Final Report: Implementing Compressive Sampling Algorithms Across The Cloud

Jason Halpern

December 9, 2012

1 Motivation

The purpose of this project is to explore various implementations of compressive sampling algorithms and to find ways of parallelizing these algorithms across several machines. If we can efficiently parallelize compressive sampling recovery algorithms, then we can handle the decompression of these signals across the cloud, which will lead to a large reduction in bandwidth. These new approaches will represent a significant advancement over the way that current wireless sensor networks handle the compression and decompression of signals.

2 Project Objective

Compressive Sampling is about acquiring and recovering a sparse signal in an efficient manner. It involves finding the number of measurements that are necessary to reconstruct the signal. The goal of the project is to research and explore several compressive sampling algorithms and to determine which algorithm is the best choice to implement in a distributed manner. After choosing an algorithm and implementing it in Java, we will look for ways to distribute the algorithm and we hope to be able to reconstruct an image across the cloud using our recovery algorithm.

3 Choosing a Compressive Sensing Algorithm

When we first started this project, we decided that we were going to choose one specific compressive sensing recovery algorithm and to focus on implementing that algorithm in Java. After implementing the code in Java, we would later find ways to distribute that algorithm across a Hadoop cluster. The first thing I did for the project was to read the paper, *An Introduction to Compressive Sampling* by Candes and Wakin [3] to get a better grasp on the significance of compressive sampling. I also read other short reports and presentations about compressive sensing to give myself a better background on the topic. The next step was to find a specific compressive sensing algorithm to implement. We mainly evaluated algorithms that we found on the Rice Computer Science website [4] in their model based compressive sensing toolbox.

We evaluated the following algorithms in the toolbox: 2D trees, cosamp, block sparsity and clustered sparsity. I first randomly picked the block sparsity algorithm and closely walked through the Matlab code line by line to make sure the mathematical complexity of the algorithm would not be too difficult for us to understand. I then took a quick look at the other algorithms to quickly gauge their complexity as well.

I also looked at several other factors when evaluating the algorithms. Two of the main factors were the amount of documentation in the code and the inclusion of a research paper that we could read in conjunction with the code. We chose the CoSaMP algorithm because it was well documented, complete and included the following paper, *CoSaMP: iterative signal recovery from complete and inaccurate measurements*, written by Needell and Tropp at the California Institute of Technology [5]. In addition, from reading the paper we learned that this algorithm is a greedy pursuit and is a good balance in terms of its running time and sampling efficiency. Most algorithms tend to be extreme in one of those categories. The CoSaMP algorithm is also the only signal reconstruction algorithm that accomplishes

all of the following: 1) It should accept samples from a variety of sampling schemes; 2) It should succeed using a minimal number of samples; 3) It should be robust when samples are contaminated with noise; 4) It should provide optimal error guarantees for every target signal; 5) It should offer provably efficient resource usage. For all of the reasons mentioned above, we chose to implement the CoSaMP algorithm.

4 Choosing a Java Library

The next step in the process was to pick a Java library that would allow us to carry out the necessary matrix operations, including matrix multiplication and transpose. The different libraries that I evaluated were Berkeleys Colt and Parallel Colt, ojAlgo, Mahout, Matlab Builder JA, Apache Commons Math, JBlas and EJML. The first library I looked at was Matlab Builder JA, which transforms Matlab code directly into Java code. Unfortunately, since Columbia does not have a license for this software we were unable to use it. I also decided against using ojAlgo and EJML because there is not a strong community of users around these libraries and they do not seem to be well supported anymore. I also eliminated JBlas even though it has done very well in benchmark performance tests against the other libraries, because it also did not seem to have strong support. It also was not clear from the documentation if all the operations that we needed were included in this library. In the end, I was deciding between Mahout [2] and Apache Commons Math. Both libraries seemed to have all the operations we needed and they also both have strong developer and user communities. The reason I chose Mahout is that it is mainly used to carry out distributed computations on top of Hadoop [1] and since an objective for this project was to distribute the compressive sensing recovery algorithm, it made sense to use a library that is often used for distributed computation.

