FFT Transform of audio files with MPI

Alexander Gustafson
ZHAW 2013
alexg.hsz@gmail.com

ABSTRACT

The goal of this seminar project was to implement a cross platform application in C++ that would analyze audio files using the Fast Fourier Transformation. One of the requirements of the seminar project was to use MPI in order to optimize the performance in a multiprocessor or multicore environment.

1. INTRODUCTION

The Fourier Transformation is a method of approximating a function or signal by breaking in into sin waves or varying frequency and phase. For mathematical functions the Fourier Transformation can result in an infinite amount of sinusoid components of increasing frequency. For discreet signals like digital audio recordings or images the Fourier Transformation has an upper bound defined by the sampling frequency or resolution of the signal. The Nyquist-Shannon sampling theorem states that a signal can only contain frequencies up to half the sampling rate. In order to simplify the project a little, I have made the assumption that all audio signal to be analyzed will have a fixed sampling rate of 44.1KHz.

Formally the Fourier Series Equation is defined by the following equation:

$$f(t) = \sum_{n=-\infty}^{\infty} \left(c_n \, e^{-(\mathrm{j}\omega_0 \mathrm{nt})} \right) \tag{1}$$

This states that a function f(t) can be described as the sum of a series of sinusoid components, where c_n indicates the contribution that a sinusoid at frequency ω_0n makes to the function f(t) . If c_n is small for large ω_0n this indicates that the function will be "smooth", and vary slowly over time.[1]

For discrete or sampled signals the Fourier Transformation has limits imposed by the "quality" of the signal to be analyzed, this bounded transformation is called the Discrete Time Fourier Series:

$$f[n] = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \left(c_k \, e^{-j \frac{2\pi}{N} k n} \right) \tag{2}$$

Where N is the number of samples in f[n]. For the analysis, what really interests us are the c_k terms, as these describe the how much of a frequency component at step k exists in the signal. So we can rewrite equation 2 in the following:

$$c_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} \left(f[n] \, e^{-j \frac{2\pi}{N} k n} \right) \tag{3}$$

For k = 0, c_k will tell us how much or a zero-frequency signal, or DC-Offset, the signal contains. c_k at k = 1 will tell us how much the sinusoid with period of length N contributes to the signal. At k = N - 1, c_k is the factor that the sinusoid with a period of length 2/N (cycling every 2 samples) contributes to the signal. Any sinusoid cycling faster than 2 samples would not be able to be represented accurately because it would be cycling "between" the samples and appear in the context of the signal to be slower ("aliasing"), hence it cannot be used for the analysis.

2. WINDOWING

Using equation 3 to analyze an audio file will result in N c_k terms, where c is a complex number. The real component tells us the amount of a signal, the imaginary component indicates the phase. From this we can gain information about the content of the signal. However, this information is static and does not give any insight into how the signal is changing over time. Also, consider a 1 minute audio recording, with a sampling rate of 44.1KHz. N will be 44100*60, giving us the coefficients of 2646000 frequency steps between 0 and 22.5KHz. This gives us a resolution orders of magnitude smaller than the human ear is able to perceive[2], so it is also too much static information.

The solution is to slice the audio signal into blocks and transform these individually. Considering that human hearing is in the range of 20Hz - 20KHz we have lower limit to what is usefull. A 20Hz signal will cycle in 1/20th of a second so N will need to be 44100 / 20, or 2205 samples. As we will see later, the optimized implementation of the Fourier Transformation needs N to be a power of 2. 2048 will give us a lower frequency limit of 21.5 Hz, this should be good enough for our purposes. It is important when choosing the size of the slices to understand how the size effects the result.

However, due to the periodic slicing, some frequency components will not lie completely within a single slice. At the ends of the slice, the original signal will become discontinuous. This will cause artifacts in the Fourier analysis known as leakage.[3] Frequencies will appear to be smeared in the frequency domain. The solution is to smooth out the edges of the blocks, and to use overlapping blocks. Optimally the overlapping smoothed segments should produce the original signal when added together. The Harris Blackman [4] window produces good results with 4 x overlapping.

