mathworks.com/access/helpdesk/help/toolbox/wavelet/wavelet.html>
3. Mix, Dwight and Olejniczak, Kraig. Elements of Wavelets for Engineers and Scientists. Wiley. 2003.
4. Polikar, Robi. "Wavelet Tutorial."
<http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html>
5. PyWavelets. <http://www.pybytes.com/pywavelets/>
6. Tallat Mahmood Shaffat, Context Dependent Compression using Adaptive Subsampling of Scientific Datasets, MS Thesis, Royal Institute of Technology (KTH) School of Information and Communication, Stockholm, Sweden, 2006. ICT/ECS-2006-12
7. Wickerhauser, Mladen. Adapted Wavelet Analysis from Theory to Software. AK Peters. 1994.

8. Appendix

8.1 Matlab Function Usage

```
[savings] = WAVECOMPRESS(x,mode,outputfile)
% compresses the 1-D data in x using the mode specified and saves
% to outputfile. The return value is compression ratio achieved

[savings] = WAVECOMPRESS2(x,mode,outputfile)
% compresses the 2-D data in x using the mode specified
% and saves to outputfile. The return value is compression
% ratio achieved

[x] = WAVEDECOMPRESS(inputfile)
```

```
% takes the compressed inputfile and returns a reconstructed
% 1-D signal x
```

```
[x] = WAVEDECOMPRESS2(inputfile)
% takes the compressed inputfile and returns a reconstructed
% 2-D signal x
```

8.2 C++ Application Usage

wavecomp

Usage: wavecomp [options] source

Options:

```
-c          compression mode (Req.)
             (1=high compress/high error, 2=medium/low, 3=low/low)
-d <D>      dimension of data (<D>= 1, 2, 3) (Req.)
-l <lim>     zero limit (ex: .0001 removes all abs(values) < .0001) (Opt.)
-p -m -s <IP> <IP> 3-Node parallel mode where this machine (Opt.)
             is master and 2 slaves are <IP> <IP> (Opt.)
-p -s       Parallel mode where this machine is a slave (Opt.)
-v          verbose (Opt.)
-i          data format instructions (Opt.)
```

wavedec

Usage: wavedec source

Options:

```
-v          verbose
```