CAB401

Parallel Digital Music Analysis

Vivek Thapar

N9770950

Contents

1.0 Introduction

1.1 Digital Music Analysis

The purpose of this program is to help determine highs (sharp) and lows (flat) of a given audio file by analysing the frequencies of the note played. The program displays a frequency diagram to visualise the audio file, displays the note octaves as the audio plays and gives detailed attributes of each note played.

1.2 The Program

The program first needs to take a sample audio file along with an XML reference file to compare to. The program samples the pieces of the audio file over a period of time and separates it into its individual frequencies using the Cooley-Tukey Fast Fourier Transform algorithm. The purpose to use an *fft* algorithm is because its ability to compute complex numbers efficiently. The UML class diagram in **Appendix A** shows the structure of the program.

2.0 Tools and Approach

2.1 Hardware

The computer used on which both programs were run and tested on had the following specifications:

Processor: Intel Xeon E5 1650

Clock Speed: 3.70GHz

Cache: 64KB L1 cache, 256KB L2 cache and 10MB L3 cache

Number of Cores: 4

Number of Threads: 8

Memory: 32GB DDR4 2133 MHz

2.2 Software

Since the program was written in C# I made use of some in built libraries and namespaces, namely the Diagnostics, namespace for tracking time and key events, and the Task Parallel Library (TPL). I used Visual Studios profiling tool to collect and analyse the programs CPU usage data.

2.3 Initial Analysis

After profiling it can be seen the *fft* functions is where the most time was spent in processing the data. This may be due to that the function was recursive which limits the amount of parallelism that could be achieved. This call/use of the *fft* function is being used in two places in the program. After initial observation it may be worth to only parallelise the loops which call said *fft* functions. **Appendix B** Shows an initial function call graph which indicate that *onsetDetection* and *fft* functions have the highest sample count. Figure 2.1 shows the overall CPU usage of the program which has direct correlation to the graph:

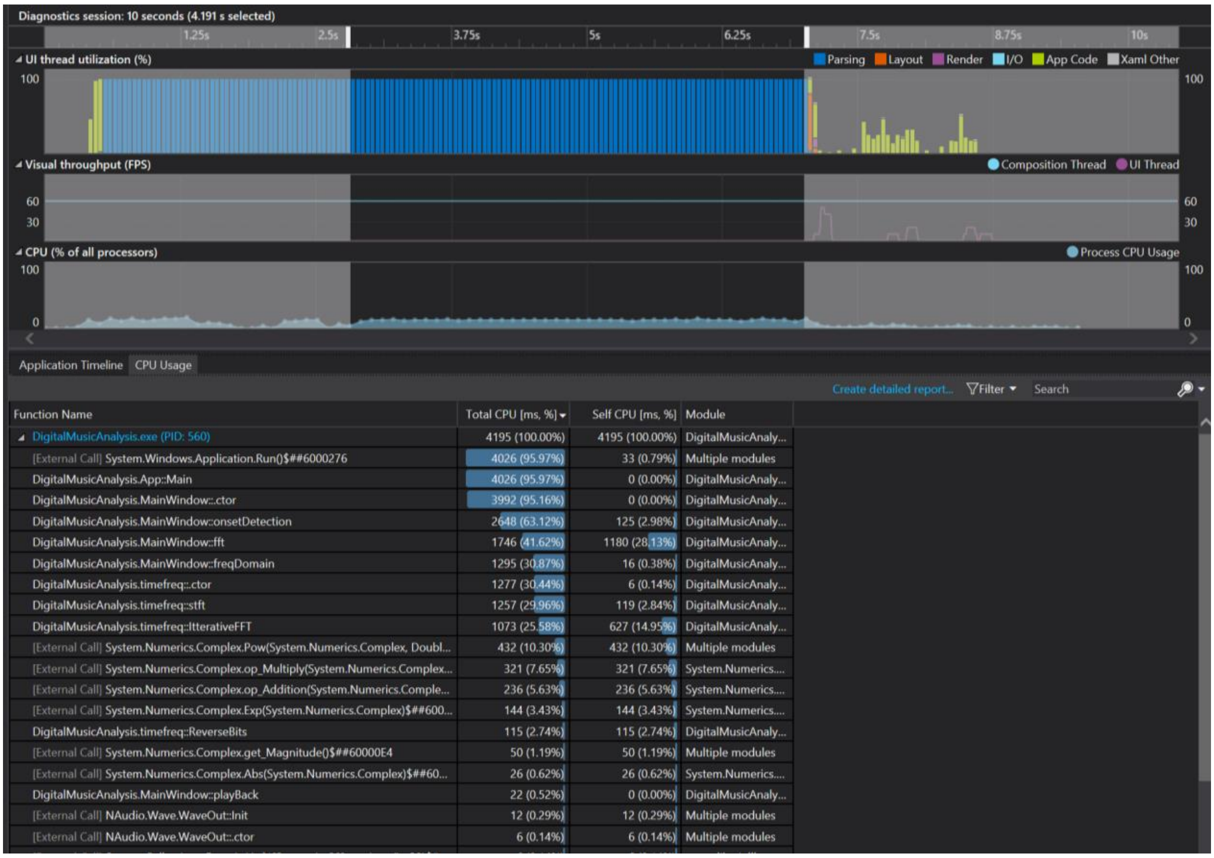


Figure 2.1: Function CPU usage

The onsetDetection function will be the main point of interest however in this assignment as it is not only calling an fft function, it also does its own calculations as well that may show worth in parallelising.

