

CAB401 Assignment 1

DIGITAL MUSIC ANALYSIS PARALLELIZATION

PATRIC MARCHANT – N9720316

Contents

The Application	2
Class diagram	2
Potential for parallelism.....	3
Framework and Hardware	4
The framework	4
The system.....	4
Parallelization attempts	5
freqDomain	5
onsetDetection	6
Comparing sequential and parallel	7
Compilers, software tools, and techniques.....	8
Software	8
Tools	8
Performance Profiler	8
Microsoft Excel	8
Techniques.....	9
Barriers	9
Reflection.....	9
Appendix.....	10
Appendix A.....	10
Appendix B.....	10
Appendix C.....	11
Appendix D.....	12
Appendix E.....	12
Appendix F	13
Appendix G	14
Appendix H	14
Appendix I.....	14
Appendix J.....	15
Appendix K.....	16
Appendix L	16
Appendix M.....	17
References	18

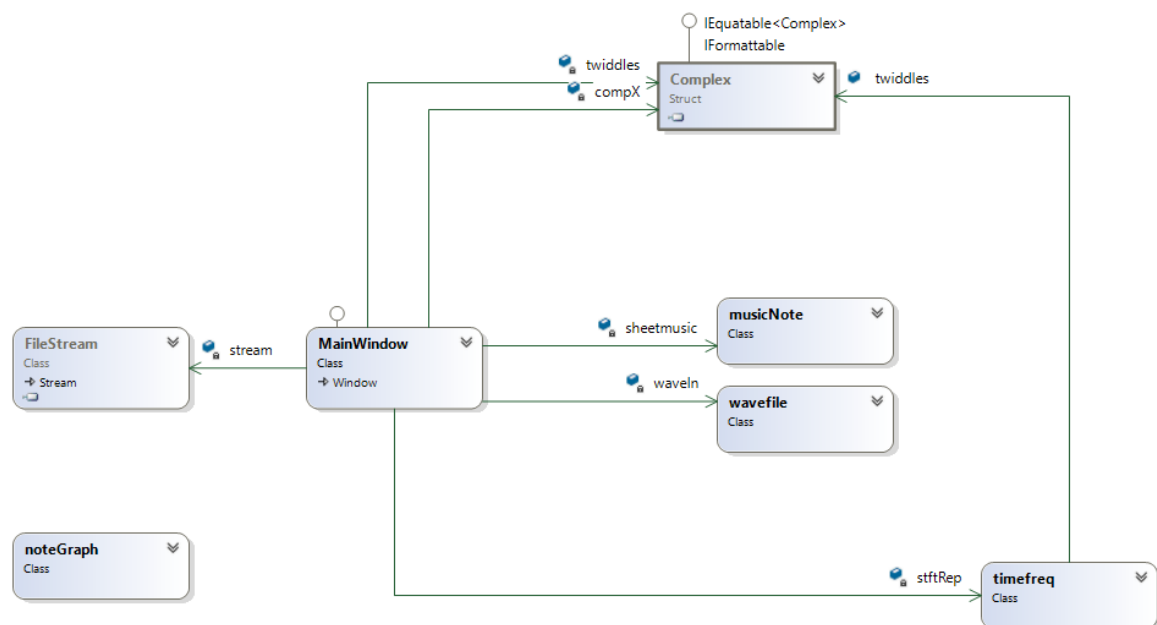
The Application

The application that we are aiming to parallelise is called “Digital Music Analysis” and is a C# application presented using the “Window” class, which aims to assist violin players to show how close they were to the perfect score of the song. This is done by the file prompting the user to input a WAV file of the recording of their performance (appendix A), and an XML file for the song’s score (appendix B), the recording WAV file is then played over the score XML file to perform the analysis. For testing and demonstration purposes, we be using Jupiter.wav, and jupiter.xml, which are included in the folder, as input files.

From here we are presented with a “Data visualiser” window containing three tabs, “Frequency” (appendix C), “Octaves” (appendix D), and “Staff” (appendix E). The “frequency” tab shows the frequency of the selected file over the course of the song and “octaves” shows the frequencies hit for each octave over the course of the song, which can be stopped, and scrubbed to any part of the performance using the buttons and slider on the form.

Finally, we have the “staff” tab, which shows red and black notes over the song, with black notes being notes which were played correctly (on pitch), and red notes being played incorrectly (off pitch), red notes can also be hovered over to display the expected pitch, actual frequency, pitch error (%), and even comments on whether the note was too flat or too sharp. Black and red boxes underneath the notes also describe if the note was played for the correct duration according to the song.

Class diagram



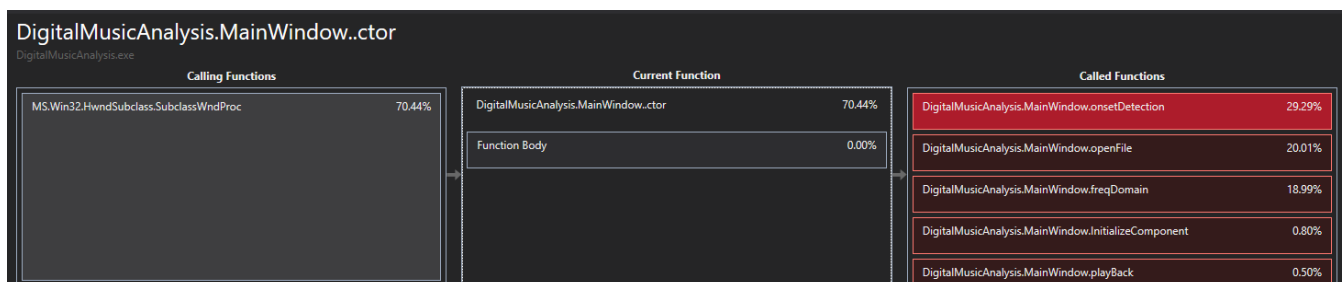
In this C# application, the main function of the program is contained within “mainWindow.xaml.cs” and uses four other user created classes: “musicNote.cs”, “timefreq.cs”, “wavefile.cs”, and “noteGraph.cs”, which will be covered as they are used. The application first initialises the window, then prompts the user to select their “wav” and “xml” file path through the “openFile” method using the “FileStream

" class, starts a thread to handle updating the slider in the "octaves" tab of the Data visualiser, and uses the wav file path as an argument for the "loadWave" function which loads the file for the "freqDomain" function.