5 Using Mahout

Mahout [2] is Apaches open source machine learning project. An interesting aspect of Mahout is that it makes it simple to distribute algorithms such as classification and clustering. In the future, an effort can be made to use more of the machine learning features of Mahout [6] with the Hadoop [1] framework. We used the matrix capabilities in Mahouts Math library in order to migrate the code from Matlab to Java. Mahout has a built-in matrix object that allows us to do operations such as matrix multiplication, addition, subtraction, as well as transpose and other matrix manipulation functions.

6 Migrating Code From Matlab to Java

The major effort in the first phase of the project was migrating the code we have in Matlab for the cosamp algorithm into Java. This phase of the project required evaluating each line of code in Matlab, determining what operations were being done and then writing the Java code to carry out these functions. Sometimes this simply involved using built-in Mahout functions, but it often required writing functions in Java to carry out equivalent operations to what was being done in Matlab. For example, for the Matlab functions abs(), sort(), zeros(), randperm() and randn(), we had to write and then test functions in Java that did the same thing as these functions in Matlab. There were also many other matrix manipulations functions that involved slicing and rearranging matrices that we had to write in Java. For example, for the following line of Matlab code:

$x(v(1:K)) = randn(K, 1);$

I wrote the function in Java called setCellValues() that assigns the values in the randn() matrix to the cells of the matrix x that are indicated by the rows in matrix v.

7 Distributing the Algorithm

After the midterm presentation, our next goal was to examine the algorithm and to find ways to distribute the reconstruction of the matrix. During the midterm presentation, we showed how the CoSaMP algorithm was successful in reconstructing a column vector, but we needed to prove that it would also work for a full matrix. Therefore, we created a 1024x10 matrix and determined that the only way

to successfully reconstruct the entire matrix would be to do so one column at a time. During each of the ten loops, another column is reconstructed and these reconstructed columns are combined to create a new matrix that is almost identical to the original (allowing for a very small error rate). New measurements need to be taken on each turn of the algorithm based on the values of the column vector and this measurement matrix is used in the reconstruction algorithm. Although we did not get a chance to distribute the matrix reconstruction using Hadoop, we have laid out a foundation for how that distribution could be accomplished.

Another important step during this phase of the project was that we made the algorithm more dynamic. The original Matlab algorithm, which was identical to what we presented during the midterm presentation, included hard-coded figures for signal length, signal sparsity and the number of measurements necessary for reconstruction. One of the key improvements to our algorithm is that we dynamically set the signal length at the beginning of the algorithm and dynamically set the signal sparsity at each turn based on the sparsity of the column vector. These changes are an improvement over the original Matlab code and were necessary for a full matrix reconstruction to work properly since the sparsity of individual columns might be different. The one number that is still hard-coded is the number of measurements needed for the reconstruction. It was unclear how this number is determined, so it made sense to leave it as is for the time being.

8 Discrete Wavelength Transform

One of our goals for the project was to be able to reconstruct an image using our algorithm. We found a different algorithm in the Rice Compressive Sensing toolbox that reconstructed an image using discrete wavelength transforms [7]. This algorithm uses Daubechies wavelets. Discrete wavelength transform is used for signal coding, to represent a discrete signal in a more redundant form, often as a preconditioning for data compression. We hypothesized that we might be able to use the first part of this algorithm to do the transforms and then we could use CoSaMP to recover the original image.

As we investigated this algorithm, it became clear that there were several important hurdles that we would not be able to overcome in the remaining time frame. Our objective was to integrate the wavelength transform with the CoSaMP recovery, but it turned out that the wavelength transform was more complex than we had original anticipated and it was not even clear if we would be able to integrate the two approaches. There were several steps in the wavelength transform that were difficult to implement in Java. For example, we could not find a function in Java that accomplishes what Matlab does in the rgb2gray() function. In addition, there were two or three Matlab functions (roots(), poly(), conv()) that I did not figure out how to implement in Java. They involve finding the roots of a polynomial. There could be elements to the 5th or 6th power and the solutions are often complex numbers, which made it difficult to find the right solution in Java.

9 Test Cases

I have used JUnit to write test cases for all the code. As I came across specific functions in Matlab that are not yet implemented in Mahout, I had to first determine what this function did and then I had to write the necessary code in Java to match the function in Matlab. Before writing the function, I would write a test case using JUnit that I could run after writing the function to make sure it was working properly. At the end of this report is a table that includes a Matlab function from the algorithm, a corresponding Java function I have written (in MatrixHelper.java) that is equivalent to the Matlab function and the test case I have written (in SignalTesting.java) for my code[1].