3.0 Methodology and Techniques

3.1 Algorithm Reconstruction

I redid the algorithm to make it run iteratively instead and there was a decent improving in run time using an altered version of the Cooley–Tukey iterative algorithm (**Appendix C**). My implementation of this algorithm differs slightly the algorithm needs a returning value for each sample from the audio file the result algorithm can be seen in below. The bit reversal loop was parallelised which showed only minor improvement.

```
using System;
using System.Numerics;
using System.Threading.Tasks;

namespace DigitalMusicAnalysis
{
    class FFT
    {
        private static int ReverseBits(int bits, int n)
        {
            int rev = n;
            int counter = bits - 1;

            n >>= 1;
            while (n > 0)
            {
                rev = (rev << 1) | (n & 1);
                counter--;
                n >>= 1;
            }

            return ((rev << counter) & ((1 << bits) - 1));
        }
```

```
        public static Complex[] IterativeFFT(Complex[] x, int L, Complex[] twiddles)
        {
            int N = x.Length;
            Complex[] Y = new Complex[N];
            int bits = (int)Math.Log(N, 2);

            Parallel.For(0, N, new ParallelOptions { MaxDegreeOfParallelism = MainWindow.numThreads }, i =>
            {
                int pos = ReverseBits(bits, i);
                Y[i] = x[pos];
            });

            for (int ii = 2; ii <= N; ii <<= 1)
            {
                for (int jj = 0; jj < N; jj += ii)
                {
                    for (int kk = 0; kk < ii / 2; kk++)
                    {
                        int e = jj + kk;
                        int o = jj + kk + (ii / 2);
                        Complex even = Y[e];
                        Complex odd = Y[o];

                        Y[e] = even + odd * twiddles[kk * (L / ii)];
                        Y[o] = even + odd * twiddles[(kk + (ii / 2)) * (L / ii)];
                    }
                }
            }

            return Y;
        }
    }
}
```

Figure 3.1: Iterative implementation of FFT algorithm

The reason behind using an iterative implementation of this was to potentially expose parallel loop. Secondly the iterative implementation performs less index computations which may explain the boost in performance. This set the bench mark for the best sequential version of the program.

3.2 Parallel Implementation

3.2.1 timefreq Class

Initial Analysis:

After profiling, it was evident that this class spent a lot of its time computing its twiddles value. In the constructor the loop had no dependencies. The main method called in this class was the *stft* function. This function needs to call an fft function to multiple times to get a frequency graph, so this was the main block of code I intended to parallelise.

Approach:

A parallel for loop was implemented to run the *twiddles* calculation block as fast as possible and there were no dependencies present in the loop to cater for. Figures 3.2 show the code before and after parallisation:

```
public timefreq(float[] x, int windowSamp)
{
    int ii;
    double pi = 3.14159265;
    Complex i = Complex.ImaginaryOne;
    this.wSamp = windowSamp;
    twiddles = new Complex[wSamp];

    for (ii = 0; ii < wSamp; ii++)
    {
        double a = 2 * pi * ii / (double)wSamp;
        twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
    }
```

```
public timefreq(float[] x, int windowSamp)
{
    double pi = 3.14159265;
    Complex i = Complex.ImaginaryOne;
    this.wSamp = windowSamp;
    twiddles = new Complex[wSamp];

    Parallel.For(0, wSamp,
        new ParallelOptions { MaxDegreeOfParallelism = MainWindow.numThreads }, ii =>
        {
            double a = 2 * pi * ii / (double)wSamp;
            twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
        });
```

Figure 3.2: Before and after code comparison of timefreq twiddles

To make the *stft* function run as fast as possible I decided to run each call to the *fft* function in parallel by running each on its own thread. The only dependencies here were the two temporary arrays of complex numbers which were simply defined in each method call. The code can be seen in Figures 3.3 and 3.4 of the before and after parallisation.

```
for (ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
{
    for (jj = 0; jj < wSamp; jj++)
    {
        temp[jj] = x[ii * (wSamp / 2) + jj];
    }

    tempFFT = ItterativeFFT(temp, wSamp);

    for (kk = 0; kk < wSamp / 2; kk++)
    {
        Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);

        if (Y[kk][ii] > fftMax)
        {
            fftMax = Y[kk][ii];
        }
    }
}
```

Figure 3.3: stft loop before parallisation

```
            Thread[] fftThreads = new Thread[MainWindow.numThreads];

            for (int i = 0; i < MainWindow.numThreads; i++)
            {
                int id = i;
                fftThreads[i] = new Thread(freqSTFT);
                fftThreads[i].Start(i);
            }
            for (int j = 0; j < MainWindow.numThreads; j++)
            {
                fftThreads[j].Join();
            }
```

```
        public void freqSTFT(object threadID)
        {
            int id = (int)threadID;
            int start = id * blockSize;
            int end = Math.Min(start + blockSize, size - 1);
            // dependencies
            Complex[] temp = new Complex[wSamp];
            Complex[] tempFFT = new Complex[wSamp];

            for (int ii = start; ii < end; ii++)
            {
                for (int jj = 0; jj < wSamp; jj++)
                {
                    temp[jj] = xx[ii * (wSamp / 2) + jj];
                }

                tempFFT = FFT.IterativeFFT(temp, wSamp, twiddles);

                for (int kk = 0; kk < wSamp / 2; kk++)
                {
                    Y[kk][ii] = (float)Complex.Abs(tempFFT[kk]);

                    if (Y[kk][ii] > fftMax)
                    {
                        fftMax = Y[kk][ii];
                    }
                }
            }
        }
    }
}
```

Figure 3.4: stft loop after parallisation

3.2.2 onsetDetection

Initial Analysis:

This function consumed a significant portion of the CPU usage as shown below:

Though it seemed not very impactful, the profiler showed that it sent 4.6% of time on the HFC calculation loop. This the looped had no dependencies to consider too. However, the more significant problem of the onsetDetection function is within the loop where another *fft* function as shown by the function call graph in **Appendix B**.