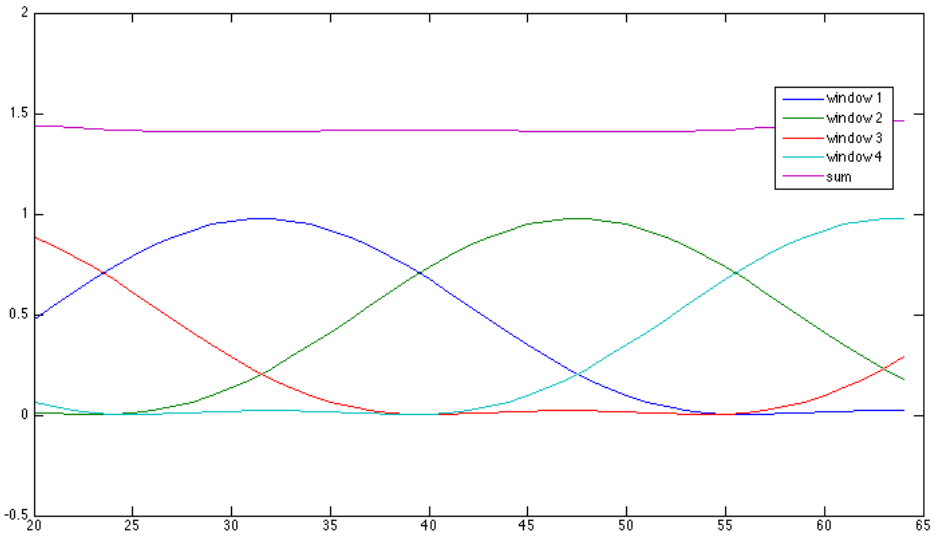

Figure 1 : Overlapping Harris Blackman window function with sum

3. FAST FOURIER TRANSFORMATION

A naive implementation of the Fourier Transformation could be implemented with two for loops. The outer loop iterates k over the frequency steps from 0 to N. The inner loop iterates n from 0 to N over the length of the sample Block. Within the inner loop each sample is multiplied by the sine function value at n for a sine with frequency k/N, the products are summed. The result is the real coefficient of the DFT for the step k. The sum of the products with the cosine function result in the imaginary component.

```
for (int k = 0; k <= N; k++) {
    for(n = 0; n < N; n++)
    {
        real[k] += samples[n] * -sin(2*k*pi*n / N);
        img[k] += samples[n] * cos(2*k*pi*n / N);
    }
}
```

It 's easy to see that we will be calculating lots of sine and cosines for large values of N. An good first step towards optimizing the Fourier calculations would be to pre-compute the sine and cosine values and store these in an N x N matrix. We can reuse this matrix for every block of samples we process, so it 's a good idea to keep the matrix beyond the scope of a processing a single block of samples.

The FFT takes advantages of relationships and redundancies in the calculation of the Fourier coefficients. Looking at the DCT we can split the summation into even and odd steps (the factor $\frac{1}{\sqrt{N}}$ has been removed to simplify the equation, consider it to be factored into c_k).

$$f[n] = \sum_{k=0}^{N-1}\left(c_k\, e^{j\frac{2\pi}{N}k\,n}\right) = \sum_{m=0}^{N/2-1}\left(c_{2\,m}\, e^{-j\frac{2\pi}{N}2\,m\,n}\right) + \sum_{m=0}^{N/2-1}\left(c_{(2\,m+1)}\, e^{-j\frac{2\pi}{N}(2\,m-1)\,n}\right) \tag{4}$$

We can factor a constant term out of the odd steps to simplify.

$$f[n] = \sum_{m=0}^{N/2-1}\left(c_{2\,m}\, e^{-j\frac{2\pi}{N/2}2\,m\,n}\right) + e^{-j\frac{2\pi}{N}n}\sum_{m=0}^{N/2-1}\left(c_{(2\,m+1)}\, e^{-j\frac{2\pi}{N/2}m\,n}\right) \tag{5}$$

Now we have 2 two sets of calculations to perform, but each with length N/2. Each of these can also be split into even and odd components and summed. Repeating this recursively leads us to an cost model of $O(5N\log_2 N)$ as opposed to $O(\,8\,N^2\,)$. [5] In this application it will be $O(\,8*4\,N^2\,)$ because of the 4 time overlapping windowing process.