After we chose to move past the discrete wavelength transform for the time being, we decided that with the remaining time it made sense to focus on creating a robust framework for testing the algorithm and refactoring much of the existing code. This would ensure that the algorithm is complete and would make it much easier to extend or build upon the work we have done this semester in the future.

[1]Some parameters have been removed to make the table more readable

I have been writing unit tests over the course of the semester for the functions that I wrote to simulate Matlab functionality. There are currently 16 tests in SignalTesting.java that make sure these functions are doing the exact same thing as the corresponding Matlab code.

The recent addition to the testing framework is the test cases that we have created. We created an input folder with different test cases as input to the CoSaMP algorithm. There are several different test cases: two full matrices with different sparsities, two column vectors, a column vector with only zeros and a single cell with a zero. We also have an output folder that includes the results after the reconstruction, which we obtained by running the algorithm in Matlab. The test then runs a matrix created from each input file through the algorithm and compares this matrix to the expected matrix in the output folder. As long as the difference between values in each cell is within our error rate, the test passes. The good thing about the structure of our framework is that all you have to do is add an input file and an expected output file and the test will automatically run the algorithm on that matrix and compare the results.

The last step in the project was to refactor the existing code/classes and to add more documentation. I refactored the Signal class to be much more clean and to have more concise methods. I also added more documentation to the individual matrix functions and to the CoSaMP specific methods. When appropriate, I added the corresponding line of Matlab code to give better insight into the algorithm.

10 Conclusion

Overall, I think we had a lot of success in this compressive sampling project. We were able to successfully implement the CoSaMP algorithm in Java in exactly the same manner as it operates in Matlab. We thoroughly researched the algorithms and Java libraries before we started the project and I strongly believe that our choice to implement the CoSaMP algorithm using Mahout were smart decisions and paved the way for our successful implementation. It took us longer than we anticipated to move the code from Matlab to Java, which meant we had less time than we had hoped to experiment with distributing the matrix and we never got a chance to use Hadoop in the project. In addition, we never had the opportunity to apply the algorithm to an image due to the complexity of wavelength transforms. That said, not only does our algorithm work properly, but there are several improvements over the Matlab code. First, our code works on complete matrices whereas the Matlab code only works on column vectors. Second, we set certain properties of the algorithm (sparsity, signal length) dynamically based on the matrix being reconstructed, whereas the Matlab code has hard-coded values for all of these properties. Third, our code design is more clear than the Matlab code, our methods are more concise and our documentation is more thorough. Fourth, we have robust unit tests and testing framework that will make it easy to extend the work we have done this semester in future projects.

I thoroughly enjoyed doing this project and learned a tremendous amount over the course of the semester. It was very interesting to learn about compressive sampling and I also believe that it was great experience in software design and the importance of testing as well. It was great to work with Marcin and YoungHoon and they did a great job of managing the project and providing me with guidance and resources at every stage of the project. Even though the progress we made was different than what we had mapped out in the original proposal, I think we made smart decisions in adapting the project over the course of the semester.

11 Timeline

September 27th - Finished analyzing the different compressive sensing recovery algorithms that we had in Matlab. Read the research papers associated with the various compressive sensing algorithms. Decided to implement the CoSaMP algorithm.

October 4th - Evaluated many different Java libraries for doing matrix operations and decided to use Mahout. Setup Mahout and ran some tests using Mahouts matrix functionality to become more familiar with the library.

October 11th - Started to migrate the code from Matlab to Java and started to write test cases for the new functions I wrote.

October 27th - Finished migrating all of the code

from Matlab to Java and wrote test cases for all the new functions I have written.

November 6th - Modified the algorithm such that it works on full matrices now as opposed to just column vectors. Modified aspects of the algorithm to be more dynamic, including setting the signal length and sparsity. The full matrix is reconstructed on column at a time in a loop. Distinct measurements are taken of each column before it is reconstructed.

November 18th - Attempted to reconstruct an entire image using a combination of the discrete wavelength transform and the CoSaMP implementation. Had difficulty moving certain aspects of the wavelength transform to Java.