Similar to the timefreq class, onsetDetection also calculated some *twiddles* values so this was ideally the first point of interest. Secondly, this function also called fft which justifies as this being the main point of processing. It can be seen in Figure 3.6 shows the code for the sequential version and shows that that within the outer loop, contained 3 more for loops each dependant on the previous. The second loop required the *twiddles* values and the third loop required the generated complex numbers in the *Y* array.

```
for (int mm = 0; mm < lengths.Count; mm++)
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    twiddles = new Complex[nearest];
    for (ll = 0; ll < nearest; ll++)
    {
        double a = 2 * pi * ll / (double)nearest;
        twiddles[ll] = Complex.Pow(Complex.Exp(-i), (float)a);
    }

    compX = new Complex[nearest];
    for (int kk = 0; kk < nearest; kk++)
    {
        if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length)
        {
            compX[kk] = waveIn.wave[noteStarts[mm] + kk];
        }
        else
        {
            compX[kk] = Complex.Zero;
        }
    }

    Y = new Complex[nearest];

    Y = fft(compX, nearest);

    absY = new double[nearest];

    double maximum = 0;
    int maxInd = 0;

    for (int jj = 0; jj < Y.Length; jj++)
    {
        absY[jj] = Y[jj].Magnitude;
        if (absY[jj] > maximum)
        {
            maximum = absY[jj];
            maxInd = jj;
        }
    }

    for (int div = 6; div > 1; div--)
    {
        if (maxInd > nearest / 2)
        {
            if (absY[(int)Math.Floor((double)(nearest - maxInd) / div)] / absY[(maxInd)] > 0.10)
            {
                maxInd = (nearest - maxInd) / div;
            }
        }
        else
        {
            if (absY[(int)Math.Floor((double)maxInd / div)] / absY[(maxInd)] > 0.10)
            {
                maxInd = maxInd / div;
            }
        }
    }

    if (maxInd > nearest / 2)
    {
        pitches.Add((nearest - maxInd) * waveIn.SampleRate / nearest);
    }
    else
    {
        pitches.Add(maxInd * waveIn.SampleRate / nearest);
    }
}
```

Figure 3.6: onsetDetection for loop calculations

Approach:

I was able to first use a loop interchange on the HFC loop to improve spatial locality since the inner loop was iterated over less than the outer. Though not explicit this made parallisation of the whole block more efficient as I proceeded to also parallelise the now outer *ii* loop to achieve great performance. Figures 3.5 show the two blocks of code before and after.

My first approach before parallisation of the larger loop was to split it into 3 individual loops to potentially expose some parallisation. To respect the dependencies of the first loop I needed a global array of arrays of *twiddles* values to be used by the second loop and the same for the second loop which created an array of arrays of *Y* values to be used later. I also needed to change the fft function to take in a twiddles array to be used instead of storing this has a locally assigned array. The computation if the nearest value required no significant time so I defined one in each loop as it had no impact.

With each of the three loops working as intended I could attempt parallisation. The first two loops worked the same way as the *timefreq* constructor and the *stft* function. For first loop I could apply the same approach I did in the *timefreq* function, which was to calculate the *twiddles* array in parallel however, I did need to assign those arrays into a global *twiddles* variable to be used in the second loop. The second loop, like the first, used a parallel for loop which just needed to assign the end array to an array of arrays. The last loop did not need to parallelised as it was not computationally heavy, it was left in its own loop. The code can be seen in Figure 3.7 for the full implementation of this loop.

```
HFC = new float[stftRep.timeFreqData[0].Length];

for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
{
    for (int ii = 0; ii < stftRep.wSamp / 2; ii++)
    {
        HFC[jj] = HFC[jj] + (float)Math.Pow((double)stftRep.timeFreqData[ii][jj] * ii, 2);
    }

}
```

```
HFC = new float[stftRep.timeFreqData[0].Length];

Parallel.For(0, stftRep.wSamp / 2, new ParallelOptions
{ MaxDegreeOfParallelism = numThreads }, ii =>
    {
        for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
        {
            HFC[jj] = HFC[jj] + (float)Math.Pow((double)stftRep.timeFreqData[ii][jj] * ii, 2);
        }
    });
```

Figure 3.5: HFC loop transformation and parallisation

```
Complex[][] yArrays = new Complex[lengths.Count][];
Complex[][] twidArrays = new Complex[lengths.Count][];
Complex[] twid;

for (int mm = 0; mm < lengths.Count; mm++)
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    twid = new Complex[nearest];

    Parallel.For(0, nearest, new ParallelOptions
    { MaxDegreeOfParallelism = numThreads }, ll =>
        {
            double a = 2 * pi * ll / nearest;
            twid[ll] = Complex.Pow(Complex.Exp(-i), (float)a);
        });

    twidArrays[mm] = twid;
}

Parallel.For(0, lengths.Count, new ParallelOptions
{ MaxDegreeOfParallelism = numThreads }, mm =>
    {
        int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
        Complex[] compX = new Complex[nearest];

        for (int kk = 0; kk < nearest; kk++)
        {
            if (kk < lengths[mm] && (noteStarts[mm] + kk) < waveIn.wave.Length)
            {
                compX[kk] = waveIn.wave[noteStarts[mm] + kk];
            }
            else
            {
                compX[kk] = Complex.Zero;
            }
        }

        yArrays[mm] = FFT.IterativeFFT(compX, nearest, twidArrays[mm]);
    });
```

```
for (int mm = 0; mm < lengths.Count; mm++)
{
    int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
    absY = new double[nearest];

    double maximum = 0;
    int maxInd = 0;

    for (int jj = 0; jj < yArrays[mm].Length; jj++)
    {
        absY[jj] = yArrays[mm][jj].Magnitude;
        if (absY[jj] > maximum)
        {
            maximum = absY[jj];
            maxInd = jj;
        }
    }

    for (int div = 6; div > 1; div--)
    {
        if (maxInd > nearest / 2)
        {
            if (absY[(int)Math.Floor((double)(nearest - maxInd) / div)] / absY[(maxInd)] > 0.10)
            {
                maxInd = (nearest - maxInd) / div;
            }
        }
        else
        {
            if (absY[(int)Math.Floor((double)maxInd / div)] / absY[(maxInd)] > 0.10)
            {
                maxInd = maxInd / div;
            }
        }
    }

    if (maxInd > nearest / 2)
    {
        pitches.Add((nearest - maxInd) * waveIn.SampleRate / nearest);
    }
    else
    {
        pitches.Add(maxInd * waveIn.SampleRate / nearest);
    }
}
```

Figure 3.7: onsetDetection loop breakdown implementation

4.0 Results

I used the stopwatch class to time different key events in the program, mainly the main program as a whole (from loading the file to the display window) which I used for my speed up graph and main comparison to the best sequential version. Additionally I also measured the execution time for the *onsetDetection* function and the *stft* functions of both programs as well.