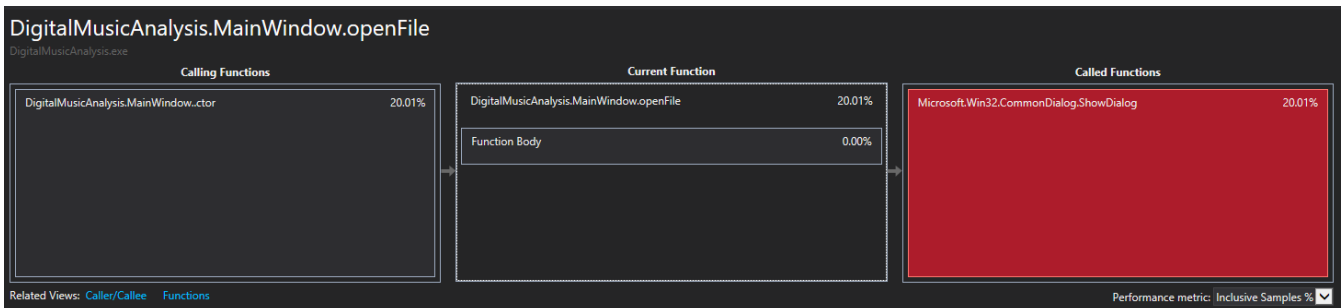
"freqDomain" is then called from the main method, which creates an instance of the "timefreq.cs" class, which handles the coded algorithms for converting a "wav" file data into a two-dimensional time and frequency array, after this, the sheet music "xml" file is loaded using the "readXML" function. The "onsetDetection" method is then called from main, which plots when each note starts and finishes, and also presents statistics about how far the frequency was off in the staff tab of the Data visualiser. From here, the time-frequency image and histogram bars for the frequency (appendix C) and staff tabs (appendix E) are loaded in respectively calling the "loadImage" and "loadHistogram" methods, and the "wav" file is played, allowing the user to see which note in which octave is being hit at different points of the song.

Potential for parallelism

Before looking into parts of the code which I could exploit for parallelism, I first turned to the performance profiler built into visual studio (my IDE) which can be used for us to analyse how the application runs, which methods the application spends most time in, and CPU usage for each, letting us avoid having to manually time every line in the code.



After analysing the code (appendix F-H), we can observe the application spends most of its time in the onsetDetection, openFile, and freqDomain methods called from the main thread, which are all running in sequential. As said before, we know that freqDomain and onsetDetection both harbour important algorithms for parsing the "wav" file and checking when notes begin and end respectively, so we will come back to these. The openFile method on the other hand, only has 3 lines of code (excluding the return statement), and when viewing the performance profiler (below), we can see all the time/CPU usage is spent presenting dialog boxes to the user, which cannot really be avoided, and therefore we will not focus our energy on this.



If we breakdown the work that the `freqDomain` method is doing, we can see it first initialises a “timefreq” object into a variable, and calls the constructor by passing in our “wav” file, and the integer 2048, which represents the amount of samples for the Fourier transform (translates to about $1/24^{\text{th}}$ of a second). When then viewing `timefreq`’s constructor, we can see that it initialises some local variables, and loops through the passed “wav” file (which in reality is just an array of floats) 2048 times to fill the “twiddles” array of complex numbers. When viewing the performance profiler for this method, we can observe that the majority of the time spent/CPU usage is from the “fft” method inside the `timefreq` class. “fft” in this case, is a recursive, divide and conquer algorithm which is called through the “stft” method which is called in `timefreq`’s constructor, and is where the coded algorithm for the Fourier transform of the “wav” file is actually taking place. Because divide and conquer algorithms are quite fast, I decided to instead attempt to parallelise the ‘for’ loop that calls the “fft” method.

Next, we will shift our attention to the `onsetDetection` method, which as we recall, determines the start and finish times of a note, as well as the frequency over each duration. When consulting the profiler, it shows that it spends most of its time in a different “fft” method (this time from “MainWindow.xaml” instead of “timefreq”) which is slightly different in that on top of the array of complex numbers for the argument, the method also requires an integer, meaning we should probably refrain from combining the separate “fft” functions.

Framework and Hardware

The framework

Because C# lies in the .NET framework, we will use the “System.Threading”, and “System.Threading.Tasks” namespaces, which provide multiple classes and interfaces which allow multithreaded code through a plethora of different options.

On top of this, we also use the “Task parallel library” (TPL), which is a set of public types and APIs in the “System.Threading” and “System.Threading.Tasks” namespaces, and works to simplify the process of adding parallelism and concurrency to applications. TPL also contains a method called “parallel.for”, which is used to split the workload of the “for” loop over every available processor.

The system

Processor: i7-6700HQ

Code Name: Skylake

of Cores: 4

of Logical cores: 8

Socket: FCBGA1440

Clock Speed: 2.60 GHz (max of 3.50 GHz)

Hyper-Threading: Yes

Memory: 16GB DDR4

Cache: 6 MB SmartCache

Parallelization attempts

freqDomain

1. For loop in timefreq.cs constructor (successful)

The first and easiest parallelisation in our program was the “for” loop in the timefreq class’s constructor, which iterated through the amount of samples and filled the “twiddles” (complex) array, this doesn’t seem to contain any data dependencies, and therefore looks like a good starting point to attempt using a “Parallel.For” in the original “for” loops place (appendix J).

This proved successful, speeding up the program by an average of approximately 478 ms!

2. stft method in timefreq.cs class (unsuccessful)

Because the most CPU usage was spent calling the “fft” method from a number of “for” loops in the “stft” method, I decided to attempt to parallelise the major two for loops in the method, which are used to essentially call the fft method and parse the result’s type before storing inside the “Y” 2d array of floats, which contains frequency/time data and is returned back to the constructor once finished.

This however, proved more difficult than expected, as my first attempts proved unsuccessful as I soon ran into issues with race conditions from the “tempFFT” and “fftMax” variables being global, and attempted implementing a lock, but was unsuccessful.

3. For loops in freqDomain method (unsuccessful)

After failing to run the parallel “for” loops in the “stft” method I decided to attempt to try parallelising the “for” loops which piece together the “pixelArray” array in the “freqDomain” method in “MainWindow.xaml.cs”, however after implementing and testing speeds of Parallel.for for both the inner and outer loops, I realised there was an average speed down of approximately 150 ms (appendix K) and it was most likely more work handling threads than work to do. Hence these Parallel.For loops were removed from the final algorithm

4. Creating threads in “stft” which call method to call “fft” (successful)

Another idea to parallelise the “fft” call was to create a method to handle calling the “fft” method (stftCallFft), which will be called inside the “stft” method. Because of this, instead of using “Parallel.For”, we need to create an array of threads and have them each run the “stftCallFft” method. “stftCallFft” also will need to calculate “chunks” for each thread to iterate over to stop overlap and increase productivity, which will be based on the thread id and how many samples are required.