3. IMPLEMENTATION WITH MPI

Due to the divide and conquer nature of the FFT, the FFT algorithm itself is a good candidate for parallel processing, especially for calculations with large values of N. For this project though, will be using a more trivial parallelization by slicing the audio signal into blocks, each of which is distributed to a worker node. The worker nodes calculate the transform and send the result back to a central dispatch node which uses the data to create a spectra graph of the audio signal.

The node with rank 0 is be the master node, dispatching data, receiving the results and sending these to the spectra graph view object.

In order to centralize the MPI handling I implemented an MPIHandler singleton class which was suppose to simplify the handling of MPI initialization and MPI calls. However, later in the project as I became for familiar with MPI handling I realized this was not necessary.

I used the Juce C++ cross platform framework for GUI and audio file management. It proved to be quite difficult to use Juce in an MPI context. Juce hides many of the application startup processes in an atempt to make application programming easier for the user, especially because it removed any plaform specific issues of running an GUI based application. It was difficult to figure out where to place the MPI checks to determine if the current running application is the one with rank 0, and start or skip the creation of the main application GUI process.

In the Juce project Main.cpp file there is a macro:

```
// This macro generates the main() routine that launches the app.
START_JUCE_APPLICATION_MPI (JuceTestApplication)
```

By commenting out this macro and adding a custom main() method such as this:

```
extern "C" int main (int argc, char* argv[])

{

    //boost::mpi::environment env(argc, argv);
    MPI_Init(&argc, &argv);

    MPIHandler* mpiHandle = MPIHandler::getInstance();

    if (mpiHandle->getRank() == 0) {

        std::cout << "my rank " << mpiHandle->getRank() << std::endl;

        juce::JUCEApplication::createInstance = &juce_CreateApplication;

        return juce::JUCEApplication::main (JUCE_MAIN_FUNCTION_ARGS);

    }else{

        doWorker();

    }

}
```

It is possible to route the program to either start the Juce based application code or to initiate the doWorker() code which creates instances of MPI workers that wait for data to process.

4. THE PROGRAM

The Appliction FFTMPI starts with a single window with the option to select an audio file, process it, and to choose it the FFT algorithm or the naive DFT algorithm should be used for calculation. The option to use FFT or DFT can be changed while an audio file is being processed in order to see the difference in speed.

After selecting an audio file, it's waveform will be displayed in the upper half of the window. The lower half will display the corresponding spectra graph, and will be updated during the calculation.