The recorded times for the sequential version is shown below:

Best Sequential	
Task	**Time(ms)**
Overall Runtime	4370
STFT	1300
OnsetDet	1600

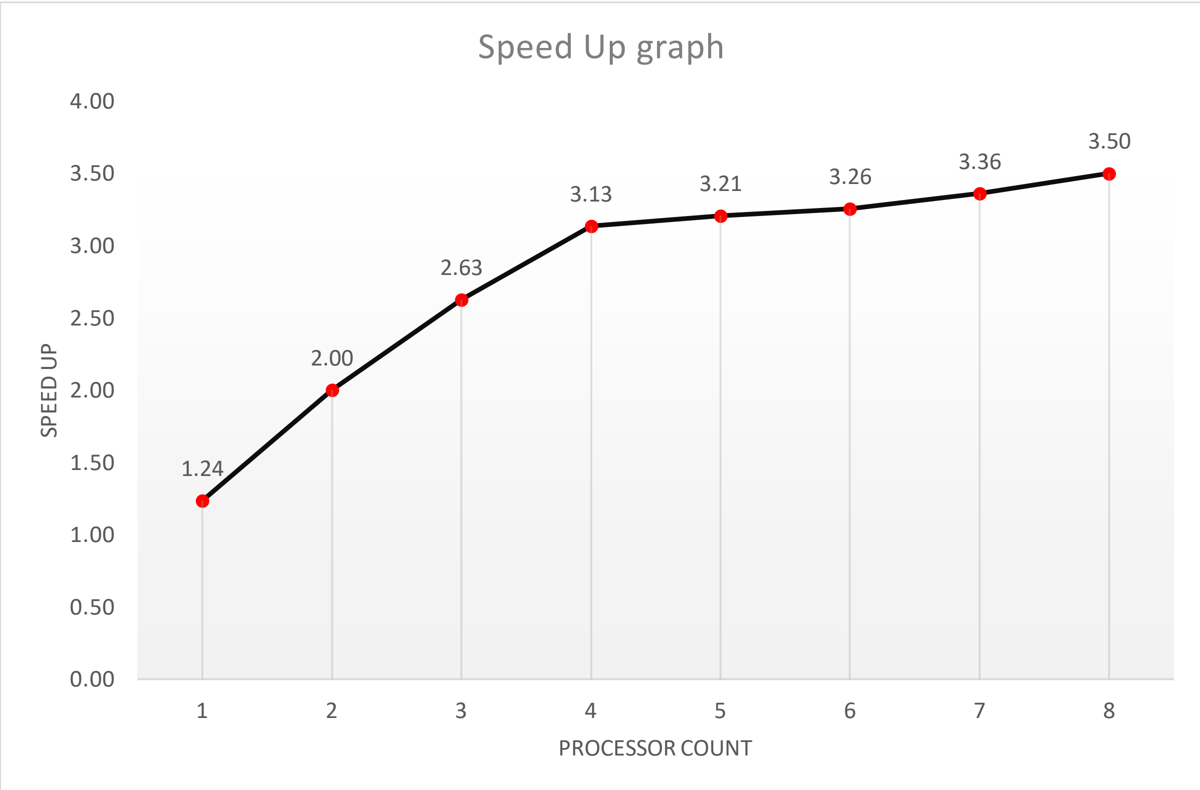
The following tables show the timed results of the parallelised version of the program for each of the 3 criteria and various number of processors from 1 to 8:

Overall Runtime	
Num Processors	**Execution time (ms)**
1	3400
2	2100
3	1600
4	1340
5	1310
6	1290
7	1250
8	1200

STFT	
Num Processors	**Execution time (ms)**
1	1430
2	940
3	670
4	520
5	460
6	475
7	480
8	380

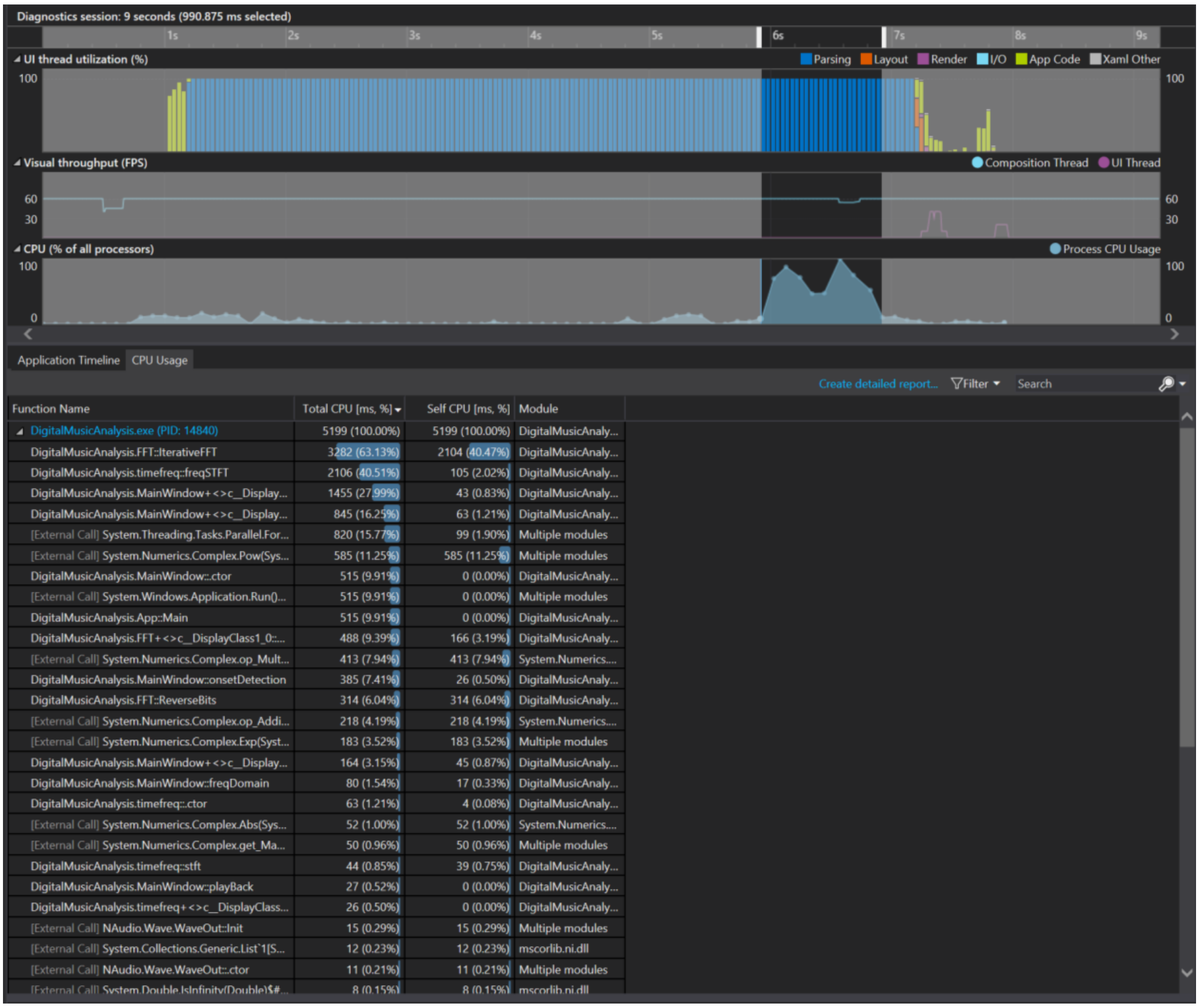
OnsetDet	
Num Processors	**Execution time (ms)**
1	1800
2	960
3	740
4	640
5	630
6	600
7	530
8	500