After implementing this, I found we also needed to globalise the “Y” array and a few other variables for the “stftCallFft” method to access them. I also found we could also parallelise the first “for” loop which fills the “Y” array in the “stft” method using “Parallel.For” as there were no data dependencies to be accounted for. Doing both (appendix L) resulted in a sizeable speedup of an average of almost 400 ms compared to our next best version.

onsetDetection

1. For loop calling “fft” in onsetDetection method (unsuccessful)

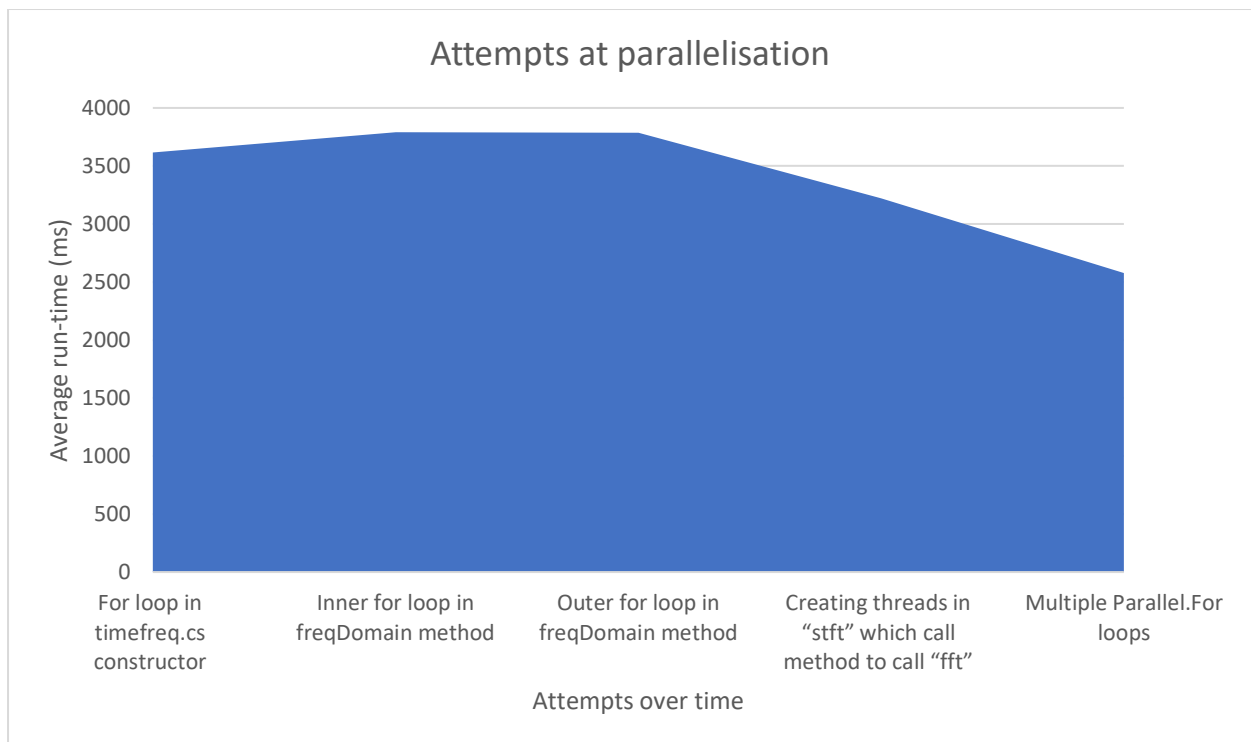
Immediately when looking at the onsetDetection method, I knew we would most likely have to follow the same process as we did in the timefreq class to parallelise it, as the method uses multiple “for” loops, with one of which handling the call to “fft” (which as we know is using the most CPU).

This failed however, as there seems to be an underlying data dependencies when attempting to add to the “pitches” list, which makes things tricky, therefore this was scrapped.

2. Multiple Parallel.For loops (successful)

After struggling with ideas to alleviate the data dependencies for the “for” loop calling “fft”, I decided to instead use some “Parallel.For” loops to optimize some of the other “for” loops in the onsetDetection method (appendix M), which ended up being successful and lowering the run time by an average of 642 ms.

Figure showing run-time of algorithm over implemented (didn’t throw an exception) parallelisation attempts



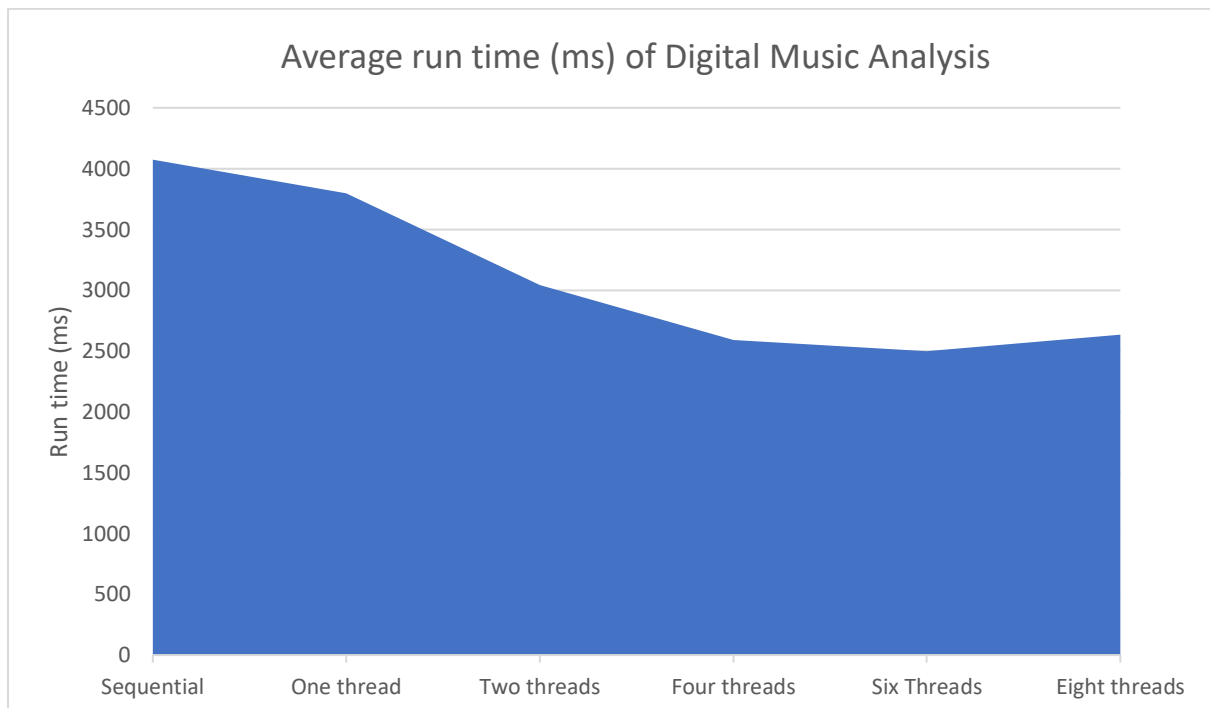
Comparing sequential and parallel

To get an accurate representation of the run time of the program I decided to use Stopwatch class from the "System.Diagnostics" namespace to time the application, and the "FileStream" and "StreamWriter" classes to print the output to a file instead of printing to the console using "System.Console.Out", which can prove more intensive depending on the application.