November 23rd - Refactored a lot of the code. Created more concise methods, renamed some variables, created a Signal class to better encapsulate the information specific to the signal matrix. Modified the methods related to the CoSaMP algorithm to be more clear.

November 30th - Created a testing framework that makes it easy to add new test cases without modifying the code. Created input files and results files and made sure the new framework works correctly.

December 8th - Added more documentation, finished refactoring. Began to work on final report.

References

[1] Apache. Hadoop. [Online]. `http://hadoop.apache.org`.

[2] Apache. Mahout. [Online]. `http://mahout.apache.org`.

[3] E. Candes and M. Wakin. An introduction to compressive sampling. *Signal Processing Magazine, IEEE*, 25(2):21–30, Mar. 2008.

[4] C. Hegde. Dsp: Model-based compressive sensing toolbox v1.1. [Online]. `http://dsp.rice.edu/software/model-based-compressive-sensing-toolbox`.

[5] D. Needell and J. A. Tropp. CoSaMP: iterative signal recovery from incomplete and inaccurate samples. *Commun. ACM*, 53(12):93–100, Dec. 2010.

[6] S. Owen, R. Anil, T. Dunning, and E. Friedman. *Mahout In Action*. Manning Publications, 2012.

[7] Wikipedia. Discrete wavelength transform. [Online]. `http://en.wikipedia.org/wiki/Discrete_wavelet_transform`.

Table 1: **Migrating Code From Matlab to Java**

Matlab	**Java**	**JUnit**
sort(Descending)	sortDescending()	sortDescTest()
abs()	getAbsMatrix()	absValueTest()
ne(scosamp,0)	notEqual()	notEqualTest()
find(, 0)	findNonzero()	findNonzerosTest()
union()	union()	unionTest()
[M,N] = size(Phi)	rowSize()/columnSize()	n/a
yy(:)	toSingleColumn()	singleColumnTest()
ww(1:K)	getIndices()	indexTest()
bb2(ttcosamp) = wcosamp	setCellValues()	n/a
xcosamp(:,kk)	getColumn()	getColumnTest()
Phi(:,ttcosamp)	getColumns()	getColumnsTest()
zeros()	fillWithZeros()	n/a
xcosamp(:,kk) = scosamp	modifyColumn()	modifyColumnTest()
norm()	norm()	normTest()
randn()	randN()	n/a
randperm()	randomPermutation()	n/a
length()	length()	lengthTest()
xcosamp(:,kk+1:end)=[]	removeColumns()	removeColumnsTest()
xhat = xcosamp(:,end)	getLastColumn()	getLastColumnTest()

Table 2: **Description of Matlab Functions**

Matlab	**Description**
sort(Descending)	Sort each column in the matrix so the values are in descending order
abs()	Converts each element in the matrix to its absolute value
ne(scosamp,0)	If a cell in a matrix does not equal zero then set its value to 1, otherwise zero
find(, 0)	Return a matrix with the row and column location of all nonzero values
union()	Form a vector that is the union of two other vectors
[M,N] = size(Phi)	M is the number of rows and N is the number of columns
yy(:)	Transform an entire matrix into a single column
ww(1:K)	Return a matrix with columns 1 through K
bb2(ttcosamp) = wcosamp	Set the columns indicated in ttcosamp of the bb2 matrix to the values in wcosamp
xcosamp(:,kk)	Return a column vector of the kk column
Phi(:,ttcosamp)	Return a matrix of columns from the Phi matrix as indicated in ttcosamp
zeros()	Fill the matrix with all zeros
xcosamp(:,kk) = scosamp	Modifies the kk column to the values indicated in scosamp
norm()	The norm function calculates several different types of matrix and vector norms.
randn()	Returns a matrix of pseudorandom values from the standard normal distribution.
randperm()	Returns a row vector of a permutation of integers from 1 to the signal length.
length()	Returns either the number of rows or the number of columns, whichever is higher
xcosamp(:,kk+1:end)=[]	Remove all columns between (kk+1) and end
xhat = xcosamp(:,end)	Return the last column of a matrix

Table 3: **Execution Times - Matlab vs. Java (in seconds)**

Matrix Size	**Java**	**Matlab**
1024x1	2.564	0.166
2048x1	4.23	0.2963
4096x1	9.83	0.61
8192x1	memory error	1.13