As shown I the above tables, it is evident that not only does the parallelised program run faster on a single processor but is also approximately 3.5 times better in performance then its best sequential version. The following speed up graph shows the various speed up with increased processors:

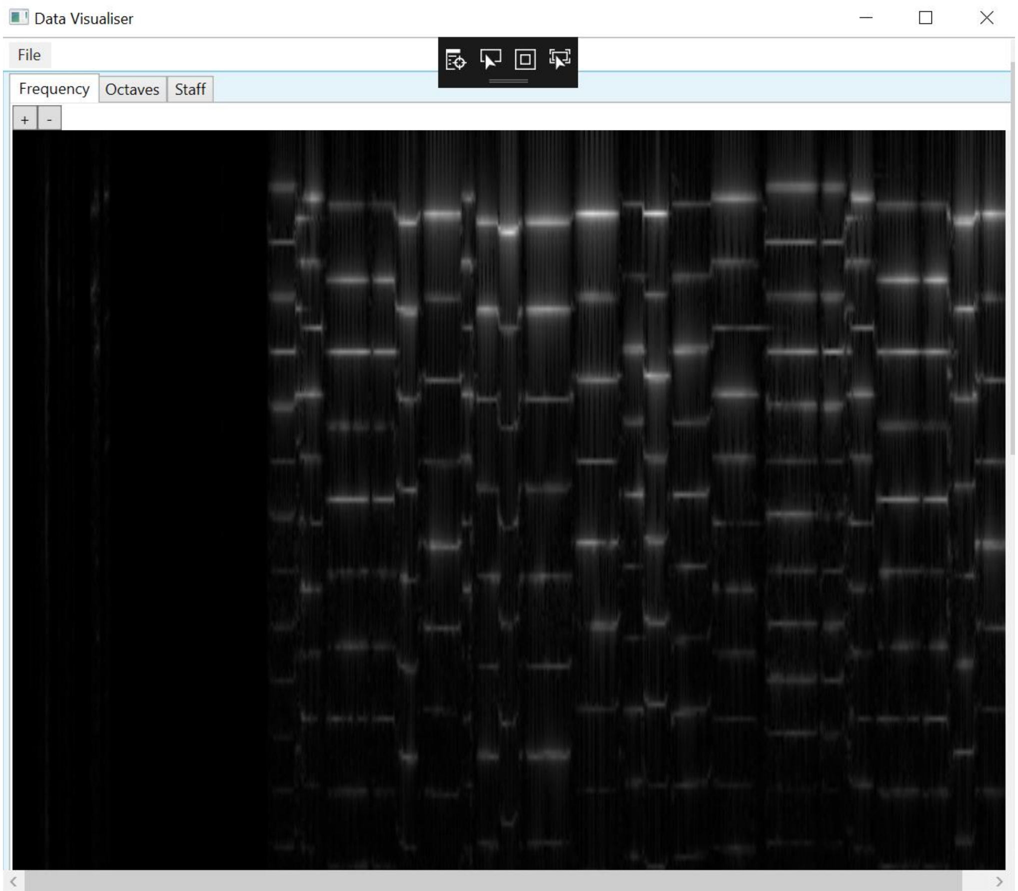


I can be seen in the graph that the speed up is very sub linear such that it there is no real significant speed up after a certain number of processors (in this case 8 processors).

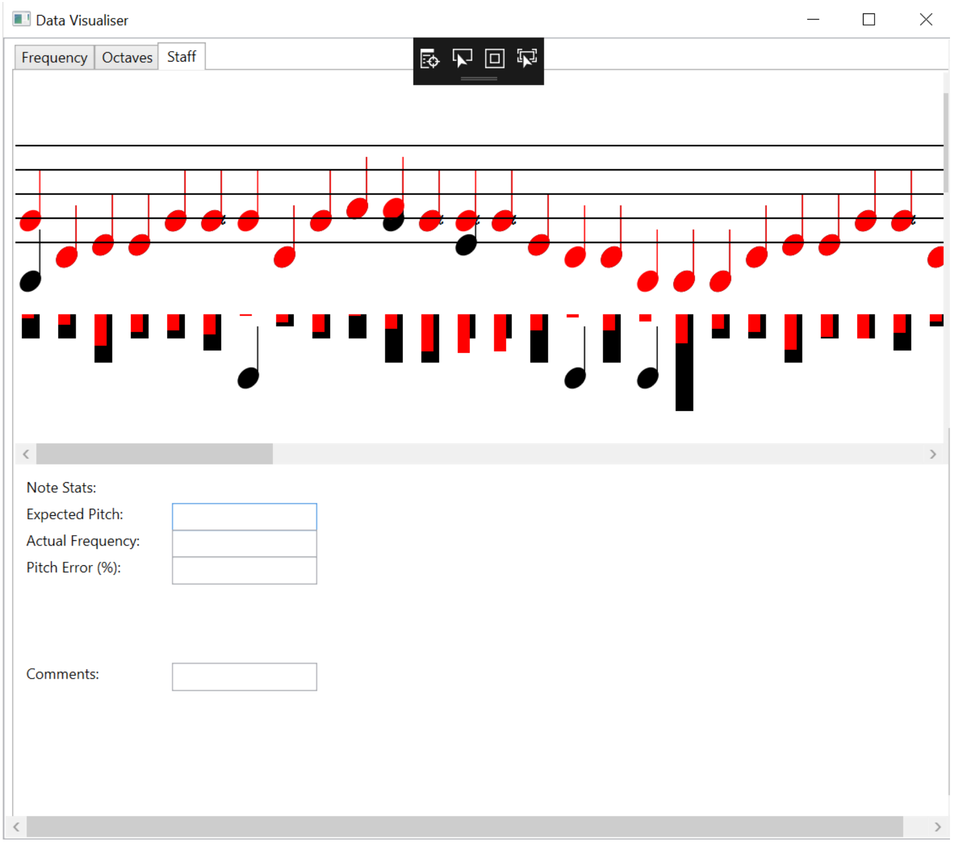
The profiled results shown below show that now most of the computation is spent on the single fft method called by the onsetDetection and stft functions. This proves to be useful now that these functions have no significant impact on the program.