Run time (ms) of Digital Music Analysis						
	Sequential	One thread	Two threads	Four threads	Six Threads	Eight threads
Test 1	3956	3815	3001	2574	2479	2608
Test 2	4121	3767	2951	2590	2525	2680
Test 3	4072	3771	3206	2629	2508	2638
Test 4	4150	3836	2949	2561	2472	2597
Test 5	4068	3802	3104	2598	2499	2641
<u>Average</u>	<u>4073</u>	<u>3798</u>	<u>3042</u>	<u>2590</u>	<u>2497</u>	<u>2633</u>

To properly test the parallelisation of the Digital Music Analysis application, I decided to do five tests for both versions of the application (sequential and parallel using native amount of threads), including testing 4 other different thread amounts (one, two, four, and six). After testing each version, I found using six threads was the most efficient, and resulted in an average speed up of 1.63, or a difference of 1576 ms compared to the sequential version.

From the above and below figures, we can observe that from the implementation of parallel processes and the use of threads, that there is a sub-linear speed up of the application as when increasing the amount of threads available to the program after 6, the application becomes slightly slower. Although six threads seems to be the optimal amount, the difference in run time between four, six, and eight threads is negligible, and seems to be caused not enough work for each thread, causing threads to wait on each other in certain situations, therefore, this implementation cannot be considered as a “perfect” or “super-perfect”/scalable parallelism speed up.



Compilers, software tools, and techniques

Software

The program is written in C#, and compiled/run using visual studio community 2017, which is an integrated development environment (IDE) made by Microsoft, running on Windows 10 Version 10.0.18362 Build 18362.

Tools

Performance Profiler


Visual Studio's inbuilt performance profiler was also used to get reports/detailed reports of the programs performance to analyse, detailing each method's call, how long it spent in the method etc.

Microsoft Excel

Microsoft Excel was also used to log and graph the programs performance data accordingly for qualitative comparisons.

Techniques

To parallelise the application, I followed the 11-step process given to us in class (below), repeating step 5-11 until I felt the application had been sufficiently parallelised

- 
1. Obtain representative and realistic data sets
 2. Time and profile sequential version
 3. View source and understand high-level structure
 4. Analyze dependencies
 5. Determine sections that could be parallelized
 6. Decide what parallelism might be worth exploiting
 7. Consider restructuring program or replacing algorithms to expose more parallelism
 8. Transform program into an explicitly parallel form.
 9. Test and Debug parallel version
 10. Time and profile parallel version
 11. Determine issues inhibiting greater performance

Barriers

When beginning the assignment, the first barrier I had to overcome was understanding the underlying science behind sound, and how octaves, frequency, and Fourier transforms work, as well as how the algorithms translated to C# and were presented to the screen (been a while since using c# window class). This proved quite difficult initially, as the program had very few comments, and was confusing until watching the video provided by the unit coordinator detailing the application.

The rest of the logbook about how I refactored the code to be parallel and issues I encountered can be found in the above "Parallelisation attempts" section.

Reflection

Before beginning this unit (and largely this report), I had only an understanding of what parallel programming and threading was, and the basic idea that threads can be used to do units of work in parallel to each other. This unit gave me a much better understanding of parallelism, how it can be implemented (safely) and the multitude of libraries and frameworks that can be used to do so.

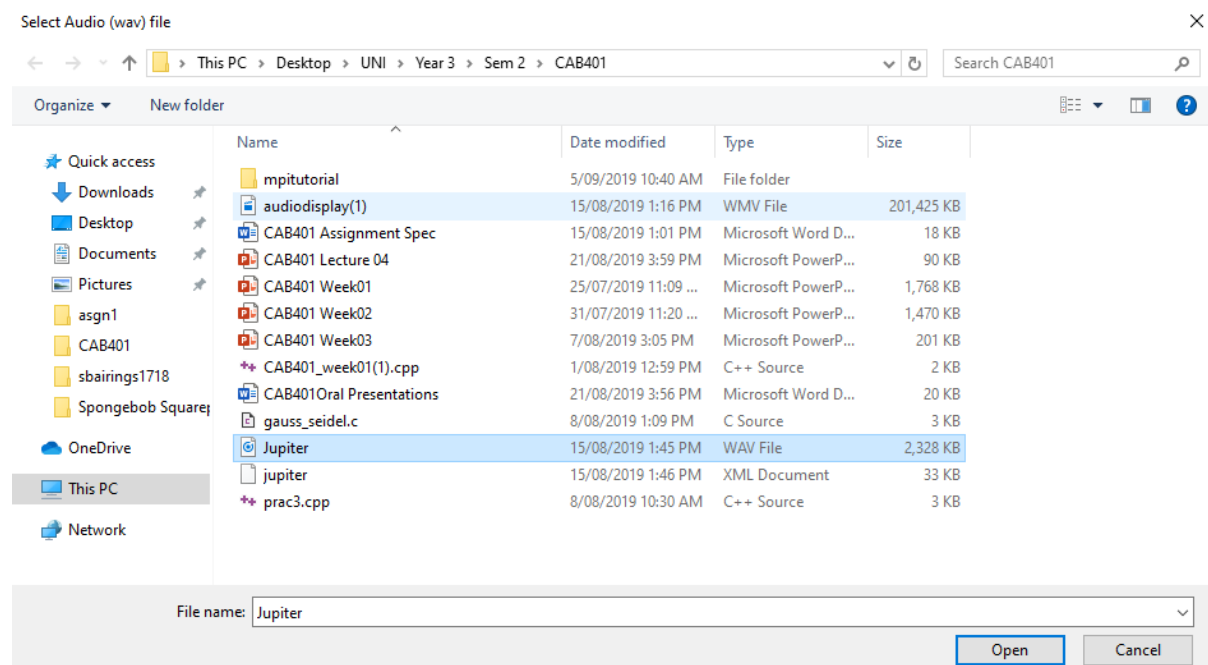
In the end, my attempt at parallelising the Digital Music Analysis application was semi-successful, yielding a sub-linear speedup of 4073 ms in the sequential version, to 2497 in the parallel version using six threads, almost halving the applications run time, which we could predict would provide a meaningful difference to the user. If I had to do this task again, I would undoubtedly begin earlier, as

I had to rush to parallelise the program to a meaningful change, and didn't get to explore all the different options and ideas for parallelism when analysing the application, because my knowledge of the math/algorithm behind the fourier transform and other algorithms is basic I stayed away from looking inside the coded algorithms for parallelisation options, which could also benefit the optimization of the application.