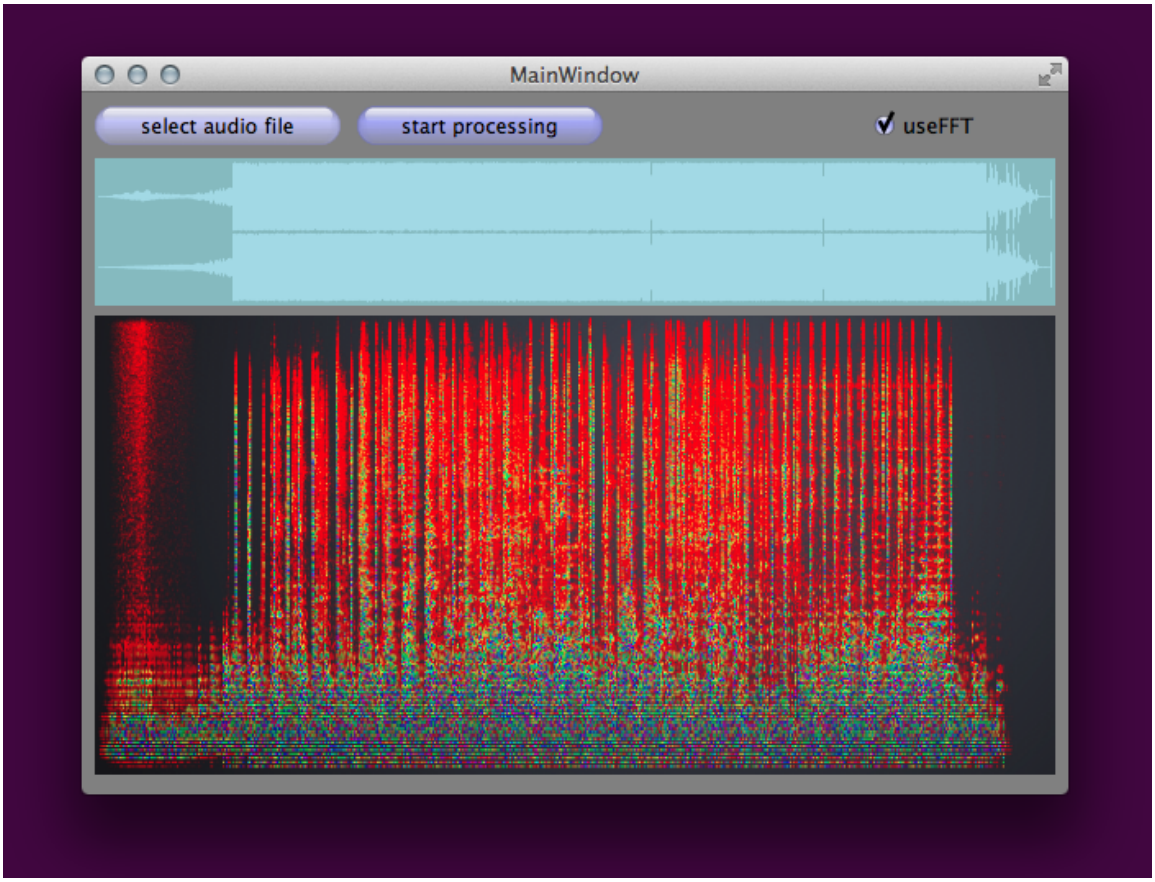

Figure 2: Application Window GUI

When "start processing" is clicked a thread in MainViewComponent is started which gets the number of mpi processor. A while loop is started that compares the current sample position in the audio file that has been processed with the total number of samples. In the while loop a for loop is started, each mpi processor is given a batch of samples to process. Each batch is offset from the previous batch by 1/4 the batch length. After data is sent to all the nodes, the for loop is exited and another for loop begins where the the results are gathered. When data is received from a worker node, the data array is given to the SpectraViewComponent which is responsible for updating the spectra graph disply.

Dispatch Thread

```
void MainViewComponent::run()
{
    int numOfProcessors = mpiHandle->getNumberOfProcesses();
    ScopedPointer<AudioFormatReader> audioReader ( formatManager.createReaderFor (currentFile));

    while(currentSamplePosition < audioReader->lengthInSamples)
    {

        if(numOfProcessors > 1)
        {
            for (int i = 1; i < numOfProcessors; i++) {

                audioReader->read(audioBuffer, 0, sliceSize,  currentSamplePosition, true, true);
                mpiHandle->send(i, msg_bufferSize, sliceSize); //send size of N
                mpiHandle->send(i, msg_usefft, fftButton->getToggleState()); //should use DFT or FFT ?
                mpiHandle->sendSampleBuffer(audioBuffer->getSampleData(0), sliceSize, i); //send data slice

                currentSamplePosition += sliceSize / 4;

            }

            for (int i = 1; i < numOfProcessors; i++)
            {
                float* samples = new float[sliceSize];
                MPI_Status status;
                MPI_Recv(samples, sliceSize/2, MPI_DOUBLE, i, msg_result_real, mpiHandle->world, &status);
                spectraViewer->addDFTData(samples);

                delete []samples;
            }
        }
    }
}
```

The worker processors, on construction do some small initialization then wait for information about the size of the buffers that will be worked with. Once the size is know, then further initialisation can continue. In case DFT processing might be required, all nodes go ahead and initialize the matrix for the DFT calculations. This is only done once per node. The FFT does not require initialization, but both the FFT and DFT require windowed data. performWindowing(samples, size) muliplies the values of samples with the Blackman Harris window. The next step is to wait for instructions about the method to use, DFT or FFT, after that the audio data is received and the processing begins.

The FFT algorithm has some requirements. Firstly the length of samples must be a power of 2. In addition the FFT algorithm expects it input to be complex values, with the real and complex values interlaced in the array. For the FFT the following steps are performed:

```
initializeFFTBufferArray(samples, fftBuffer, size);
performFFT(fftBuffer, size, -1);
deInterlace(fftBuffer, size*2);
MPI_Send(&fftBuffer[0],size/2, MPI_FLOAT, 0, msg_result_real, mpiHandle->world);
```

initializeFFTBufferArray takes two arrays "samples" and "fftBuffer" and interspaces all the values of samples into fftBuffer with a zero value in between. The deInterlace function reverses the process, it takes every two values from fftBuffer, squares them, and puts the sum back into samples.

Samples is then sent back to node 0 to be further processed.

The actual FFT implementation is quite complex. It is a loop-structure version of the FFT instead of a recursive implementation with two parts. The first part performs a reverse-binary reindexing the second part is the Danielson-Lanczos section. [6] The key to the second part is that all the results of the calculation can be stored back in the original array, therefor it is memory efficient. The other feature of the Danielson-Lanczos section is that is only needs to call the sine and cosine functions once before the for loops. The other sine and cosine values needed for the Fourier calculation are calculated recurrently from these first two calculations. I did not understand exactly how this work, but it has to do with the fact that the nth roots of 1 on the complex plane are equally spaced vectors on the unit circle, so if we know the first one, we can deduce the next values without using the sine and cosine functions again. [7] I think this is similar to a rotation performed with matrix multiplication.

4. DISPLAYING THE FFT RESULTS

The results of the FFT algorithm are complex numbers representing the real and imaginary contribution of a sinusoid at a particular frequency. These two values are squared, summed, the square-root of the sum is taken, then the value is normalized by dividing by N. Taking the log10 of this value and multiplying by 20 would give us the value in dB.[8] I wasn't able to generate good pictures with a dB scale so I didn't include this calculation.

As the result vectors are handed to the SpectraViewComponent they are stored in a std::vector< float* > buffer. As there are generally alot more samples in a recording than pixels in the display, not every pixel result vector is drawn on the display. A ratio is calculated, expected number of result vectors vs the width of the display, and stored as xStep. Everytime xStep number of new result vectors is recieved, the display draws the newest vector on the display.

Similarly the number of frequencies to display might be much larger that the height of the window to draw into. The drawing of the result vector is scaled logarithmically so that more detail is given to the lower end of the spectrum, and higher frequencies are squished together toward the top. This help visualize those frequencies to wich er are more sensitive.