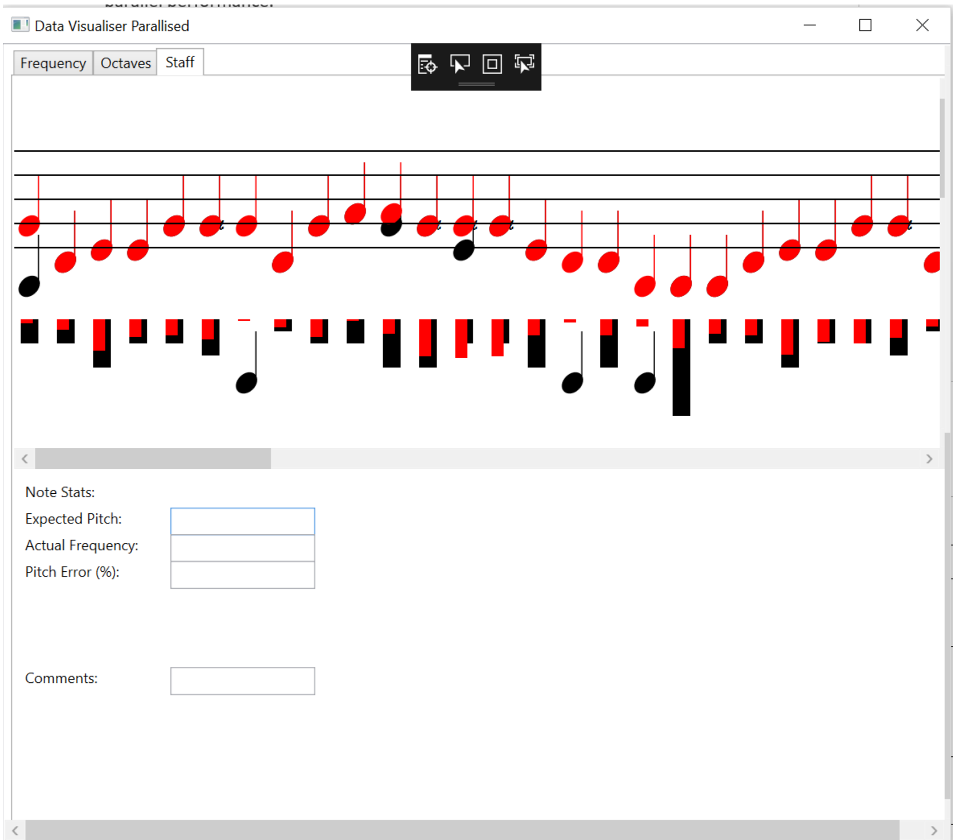


The results for the program after parallelising also prove to be consistent to its sequential version which was to be expected:


Data Visualiser
File
Frequency
Octaves
Staff


Data Visualiser Parallised
File
Frequency
Octaves
Staff


Data Visualiser
Frequency
Octaves
Staff
Note Stats:
Expected Pitch:
Actual Frequency:
Pitch Error (%):
Comments:


Data Visualiser Parallised
Frequency
Octaves
Staff
Note Stats:
Expected Pitch:
Actual Frequency:
Pitch Error (%):
Comments:

5.0 Conclusion and Reflection

Overall I achieved a really large performance boost, which was more than expected. I definitely learned a lot from this assignment when it came to variable dependence, notably in the onsetDetection function. It made me having to consider what I am parallelising and not just throw a parallel for loop and call it parallelised. However, my approach to improving the stft function was unfortunately this function was not as scalable as intended as with I higher thread count, there will be more threads assigned to loop over which would impact the performance after 8 processors.

Having never used Visual Studios profiler I feel like this tool may prove useful to me in future projects where I need to determine where programs spend most of their time running certain tasks.