Appendix

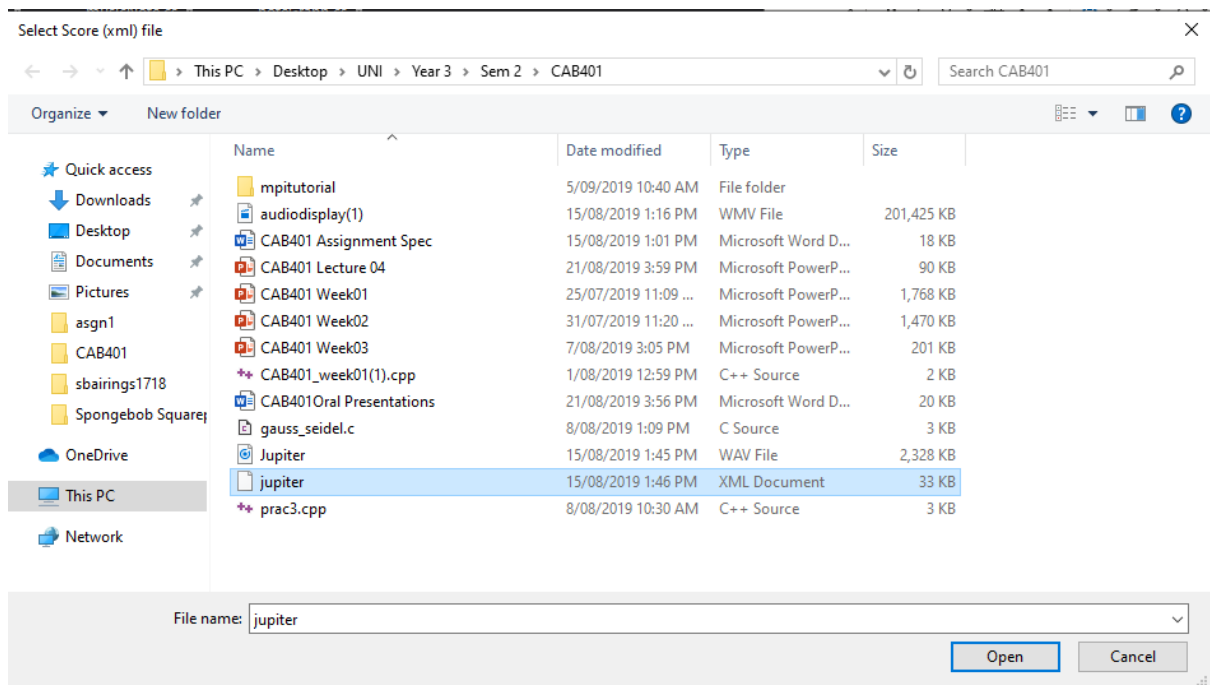
Appendix A

Window to select WAV (performance recording) file



Appendix B

Window to select XML (song score) file



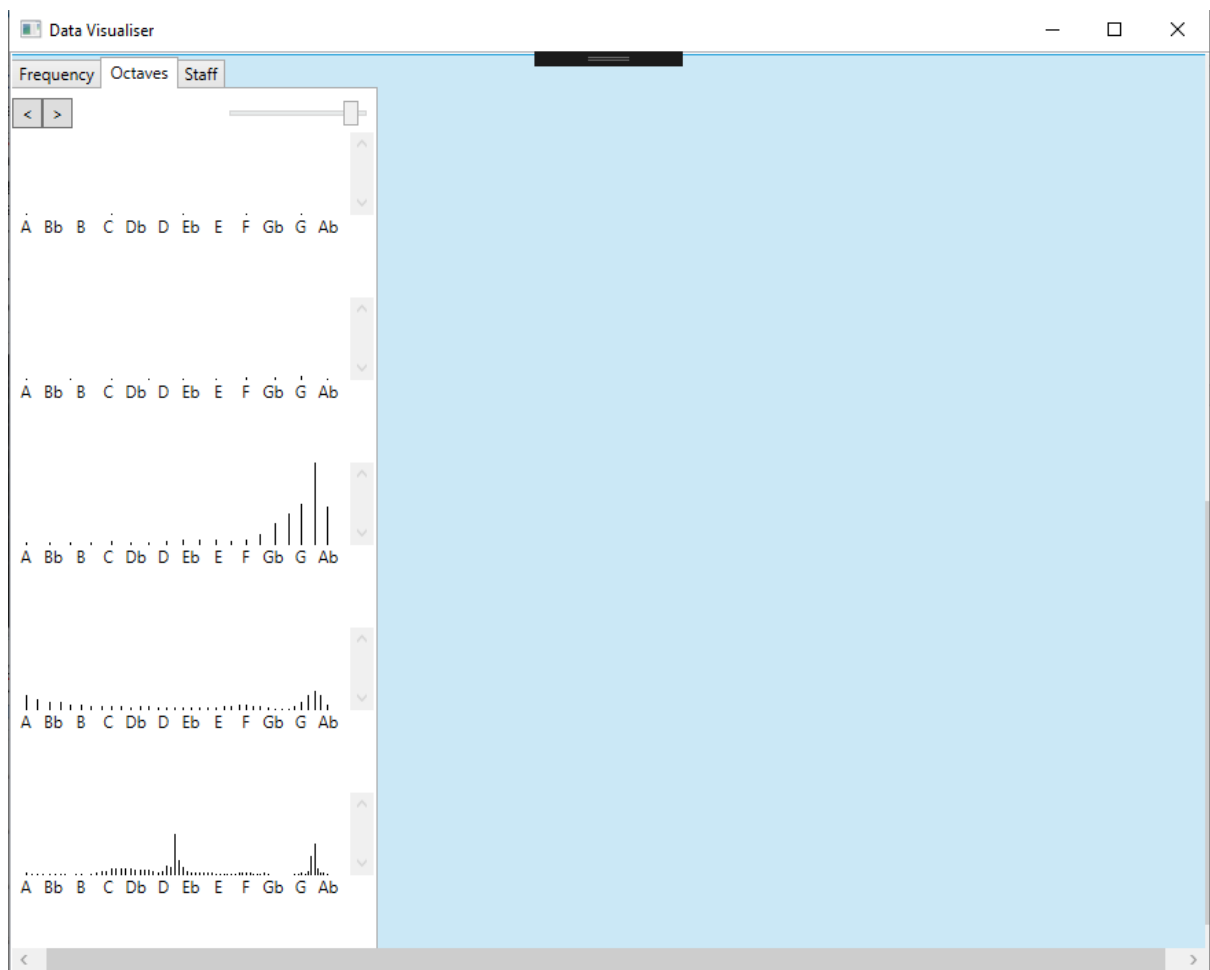
Appendix C

“Frequency” tab of data visualiser window.