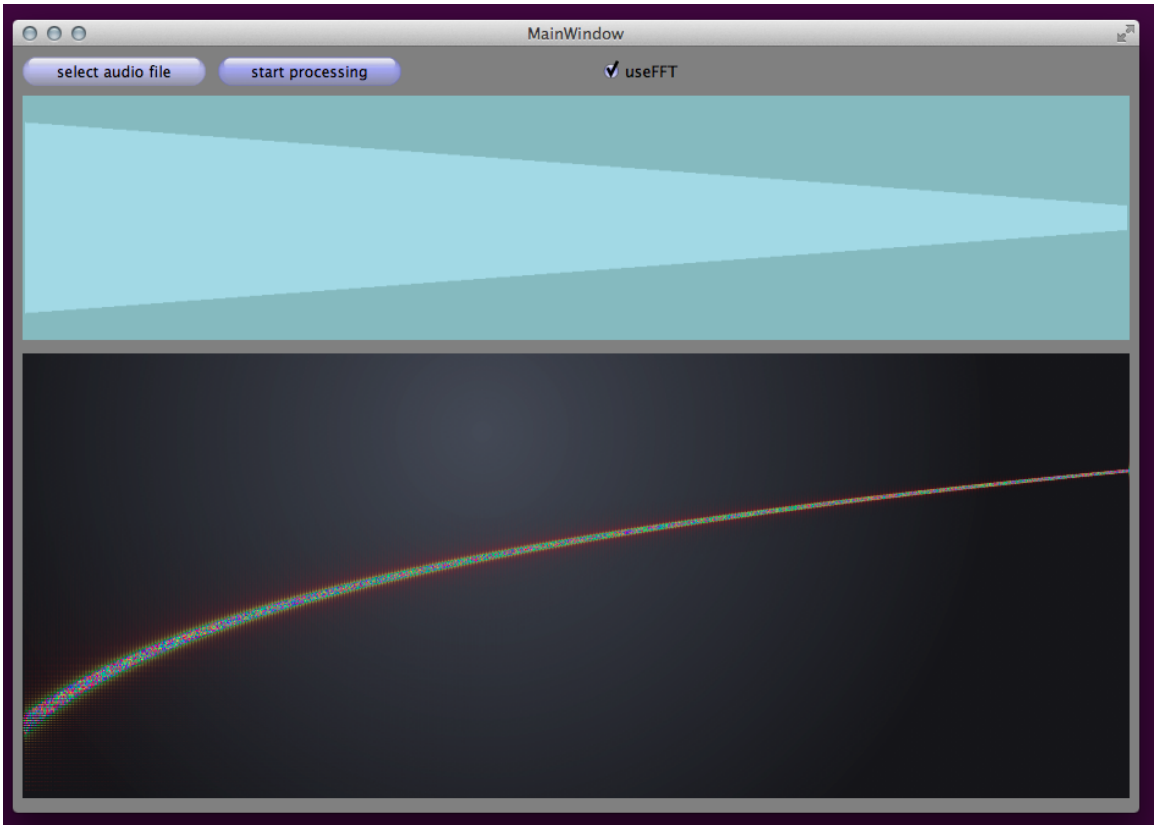

Figure 3 : sine wave with increasing frequency

5. TO DO

The application should be able to process more than one audio file. Currently some pieces of the application do not reset themselves properly to prepare for another audio file. The communication between the nodes must be extended to allow themselve to reset into the initial state.

The application does not properly shut down. When the main program shuts down it does not notify the nodes that they should shut down as well.

More user configurable settings. The color spectrum and brightness of the signal could be adjusted by the user. It would be nice to be able to zoom in on parts of the spectrum and audio signal.

Audio playback with position indication in the wave and spectrum displays. This would be nice to be able to hear the sound that corresponds to the fft display.

Batch fft analysis, and saving raw fft data to a file. This would be especially usefull for the purpose of being able to extract features of a signal and compare them to other audio files.

6. COMPILATION

In the Build/ directory of the project the is a Linux and MacOSX folder. The MacOSX folder has an XCode project that can be opened to compile with XCode. The MacOSX version expects boost and openmpi to be installed. The Linux verison also required boost and mpi to be installed. In addition the Juce framework has the following dependencies under Ubuntu:

```
sudo apt-get -y install g++
sudo apt-get -y install libfreetype6-dev
sudo apt-get -y install libx11-dev
sudo apt-get -y install libxinerama-dev
sudo apt-get -y install libxcursor-dev
sudo apt-get -y install mesa-common-dev
sudo apt-get -y install libasound2-dev
sudo apt-get -y install freeglut3-dev
sudo apt-get -y install libxcomposite-dev
```

I am currently not able to compile on Ubuntu, but I think this is a problem with my Boost configuration. I am working on this still.

References

1 R. Baraniuk. "Signals and Systems", p115, from Connexions, Rice University — http://cnx.org/content/col10064/latest.

2 http://en.wikipedia.org/wiki/Psychoacoustics

3 http://www.katjaas.nl/FFTwindow/FFTwindow.html

4 http://en.wikipedia.org/wiki/Window_function#Blackman.E2.80.93Harris_window

5 G. E. Karniadakis and R. M. Kirby II, Parallel Scientific Computing in C++ and MPI, Cambridge University Press

6 http://www.drdobbs.com/cpp/a-simple-and-efficient-fft-implementatio/199500857

7 http://www.katjaas.nl/rootsofunity/rootsofunity.html

8 http://www.dspdimension.com/admin/dft-a-pied/