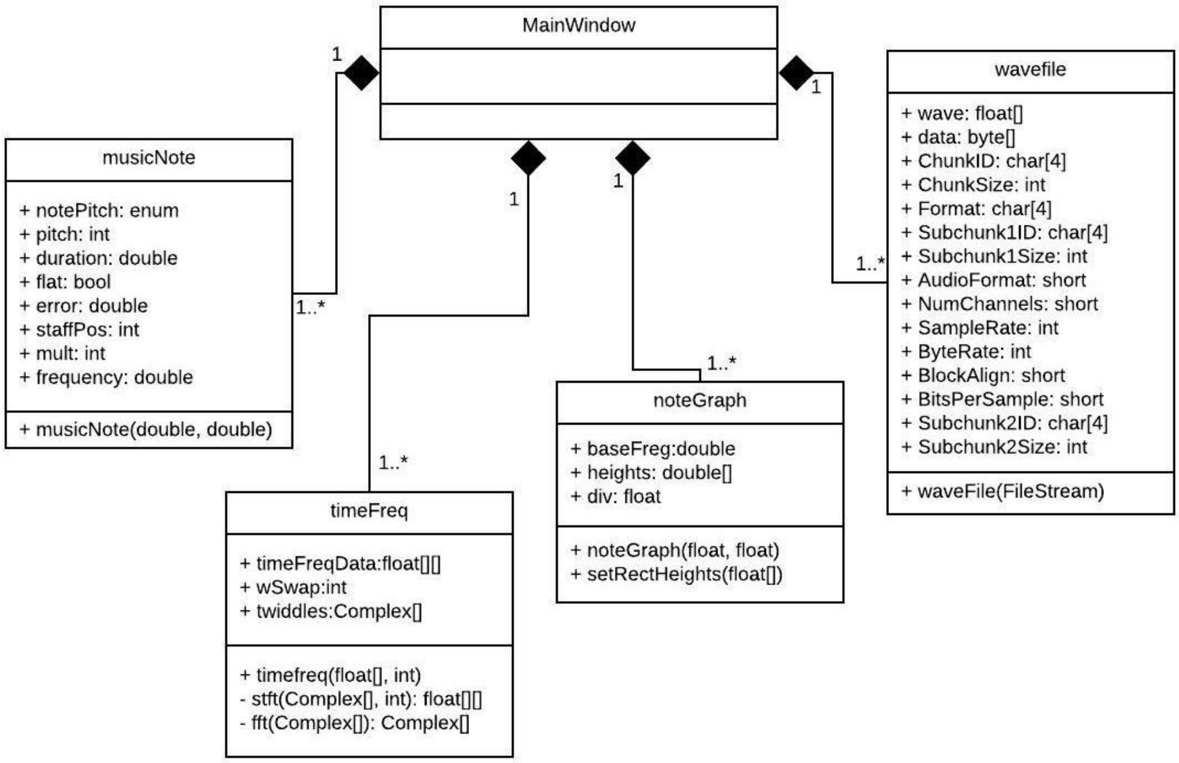
The next step would be too to further parallelise or make use of a more efficient FFT algorithm which would definitely be something interesting to look at in the future.