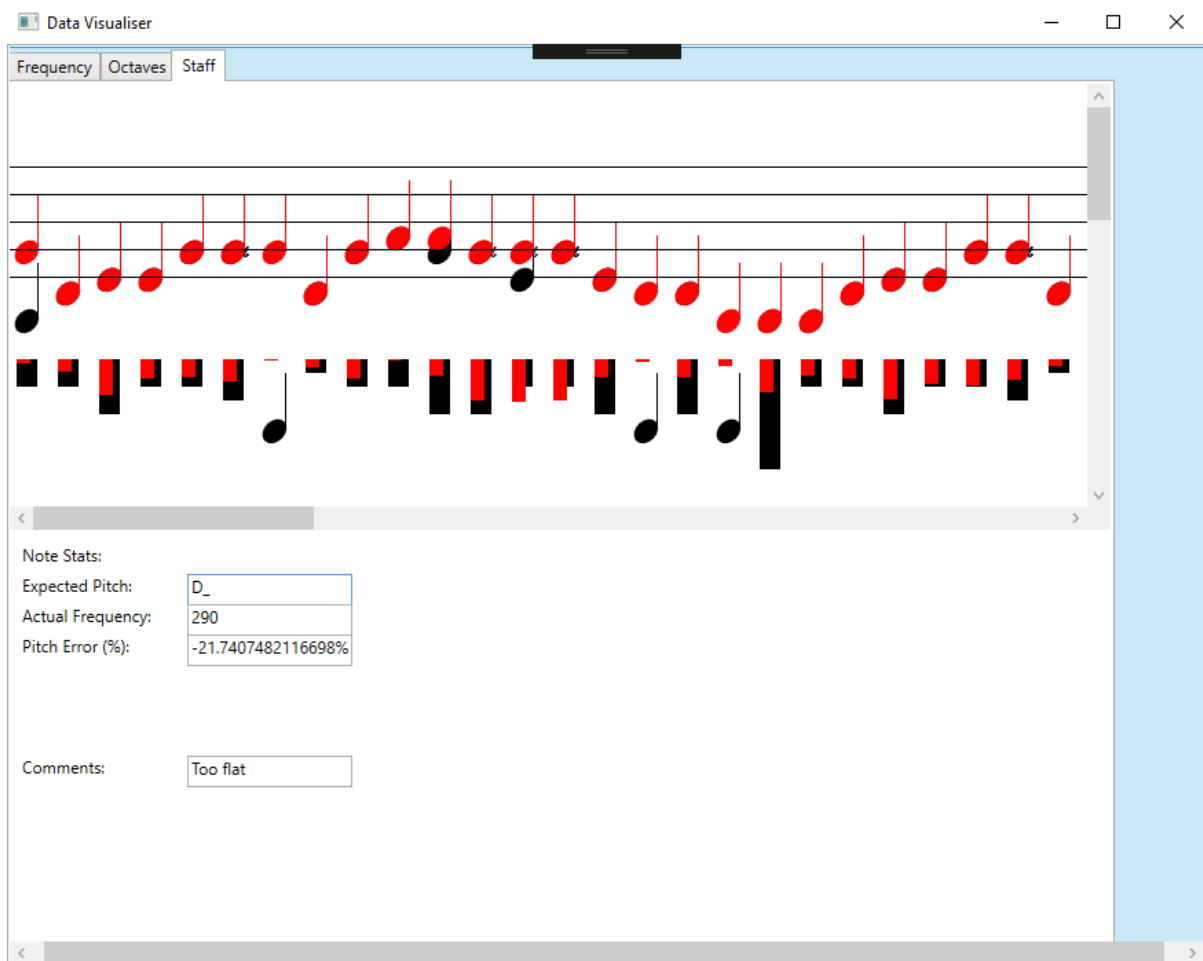
Appendix D

“Octaves” tab of data visualiser window



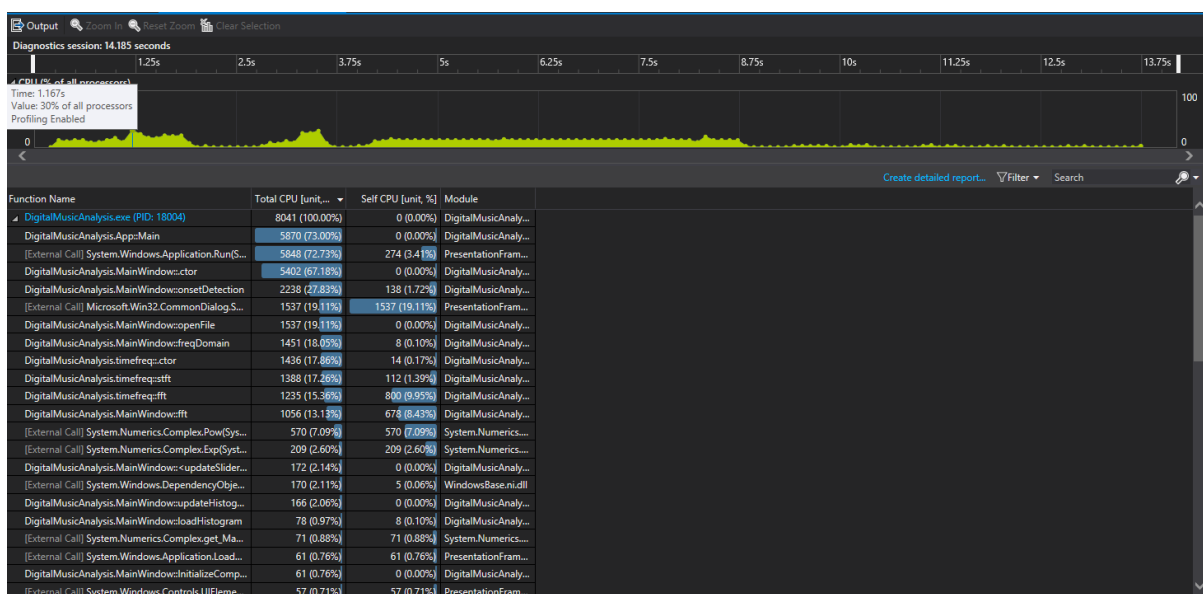
Appendix E

“Staff” tab of data visualiser window



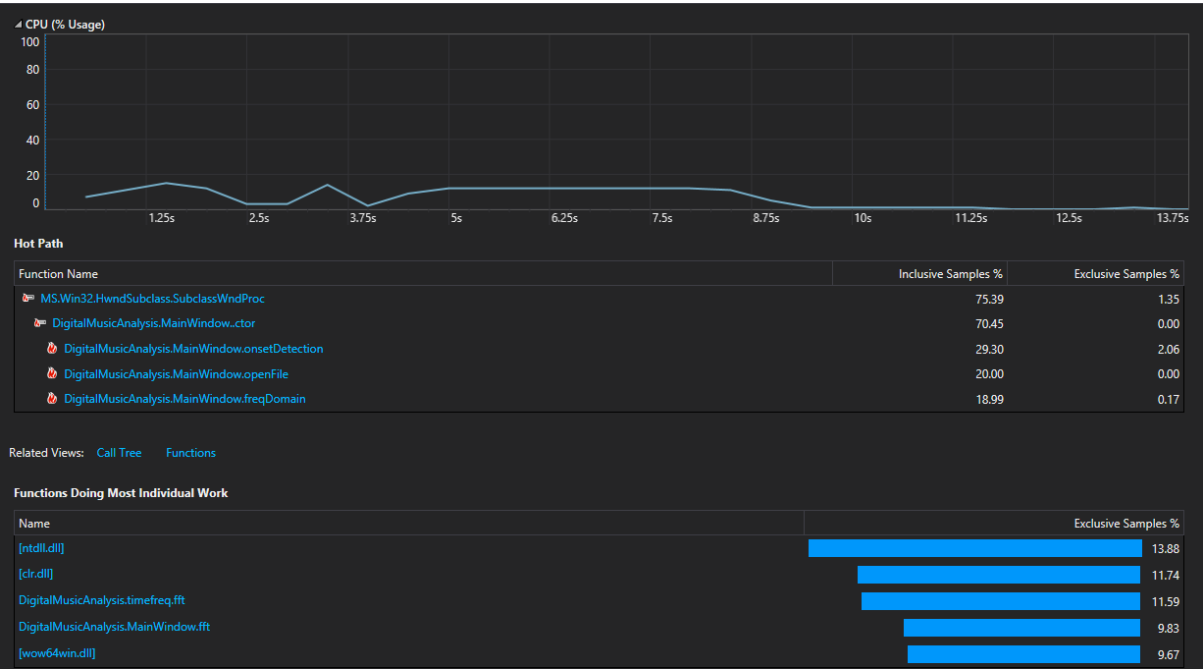
Appendix F

Initial profiler diagnostics (only ever reaches 30% usage on all cores)



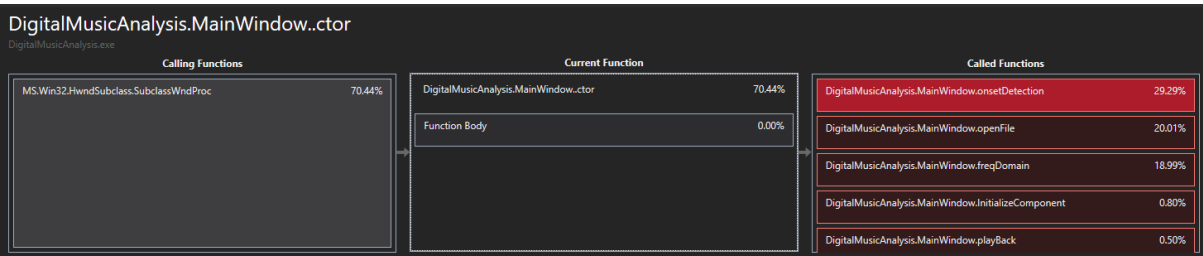
Appendix G

Profiler detailed report, showing onsetDetection, openFile, and freqDomain as the highest inclusive samples, and the Fourier transform methods in the main window and timefreq classes as the methods doing the most individual work



Appendix H

In a more detailed breakdown, we can see the majority of time spent is in the onsetDetection, openFile, and freqDomain methods.



Appendix I

Implementing timer using FileStream class to output to a file

```

namespace DigitalMusicAnalysis
{
    public partial class MainWindow : Window
    {
        private WaveFileReader waveReader;
        private wavefile waveIn;
        private timefreq stftRep;
        private float[] pixelArray;
        private musicNote[] sheetmusic;
        private WaveOut playback; // = new WaveOut();
        private Complex[] twiddles;
        private Complex[] compX;
        private string filename;
        private enum pitchConv { C, Db, D, Eb, E, F, Gb, G, Ab, A, Bb, B };
        private double bpm = 70;
        const string fileName = @"timeLogs.txt";
        FileStream stream = new FileStream(fileName, FileMode.Create);
        public int processorCount = System.Environment.ProcessorCount; //System.Environment.ProcessorCount

        public MainWindow()
        {
            Stopwatch stopwatch = new Stopwatch();
            InitializeComponent();
            filename = openFile("Select Audio (wav) file");
            string xmlfile = openFile("Select Score (xml) file");
            stopwatch.Start();
            Thread check = new Thread(new ThreadStart(updateSlider));
            loadWave(filename); //loading in file
            freqDomain(); //short term fourier
            sheetmusic = readXML(xmlfile);
            onsetDetection(); //check when notes start and finish, based on freqDomain()
            loadImage();
            loadHistogram();
            playBack();
            check.Start();

            button1.Click += zoomIN;
            button2.Click += zoomOUT;

            slider1.ValueChanged += updateHistogram;
            playback.PlaybackStopped += closeMusic;
            stopwatch.Stop();
            using(StreamWriter writer = new StreamWriter(stream))
            {
                writer.Write("Elapsed milliseconds: {0}", stopwatch.ElapsedMilliseconds);
            }
        }
    }
}

```

Appendix J

Implementing Parallel.For loop in timefreq.cs constructor


```

1  using System;
2  using System.Numerics;
3  using System.Threading.Tasks;
4
5  namespace DigitalMusicAnalysis
6  {
7      public class timefreq
8      {
9          public float[][] timeFreqData;
10         public int wSamp;
11         public Complex[] twiddles;
12         public int processorCount = System.Environment.ProcessorCount; //System.Environment.ProcessorCount
13
14         public timefreq(float[] x, int windowSamp)
15         {
16             //int ii; (not needed due to parallel for)
17             double pi = 3.14159265;
18             Complex i = Complex.ImaginaryOne;
19             this.wSamp = windowSamp;
20             twiddles = new Complex[wSamp];
21
22             //// replaced standard for loop with Parallel.For
23             Parallel.For(0, wSamp, new ParallelOptions { MaxDegreeOfParallelism = processorCount }, ii =>
24             {
25                 double a = 2 * pi * ii / (double)wSamp;
26                 twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
27             });
28
29             timeFreqData = new float[wSamp/2][];
30
31             int nearest = (int)Math.Ceiling((double)x.Length / (double)wSamp);
32             nearest = nearest * wSamp;
33
34             Complex[] compX = new Complex[nearest];
35             for (int kk = 0; kk < nearest; kk++)
36             {

```

Appendix K

Attempt at parallelising inner, and outer (separately) for loops in freqDomain method

```

private void freqDomain()
{
    stftRep = new timefreq(waveIn.wave, 2048);
    pixelArray = new float[stftRep.timeFreqData[0].Length * stftRep.wSamp / 2];
    /*
    Parallel.For(0, stftRep.wSamp / 2,
    new ParallelOptions { MaxDegreeOfParallelism = processorCount }, jj =>

    OR

    Parallel.For(0, stftRep.timeFreqData[0].Length,
    new ParallelOptions { MaxDegreeOfParallelism = processorCount }, ii =>
    */
    for (int jj = 0; jj < stftRep.wSamp / 2; jj++)
    {
        for (int ii = 0; ii < stftRep.timeFreqData[0].Length; ii++)
        {
            pixelArray[jj * stftRep.timeFreqData[0].Length + ii] = stftRep.timeFreqData[jj][ii];
        }
    }
}

```

Appendix L

Parallelising first for loop and call to fft method from stft by using a “for” loop of threads to call another method

```

65 float[][] stft(Complex[] x, int wSamp)
66 {
67     int ii = 0;
68     int jj = 0;
69     int kk = 0;
70     //int ll = 0; using parallel for, not nessisary
71     N = x.Length;
72     this.X = x;
73     //float fftMax = 0; //globalised
74
75     Y = new float[wSamp / 2][]; //globalised declaration
76
77     //for (ll = 0; ll < wSamp / 2; ll++)
78     Parallel.For(0, wSamp / 2, new ParallelOptions { MaxDegreeOfParallelism = processorCount }, ll =>
79     {
80         Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
81     });
82
83     //creating threads to call method to call fft
84     Thread[] threadArray = new Thread[processorCount];
85
86     for(int t = 0; t < processorCount; t++)
87     {
88         threadArray[t] = new Thread(stftCallFft);
89         threadArray[t].Start(t);
90     }
91
92     for(int t = 0; t < processorCount; t++)
93     {
94         threadArray[t].Join(); //joining threads after finish
95     }

```

Method which is called by for loop of threads inside “stft”

```

192 public void stftCallFft(object data)
193 {
194     int threadNum = (int)data;
195     int ChunkSize = (2 * (int)Math.Floor(N / (double)wSamp) - 1) / processorCount;
196     int start = threadNum * ChunkSize;
197     int end = Math.Min(start + ChunkSize, (2 * (int)Math.Floor(N / (double)wSamp) - 1));
198
199     Complex[] temp = new Complex[wSamp];
200     Complex[] tempFFT = new Complex[wSamp];
201
202     for (int i = start; i < end; i++)
203     {
204         for (int jj = 0; jj < wSamp; jj++)
205         {
206             temp[jj] = X[i * (wSamp / 2) + jj];
207         }
208
209         tempFFT = fft(temp);
210
211         for (int kk = 0; kk < wSamp / 2; kk++)
212         {
213             Y[kk][i] = (float)Complex.Abs(tempFFT[kk]);
214
215             if (Y[kk][i] > fftMax)
216             {
217                 fftMax = Y[kk][i];
218             }
219         }
220     }
221 }
222
223
224
225

```

Appendix M

Replacing some for loops with “Parallel.for” in the onsetDetection method

```

380
381 //first attempt at parallelising fft function call
382 //Parallel.For(0, lengths.Count, new ParallelOptions { MaxDegreeOfParallelism = processorCount }, mm =>
383 for (int mm = 0; mm < lengths.Count; mm++)
384 {
385     int nearest = (int)Math.Pow(2, Math.Ceiling(Math.Log(lengths[mm], 2)));
386     twiddles = new Complex[nearest];
387     compX = new Complex[nearest];
388
389     //merged below loops together into single parallel version
390     //for (ll = 0; ll < nearest; ll++)
391     //for (int kk = 0; kk < nearest; kk++)
392
393     Parallel.For(0, nearest, new ParallelOptions { MaxDegreeOfParallelism = processorCount }, pp =>
394     {
395         double a = 2 * pi * pp / (double)nearest;
396         twiddles[pp] = Complex.Pow(Complex.Exp(-i), (float)a);
397
398         if (pp < lengths[mm] && (noteStarts[mm] + pp) < waveIn.wave.Length)
399         {
400             compX[pp] = waveIn.wave[noteStarts[mm] + pp];
401         }
402         else
403         {
404             compX[pp] = Complex.Zero;
405         }
406     });
407
408     Y = new Complex[nearest];
409
410     Y = fft(compX, nearest);
411
412     absY = new double[nearest];
413
414     double maximum = 0;
415     int maxInd = 0;
416
417     //for (int jj = 0; jj < Y.Length; jj++) //changed to Parallel.For without issues
418     Parallel.For(0, Y.Length, new ParallelOptions { MaxDegreeOfParallelism = processorCount }, jj =>
419     {
420         absY[jj] = Y[jj].Magnitude;
421         if (absY[jj] > maximum)
422         {
423             maximum = absY[jj];
424             maxInd = jj;
425         }
426     });
427
428     for (int div = 6; div > 1; div--)
429     {
430         if (maxInd > nearest / 2)

```

References

Intel® Core™ i7-6700HQ Processor (6M Cache, up to 3.50 GHz) Product Specifications. (2019). Retrieved 6 September 2019, from <https://ark.intel.com/content/www/us/en/ark/products/88967/intel-core-i7-6700hq-processor-6m-cache-up-to-3-50-ghz.html>

Task Parallel Library (TPL). (2019). Retrieved 13 September 2019, from <https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl>

System.Threading Namespace. (2019). Retrieved 13 September 2019, from <https://docs.microsoft.com/en-us/dotnet/api/system.threading?view=netframework-4.8>

System.Threading.Tasks Namespace. (2019). Retrieved 13 September 2019, from <https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks?view=netframework-4.8>