A link to a GitHub repository is given in **Appendix D**

6.0 Appendices

6.1 Appendix A

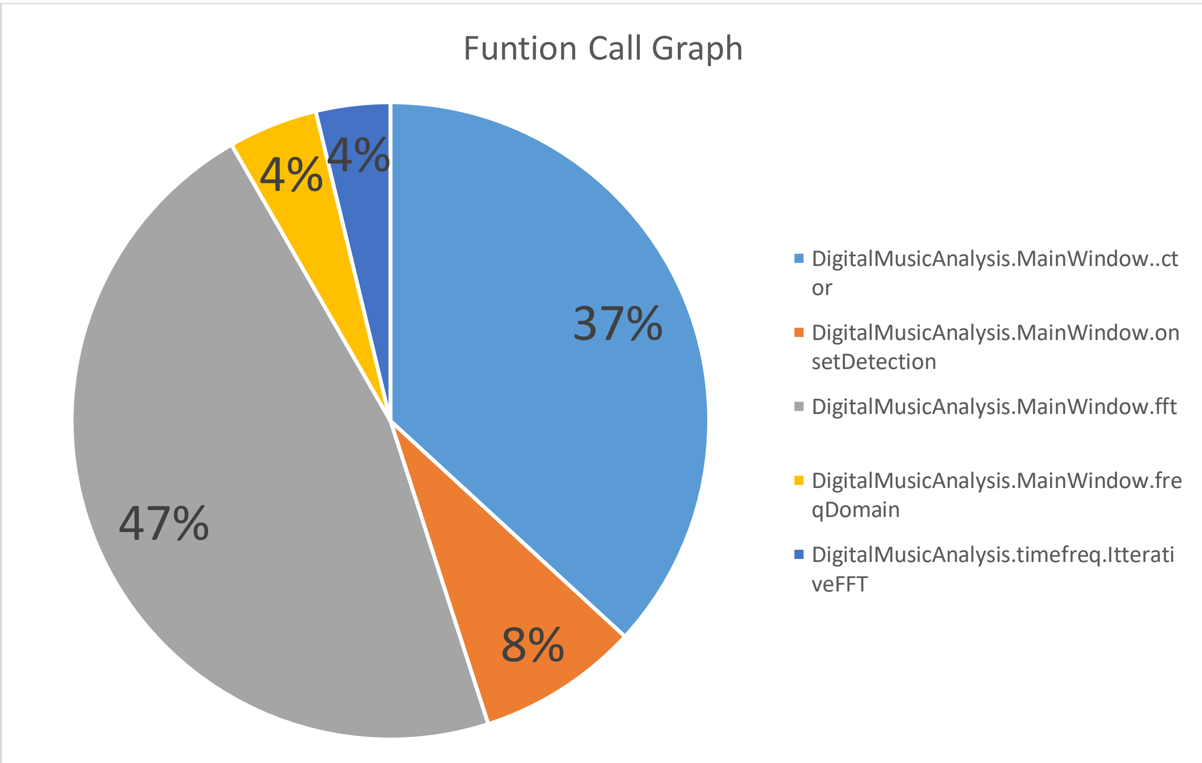
Digital Music Analysis



Digital Music Analysis UML Class Diagram

6.2 Appendix B

Level	Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %	Module Name
5	DigitalMusicAnalysis.App.Main	14,098	0	72.02	0	DigitalMusicAnalysis.exe
13	DigitalMusicAnalysis.MainWindow..ctor	13,110	0	66.97	0	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.openFile	8,264	0	42.22	0	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.onsetDetection	2,918	146	14.91	0.75	DigitalMusicAnalysis.exe
15	DigitalMusicAnalysis.MainWindow.fft	1,972	96	10.07	0.49	DigitalMusicAnalysis.exe
16	DigitalMusicAnalysis.MainWindow.fft	1,821	70	9.3	0.36	DigitalMusicAnalysis.exe
17	DigitalMusicAnalysis.MainWindow.fft	1,717	59	8.77	0.3	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.freqDomain	1,617	18	8.26	0.09	DigitalMusicAnalysis.exe
18	DigitalMusicAnalysis.MainWindow.fft	1,610	75	8.22	0.38	DigitalMusicAnalysis.exe
15	DigitalMusicAnalysis.timefreq..ctor	1,594	25	8.14	0.13	DigitalMusicAnalysis.exe
16	DigitalMusicAnalysis.timefreq.stft	1,552	145	7.93	0.74	DigitalMusicAnalysis.exe
19	DigitalMusicAnalysis.MainWindow.fft	1,478	83	7.55	0.42	DigitalMusicAnalysis.exe
20	DigitalMusicAnalysis.MainWindow.fft	1,354	90	6.92	0.46	DigitalMusicAnalysis.exe
17	DigitalMusicAnalysis.timefreq.ItterativeFFT	1,342	859	6.86	4.39	DigitalMusicAnalysis.exe
21	DigitalMusicAnalysis.MainWindow.fft	1,212	89	6.19	0.45	DigitalMusicAnalysis.exe
22	DigitalMusicAnalysis.MainWindow.fft	1,076	86	5.5	0.44	DigitalMusicAnalysis.exe
23	DigitalMusicAnalysis.MainWindow.fft	962	73	4.91	0.37	DigitalMusicAnalysis.exe
24	DigitalMusicAnalysis.MainWindow.fft	846	92	4.32	0.47	DigitalMusicAnalysis.exe
25	DigitalMusicAnalysis.MainWindow.fft	733	90	3.74	0.46	DigitalMusicAnalysis.exe
26	DigitalMusicAnalysis.MainWindow.fft	604	96	3.09	0.49	DigitalMusicAnalysis.exe
27	DigitalMusicAnalysis.MainWindow.fft	479	93	2.45	0.48	DigitalMusicAnalysis.exe
28	DigitalMusicAnalysis.MainWindow.fft	355	87	1.81	0.44	DigitalMusicAnalysis.exe
29	DigitalMusicAnalysis.MainWindow.fft	240	89	1.23	0.45	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.playBack	149	0	0.76	0	DigitalMusicAnalysis.exe
18	DigitalMusicAnalysis.timefreq.ReverseBits	127	127	0.65	0.65	DigitalMusicAnalysis.exe
30	DigitalMusicAnalysis.MainWindow.fft	116	64	0.59	0.33	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.InitializeComponent	86	0	0.44	0	DigitalMusicAnalysis.exe
13	DigitalMusicAnalysis.MainWindow.updateHistogram	38	0	0.19	0	DigitalMusicAnalysis.exe
4	DigitalMusicAnalysis.MainWindow.updateSlider	34	0	0.17	0	DigitalMusicAnalysis.exe
31	DigitalMusicAnalysis.MainWindow.fft	33	23	0.17	0.12	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.loadWave	31	0	0.16	0	DigitalMusicAnalysis.exe
6	DigitalMusicAnalysis.App..ctor	28	0	0.14	0	DigitalMusicAnalysis.exe
13	DigitalMusicAnalysis.MainWindow.<updateSlider>b__26_0	27	0	0.14	0	DigitalMusicAnalysis.exe
15	DigitalMusicAnalysis.wavefile..ctor	24	15	0.12	0.08	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.loadHistogram	22	1	0.11	0.01	DigitalMusicAnalysis.exe
32	DigitalMusicAnalysis.MainWindow.fft	6	5	0.03	0.03	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.readXML	4	0	0.02	0	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.loadImage	4	0	0.02	0	DigitalMusicAnalysis.exe
14	DigitalMusicAnalysis.MainWindow.loadHistogram	2	0	0.01	0	DigitalMusicAnalysis.exe
6	DigitalMusicAnalysis.MainWindow..ctor	2	0	0.01	0	DigitalMusicAnalysis.exe
7	DigitalMusicAnalysis.MainWindow.openFile	2	0	0.01	0	DigitalMusicAnalysis.exe
3	DigitalMusicAnalysis.MainWindow..ctor	2	0	0.01	0	DigitalMusicAnalysis.exe
4	DigitalMusicAnalysis.MainWindow.openFile	2	0	0.01	0	DigitalMusicAnalysis.exe
0	DigitalMusicAnalysis.App.Main	2	0	0.01	0	DigitalMusicAnalysis.exe
8	DigitalMusicAnalysis.MainWindow..ctor	2	0	0.01	0	DigitalMusicAnalysis.exe
9	DigitalMusicAnalysis.MainWindow.openFile	2	0	0.01	0	DigitalMusicAnalysis.exe
0	DigitalMusicAnalysis.MainWindow..ctor	2	0	0.01	0	DigitalMusicAnalysis.exe
1	DigitalMusicAnalysis.MainWindow.openFile	2	0	0.01	0	DigitalMusicAnalysis.exe
5	DigitalMusicAnalysis.MainWindow..ctor	1	0	0.01	0	DigitalMusicAnalysis.exe
6	DigitalMusicAnalysis.MainWindow.openFile	1	0	0.01	0	DigitalMusicAnalysis.exe



Function Call Data and Graph

6.3 Appendix C

```
algorithm iterative-fft is
   input: Array a of n complex values where n is a power of 2
   output: Array A the DFT of a

   bit-reverse-copy(a,A)
   n ← a.length
   for s = 1 to log(n)
       m ← 2^s
       ω_m ← exp(−2πi/m)
       for k = 0 to n-1 by m
           ω ← 1
           for j = 0 to m/2 – 1
               t ← ω A[k + j + m/2]
               u ← A[k + j]
               A[k + j] ← u + t
               A[k + j + m/2] ← u – t
               ω ← ω ω_m

   return A
algorithm bit-reverse-copy(a,A) is
  input: Array a of n complex values where n is a power of 2,
  output: Array A of size n

  n ← a.length
  for k = 0 to n – 1
      A[rev(k)] = a[k]
```

Cooley-Turkey FFT algorithm pseudocode

6.4 Appendix D

https://github.com/Bloedaeth/CAB401-DigitizedMusic

7.0 References

Liu, B. (n.d.). Parallel Fast Fourier Transform. Retrieved From https://cs.wmich.edu/gupta/teaching/cs5260/5260Sp15web/studentProjects/tiba&hussein/03278999.pdf

Cooley-Tukey Algorithm. Retrieved from https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm