Digital Music Analysis Parallelisation Application

Student Details

Name: Daniel Pretlove

Student Number: N10193308

Digital Music Analysis Details

Name: Digital Music Analysis

Language: C#

Integrated Development Environment: Microsoft Visual Studio 2019

Description: The digital music analysis is a C# Window Form application, that is used for people that are starting to learn the violin, that can't distinguish between the sounds of the different notes, and does not have an instructor to help them distinguish between them. The application translates a .wav file and is transformed into an image and compares it to another file which is an .xml form which marks out the errors that is made by the learner, then it analysis the frequency of each sample that is being played, The frequency, octaves and the staff tabs in the window allows you to select the 3 options through every particular range of the first 2048 samples of the input sounds. Please refer to Appendix A to see the frequency, octaves, and the staff view

How the Sequential Application Works

As it was mentioned before the sequential application that will be parallelized is the Digital Music Analysis in C#, in which the application is to produce analysed notes in a .wav and .xml file from the user, The MainWindow.xaml file that plays the .wav file and displays the information as a frequency, octaves and staff, within the sequential application, there are 4 classes which are musicNotes.cs, noteGraph.cs, timefreq.cs and wavefile.cs where the main class is a XAML file named as MainWindow.xaml.cs. Where most of the code inside of MainWindow.xaml.cs are being produced.

The musicNote.cs class has code related to how a musical node is being produced, which includes information of the note pitch, duration, flat, error, staffPos, mult and frequency. The Second class noteGraph.cs, is to set the heighted bar graph to the musical note being played where it is then plotted on top of the xml and will then show an overview of the height of the base frequency of each note being played. The third class timefreq.cs analysis the time-frequency to the signals of the musical notes that are being played from the MainWindow.xaml.cs class and the timefreq will then compute the time frequency. The last class, wavefile.cs is used to help compute and operate a wav file. The class is used as a BinaryReader to read and compute the chunk id, chunk size, format audio, number of channels, sample, and byte rate etc.

The musicNote, noteGraph, timefreq and wavefile classes will be used within the MainWindow.xaml.cs class and within the MainWindow() method that produces the graphical user interface that is used to analyse the application to the musical note that are being played from the wav file and will run through all the code and produces a GUI that will operate it's computation accordingly. Then it will prompt the user to select the wav and xml file that will load the wav file and make a time-frequency graph which then loads and reads the xml file that produces the onsetDetection computation and a histogram graph that will be playing the music from the wav file that the user has selected. Please refer to Appendix B for the class diagram image

The Main functions has functions that are used to help increase the usability within the application such as the openFile function which allows you to open either a wav or xml file. The application then loads the digital audio file using the loadWave function and is then converted into an array. The main function then calls the freqDomain function that analyses the data we gathered from the loadWave function and will calls the time from the timefreq.cs that then transform the data into time-frequency then after converting the wave file data into the time frequency array, the main function will then load the sheet music file using the readXML function. Then the onSetDetection function will analyse the nodes from start to finish and will show the stats in the staff tab.

Analysis of Potential Parallelism

When I ran the application I realized that the application was only being processed with only one thread, where the two major functions of freqDomain and onsetDetection are doing most of the work in the application, By parallelizing the two function it will reduce the run time of the application.

The freqDomain function does 28.04% of the applications work, in which the function is used to transform data into time-frequency representations that calls the timefreq.cs and passes the wave file with the size of the wSamp. Within the profile report you can see that the loop is running 2048 times and transforming the data into time frequency representations. Please refer to Appendix E to see the profiling report that shows the freqDomain function.

By analysing the freqDomain function you can see that the stftRep property is doing most of the work. In which you can see that the stft function is using 28.35% of the applications work, so therefore it is worth exploring the timefreq class stft function. Please refer to Appendix F to see the profiling report of the stft property in the timefreq class.

The fft function is doing 18.05% of the applications work, where the function is used for pitch detection. Please refer to Appendix G to see the fft functions profiling report which uses the divide and conquer algorithm.

The profile report of the fft function you can see that the for loop is calling the fft function recursively, which is using the divide and conquer algorithm and by analysing the fft function I found that it the function is using 9.01% out of 18.05% of the functions usage which is using about 50% of the functions usage, so it is worth exploring the for loop to see if it is safe to parallelize the for loop.

The onsetDetection function does 28.43% of the applications work, in which the function is used to measure each note and the frequency of the notes over each duration. Please refer to appendix H to see the profiling report that is referring to the loop that is using the HFC array.

Within the profile report, the nested loop is looping the HFC array with the values of the frequency data, and it is iterating a fixed number of times and because of that it is safe to parallelize the onsetDetection function.

Another major for loop that could be a potential iteration to parallelize is the for loop that determines the start and end time of the notes being played which is based on the onsetDetection. Please refer to appendix I for the loop that determines the start end time of the notes being played.

The profile report shows that it is doing 7.18% out of 28.43% of the functions work, There are also many data dependencies involved In this for loop and it is worth parallelizing the for loop since it is doing close to 50% of the functions job.

Parallelism Abstractions

Threading

I have used threading for this application, by creating multiple threads with the thread class and I have used the Environment.ProcessorCount to get the number of processors on the current machine it is being run on. I have created threads by using the Environment.ProcessorCount Property of the system, which the application then runs on every core of the machine, that was created on the timefreq.cs and was used on both the freqDomain and the onsetDetection functions with the private class property of stftRep.

Task Parallel Library (TPL)

Task Parallel Library contains Parallel.For which uses the default task scheduler to schedule the iterations, which uses the current thread as well as several thread pools threads. I have used Parallel.For to parallelize the primary for loop that doing most of the work of the application to speed up the execution of the application.

Timing and Profiling

Before Parallelisation

Digital Music Analysis Application Before Parallelization (Time by Milliseconds)				
Functions	Test1	Test2	Test3	Avg
loadWave	22	20	20	20.6
freqDomain	2525	2445	2440	2470
Sheetmusic	1	1	1	1
onsetDetection	2512	2510	2539	2520.3
loadImage	4	4	4	4
loadHistogram	2	2	2	2
playBack	52	52	53	52.3
check.start()	0	0	0	0
Overall Application	5124	5039	5065	5076

How I got the accurate execution sequence of each function and the overall application, is that I've used the StopWatch property that is inside of the System.Diagnostics library, by doing so, I was able to determine where most of the work of the application are being handled, and the timing table shows that the onsetDetection and the freqDomain functions are doing most of the work of the application. By parallelizing these two functions it can reduce the overall run time of the application. Please refer to Appendix J to see the applications overall workload.

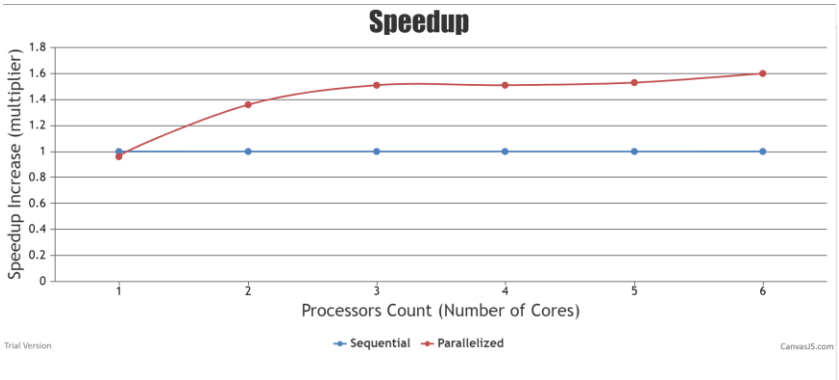
Within the profiler report, you can see that the overall workload of the application is in the MainWindow.ctor() which is using 83.66% application, and by analysising the onsetDetection function you can see that the function is using most of it's time behind the for loop which determines the and the finishing time of the notes based on the onset detection and the fft function which is a divide and conqure algorithm.

After Parallelisation

Digital Music Analysis Application After Parallelization (Time by Milliseconds)				
Functions	Test1	Test2	Test3	Avg
loadWave	23	21	21	21.6
freqDomain	1067	1110	1071	1082.6
Sheetmusic	1	1	1	1
onsetDetection	1938	1935	1928	1933.6
loadImage	4	5	5	4.6
loadHistogram	2	2	3	2.3
playback	55	56	54	55
check.start()	0	0	0	0
Overall Application	3094	3133	3086	3103.6

The parallelized timing table shows each of the function with 6 cores. You can see that the freqDomain and the onsetDetection execution timing has significantly decreased, Where the freqDomain average execution time was 2470 milliseconds long in the sequential version and now the average time is 1082.6 milliseconds within the parallelization version, The sequential execution time of the onsetDetection function is 2520.3 millisecond long and the average execution time is now 1933.6 milliseconds long, which the overall application was averaging 5076.3 of the execution time previously but the average time is now 3103.6 millisecond long.

Result



The graphs shows that the speedup increase of the parallelization version In the application, is the number of cores that it is running, which shows that the application speedup increases drastically with the application running on 2 to 3 cores parallelized but then the speedup performance boost does not show a significant increase afterwards.

Scalability

The scalability of the application is a scalable parallelism since the speedup does in fact give a great increase of the performance with the number of processors you put in.

Test Parallelized and Sequential Project Results

By Outputting the CompX data onto a .txt, for both the sequential and parallelization version of the project, in which the variable is used to determine the start and finishing times of notes based on the onsetDetection. Please refer to Appendix C for the stream writing code.

By analysing both the sequential and the parallelized version, you can see that the parallelization data is correct based on the data being identical to the sequential version's CompX data. Please see Appendix D for the Sequential and Parallelization CompX data results.

Software, Tools and Techniques

Software

Visual Studio 2019

I used Visual studio 2019 the community edition, to edit, and run the Digital Music Analysis Application that was written in C#.

Lucid Chart

I used lucid chart to create a class diagram that allowed to visualize how all the classes and data dependences worked together in the application.

Tools

I used the performance profiler in visual studio 2019 to get the report of the applications performance. Which gives a detailed report of how much usage each of the function are using in the application, and you can check inside of the function to see what piece of code is doing most of the functions work.

Techniques

I used the Task Parallel Library to parallelize the Digital Music Analysis Application and the Thread class for the application to run on multiple cores which is included inside of the .NET framework of 4.8.0.

Overcoming Barriers

When parallelizing the Digital Music Analysis application the first obstacle that I faced was understanding how the application works and how the application is processing the data, after watching Wayne Kelly's video, it helped me a lot in understanding how the freqDomain and the onsetDetection functions in the MainWindow.xaml.cs class and the stft function in the timefreq.cs class.

The Second Obstacle was that I wasn't able to print the functions time sequences on the console because of the modules that I have not referenced onto the application, so once I have referenced the modules I needed I was able to print the time sequences of the function to understand how long, it takes for each function to finish within the MainWindow.xaml.cs class.

The final obstacle was getting the application to run on multiple cores, which I needed to understand how the timefreq class worked, to create an array of threads with the size of the number of processors on the current machine and then joining all of the threads for the stft function which is used to get the wSamp to run on multiple threads.

Modified code changes in the parallelization version

Changes in the onsetDetection function

by analysing the onsetDetection function, in line 418 of the original code, which determines the start and finish time of notes based on the onset detection I changed the original for loop with a Parallel.For to run the loop in parallel. Which the loop was doing most of the work of the onsetDetection function.

Status: Successful

Another for loop that I have parallelized within the onsetDetection function is the for loop that loops time frequency representations in line 362, which does another major portion of the function.

Status: Successful

Changes in the Timefreq.cs

In line 19 of the timefreq.cs, where the for loop is calculating the value of the wSamp and placing it into the twiddles array, I have changed the for loop to Parallel.For for the loop to run in parallel.

Status: Successful

In line 66 I have replace the original for loop to a parallelized for loop, where the application loops over the wSamp, To reduce the workload of a single processor/ thread, I have dedicated the workload of the wSamp loop onto multiple threads/processor by creating a threads array that fills out, and starts all of the current threads on your current machine, then to ensure the threads do not get pushed into a deadlock state where every thread is constantly waiting for each other, I have used threads.join where the current thread pauses and waits for another thread to complete its task and then the current thread will continue once it's his turn to complete his task and the cycle will continue up until all of the threads have completed all of their tasks.

In line 807, by replacing the original for loop to a parallelization for loop in which the loop is a pitch detection, By parallelizing the for loop it has increased the execution time to more than 10 times the original execution time. Please refer to Appendix K for the pitch detection code.

Status: Failed

Reflection

During my time up taking the unit CAB401 I have learnt amazing knowledge in how to parallelize in multiple languages / libraries, I have learnt how multithreading and how mutex locks work in CAB403, so I came into this unit with a brief knowledge in how parallelization works, but this unit has taught me, and understanding when to parallelize a loop, and allowing the application to run on multiple threads.

My attempt for the digital music analysis gave the application an average of a 64% speedup runtime in the application, where I managed to get a sub-linear speedup performance in the application.

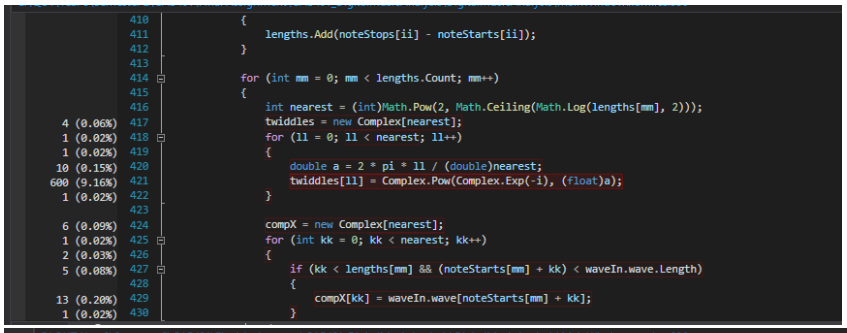
By understand how the divide and conquer algorithm works and removing the recursion within the fft function I could have rewritten the algorithm for the application to run more efficiently.

Source Code

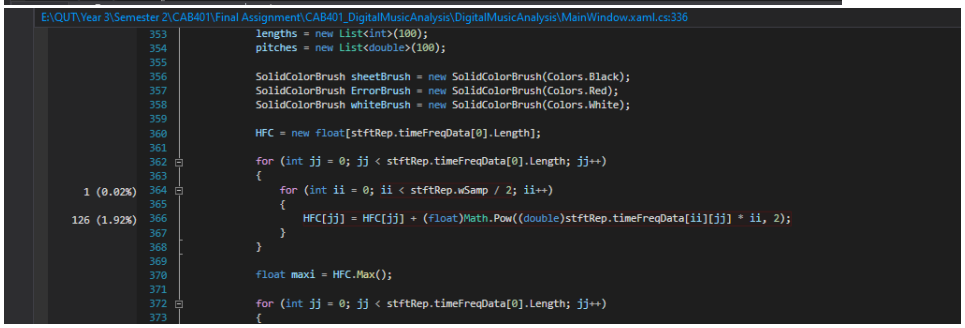
MainWindow.xaml.cs

Before and After parallelization of the onsetDetection function.

Before Parallelization

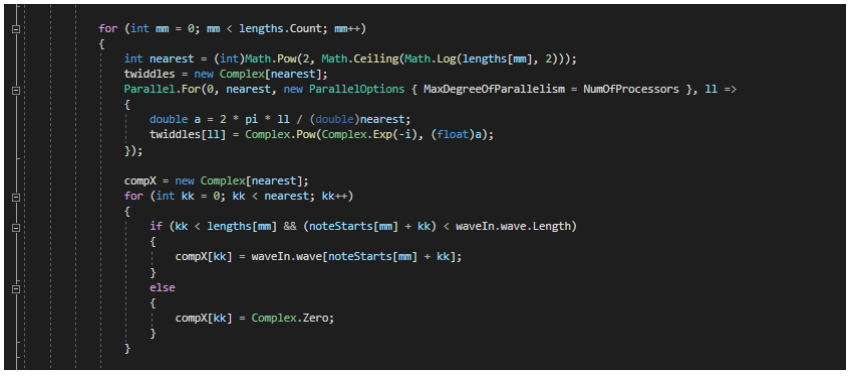


```
                {
                    lengths.Add(noteStops[ii] - noteStarts[ii]);
                }

                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];
                        }
```

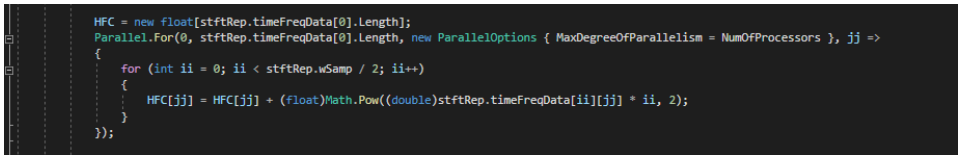


```
E:\QUT\Year 3\Semester 2\CAB401\Final Assignment\CAB401_DigitalMusicAnalysis\DigitalMusicAnalysis\MainWindow.xaml.cs:336
                lengths = new List<int>(100);
                pitches = new List<double>(100);

                SolidColorBrush sheetBrush = new SolidColorBrush(Colors.Black);
                SolidColorBrush ErrorBrush = new SolidColorBrush(Colors.Red);
                SolidColorBrush whiteBrush = new SolidColorBrush(Colors.White);

                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);
                    }
                }

                float maxi = HFC.Max();

                for (int jj = 0; jj < stftRep.timeFreqData[0].Length; jj++)
                {
```

After Parallelization



```
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];
    Parallel.For(0, nearest, new ParallelOptions { MaxDegreeOfParallelism = NumOfProcessors }, 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;
        }
    }
```



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

Timefreq.cs

Before and After parallelization of the timefreq class

Before Parallelization

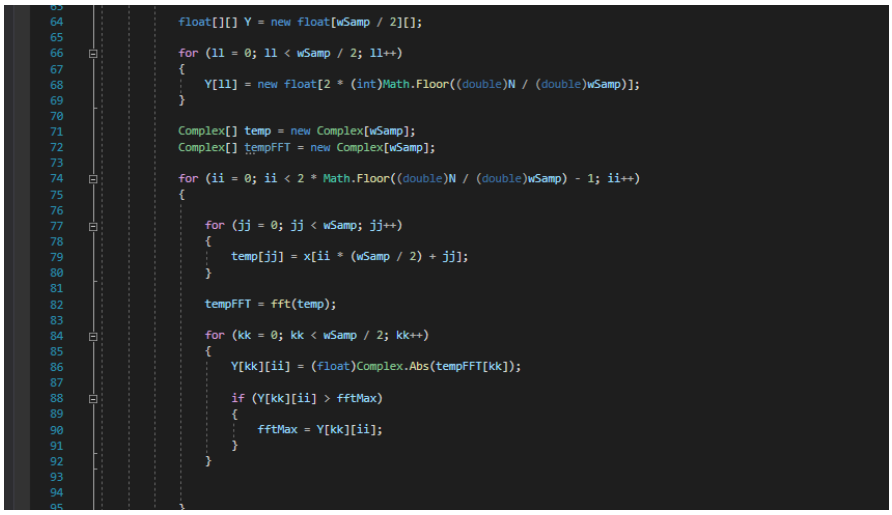
```
11
                1 reference
12              public timefreq(float[] x, int windowSamp)
13              {
14                  int ii;
15                  double pi = 3.14159265;
16                  Complex i = Complex.ImaginaryOne;
17                  this.wSamp = windowSamp;
18                  twiddles = new Complex[wSamp];
19                  for (ii = 0; ii < wSamp; ii++)
20                  {
21                      double a = 2 * pi * ii / (double)wSamp;
22                      twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
23                  }
```

After Parallelization

```
E:\QUT\Year 3\Semester 2\CAB401\Final Assignment\CAB401_DigitalMusicAnalysis\DigitalMusicAnalysisParallelized\timefreq.cs:146
19          public float fftMax = 0;
20          public timefreq(float[] x, int windowSamp)
21          {
22              double pi = 3.14159265;
23              Complex i = Complex.ImaginaryOne;
24              this.wSamp = windowSamp;
25              twiddles = new Complex[wSamp];
26              Parallel.For(0, wSamp, new ParallelOptions { MaxDegreeOfParallelism = NumOfProcessors }, ii =>
27              {
28                  double a = 2 * pi * ii / (double)wSamp;
29                  twiddles[ii] = Complex.Pow(Complex.Exp(-i), (float)a);
30              });
31
```

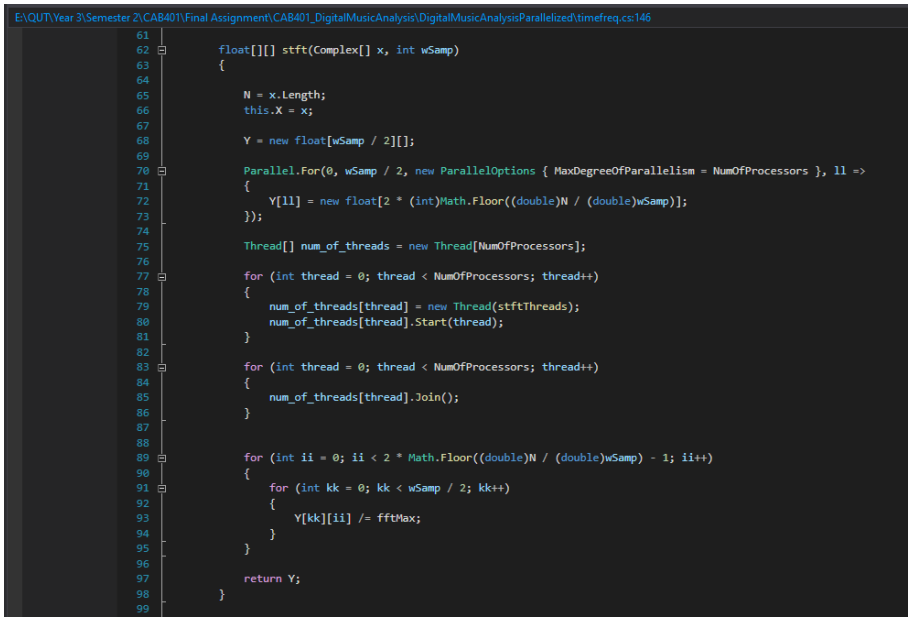
Before and after threading of the timefreq class

Before threading



```
float[][] Y = new float[wSamp / 2][];

for (ll = 0; ll < wSamp / 2; ll++)
{
    Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
}

Complex[] temp = new Complex[wSamp];
Complex[] tempFFT = new Complex[wSamp];

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 = fft(temp);

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

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


}
```

After threading



```
E:\QUT\Year 3\Semester 2\CAB401\Final Assignment\CAB401_DigitalMusicAnalysis\DigitalMusicAnalysisParallelized\timefreq.cs:146

float[][] stft(Complex[] x, int wSamp)
{

    N = x.Length;
    this.X = x;

    Y = new float[wSamp / 2][];

    Parallel.For(0, wSamp / 2, new ParallelOptions { MaxDegreeOfParallelism = NumOfProcessors }, ll =>
    {
        Y[ll] = new float[2 * (int)Math.Floor((double)N / (double)wSamp)];
    });

    Thread[] num_of_threads = new Thread[NumOfProcessors];

    for (int thread = 0; thread < NumOfProcessors; thread++)
    {
        num_of_threads[thread] = new Thread(stftThreads);
        num_of_threads[thread].Start(thread);
    }

    for (int thread = 0; thread < NumOfProcessors; thread++)
    {
        num_of_threads[thread].Join();
    }


    for (int ii = 0; ii < 2 * Math.Floor((double)N / (double)wSamp) - 1; ii++)
    {
        for (int kk = 0; kk < wSamp / 2; kk++)
        {
            Y[kk][ii] /= fftMax;
        }
    }

    return Y;
}
```

```
public void stftThreads(object data)
{
    int threadId = (int)data;
    int size = (2 * (int)Math.Floor(N / (double)wSamp) - 1) / NumOfProcessors;
    int StartThread = threadId * size;
    int EndThread = Math.Min (StartThread + size, (2 * (int)Math.Floor(N / (double)wSamp) - 1));
    Complex[] temp = new Complex[wSamp];
    Complex[] tempFFT = new Complex[wSamp];

    for (int ii = StartThread; ii < EndThread; ii++)
    {

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

        tempFFT = fft(temp);

        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];
            }
        }
    }
}
```

Compile Instructions

- Open Visual Studio
- Open Digital Music Analysis Project
- Press Run or F5 to Compile the project
- Once project is running you are required to select a wav file, which will be Jupitar.wav in this scenario
- Now that the wav file has been selected, you are required to select an xml file, while will Jupitar.xml in this scenario
- Now wait for the project to read the Jupitar datasets
- Once done the Jupitar.wav file will be running through the Windows Form application.

Hardware Requirements

Processor: Intel® Core™ i5-8400 Processor

Base Frequency: 2.80GHz

Max Turbo Frequency: 4.00GHz

GPU: Nvidia GTX 1050TI

Number of Cores: 6

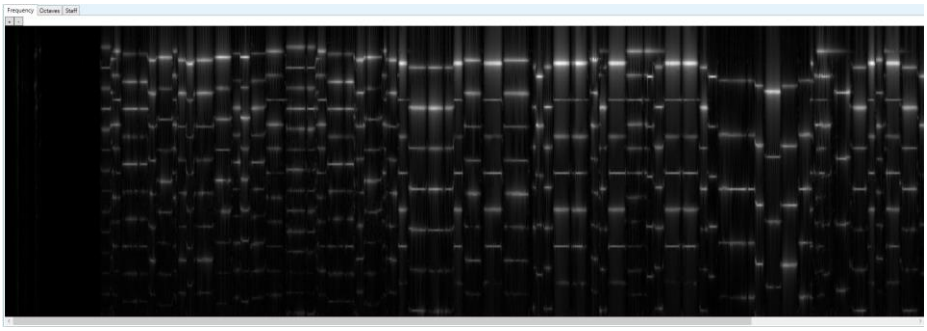
Number of Threads: 6

Memory: RAM: 16 GB

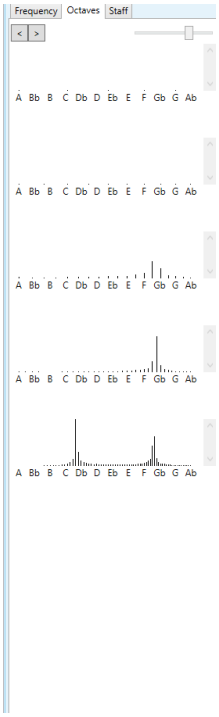
Appendices:

Appendix A:

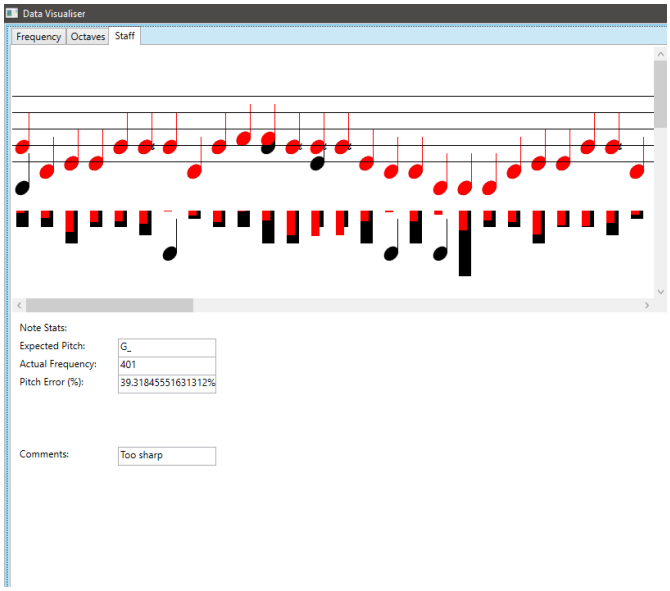
Frequency view on Jupitar.wav



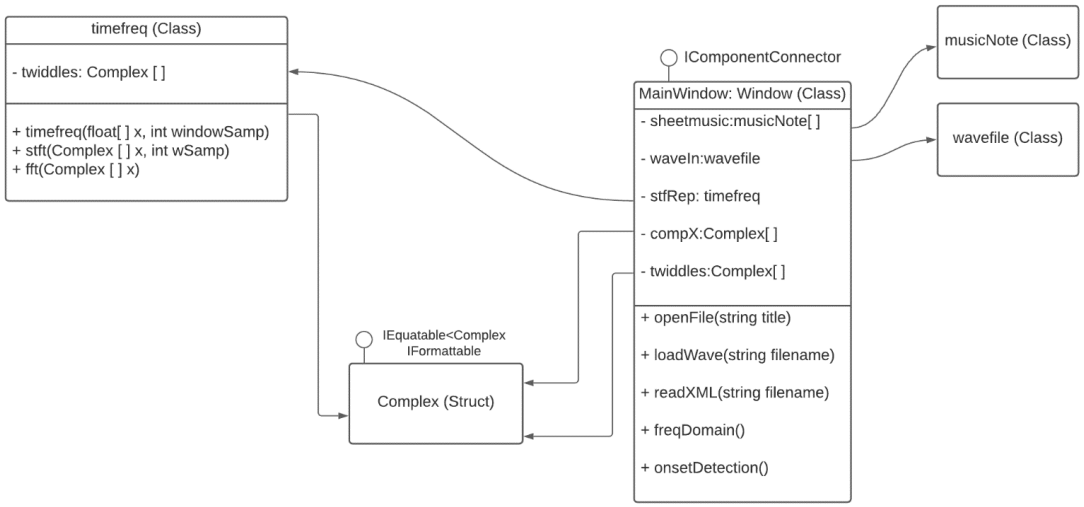
Octaves view on Jupitar.wav



Staff view on Jupitar.wav



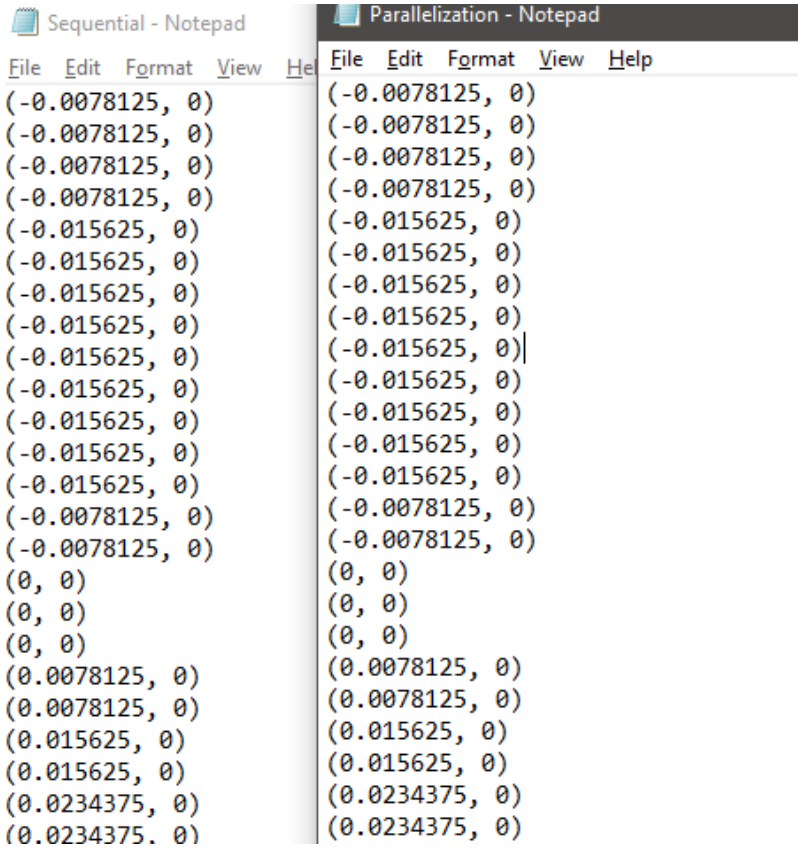
Appendix B



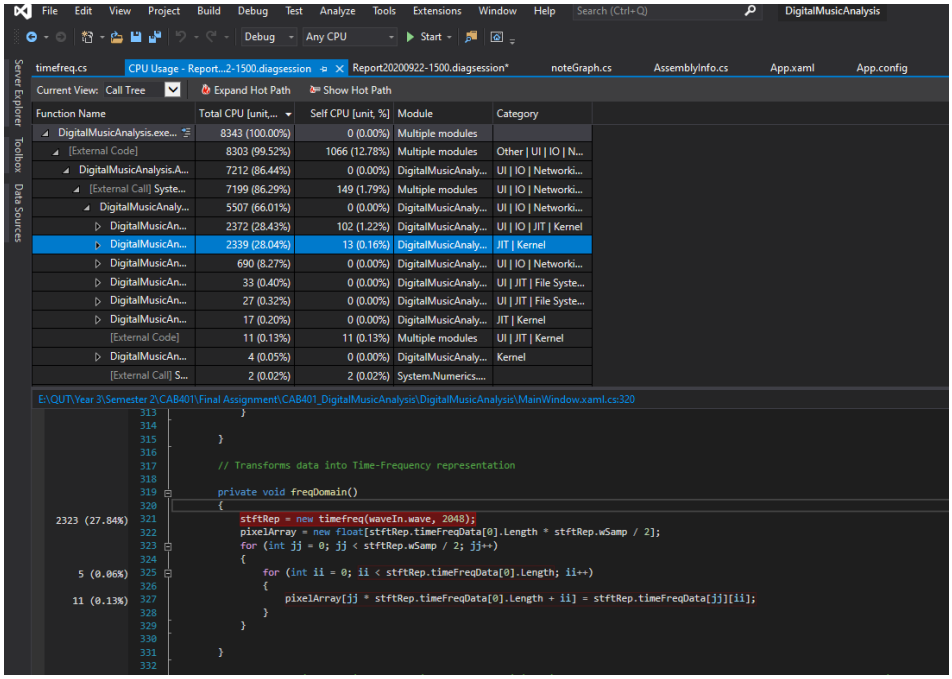
Appendix C

```
string path = "E://QUT/Year 3/Semester 2/CAB401/Final Assignment/CAB401_DigitalMusicAnalysis/DigitalMusicAnalysis";

using (StreamWriter outputFile = new StreamWriter(System.IO.Path.Combine(path, "Sequential.txt")))
{
    foreach (var line in compX)
    {
        outputFile.WriteLine(line);
    }
```

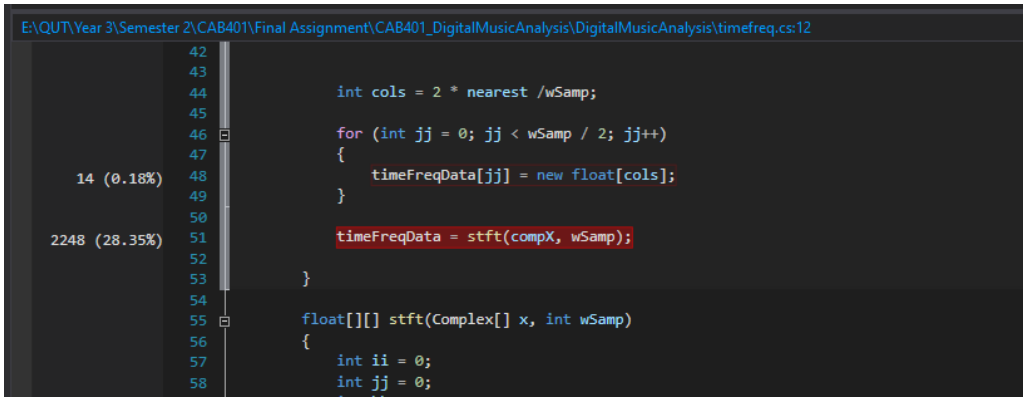
Appendix D



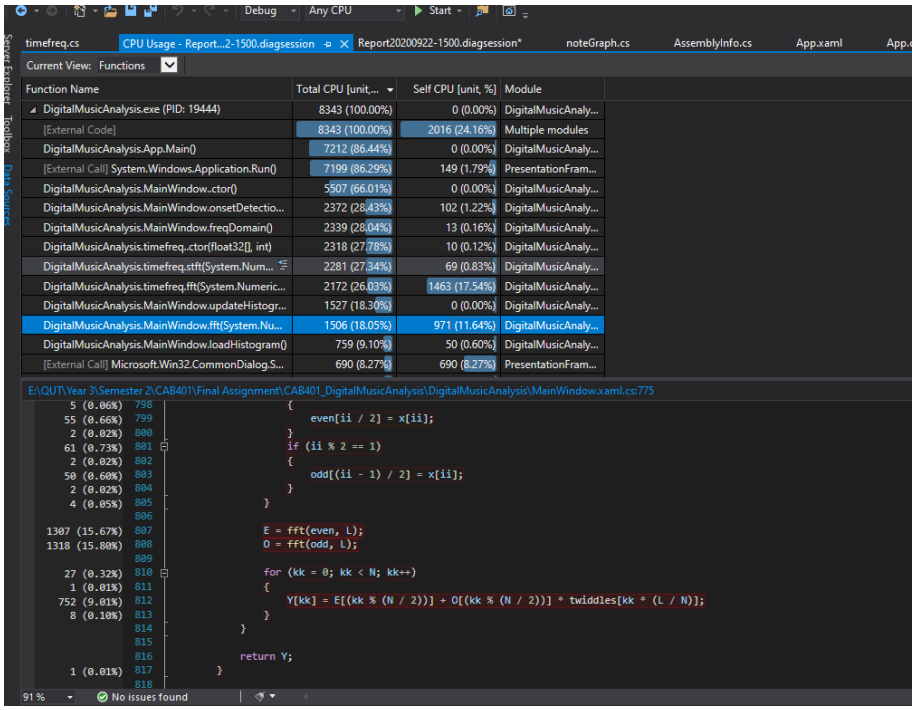
Appendix E



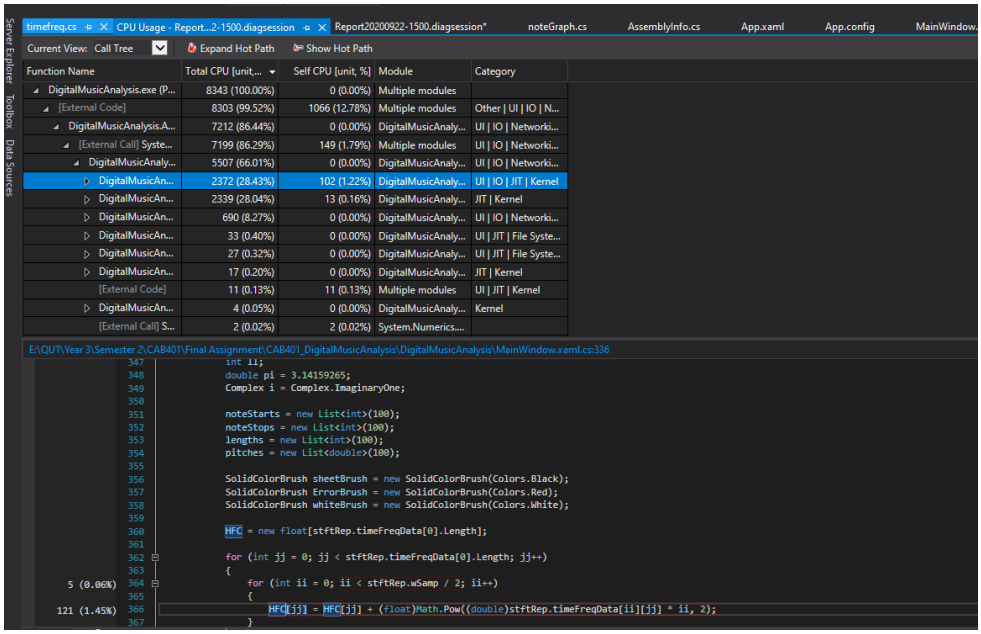
Appendix F



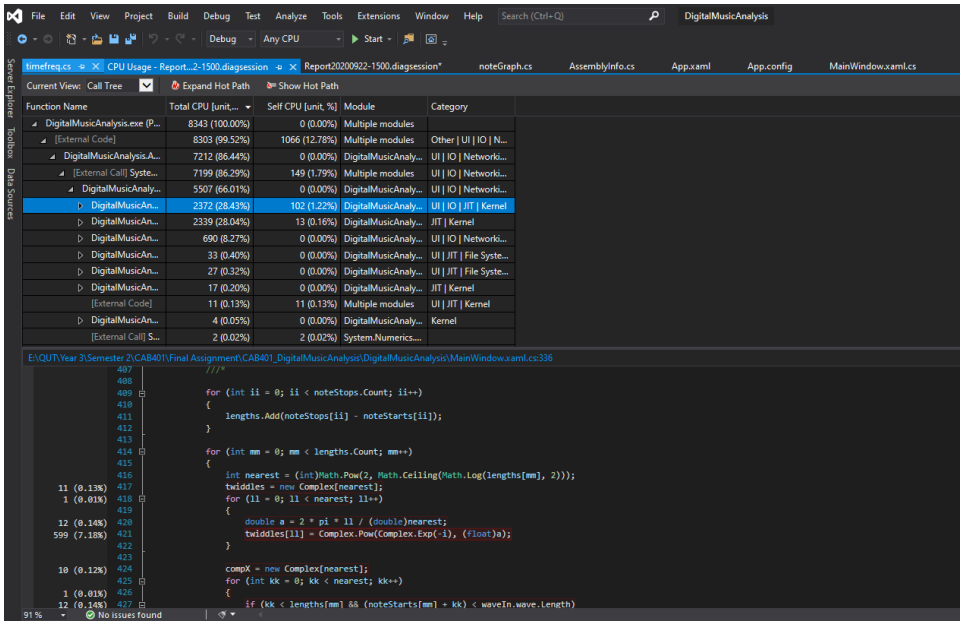
Appendix G



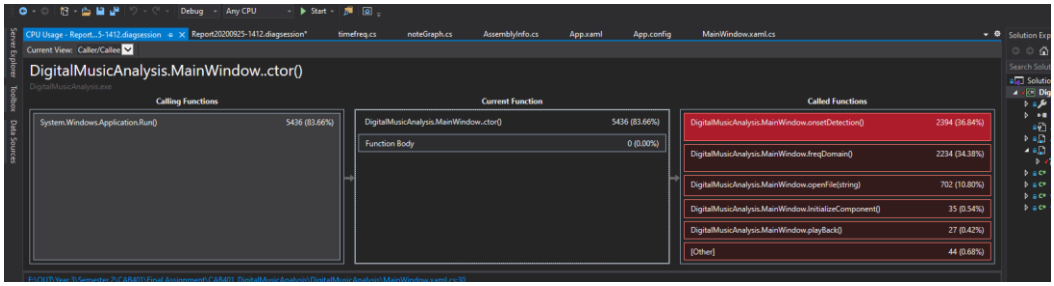
Appendix H



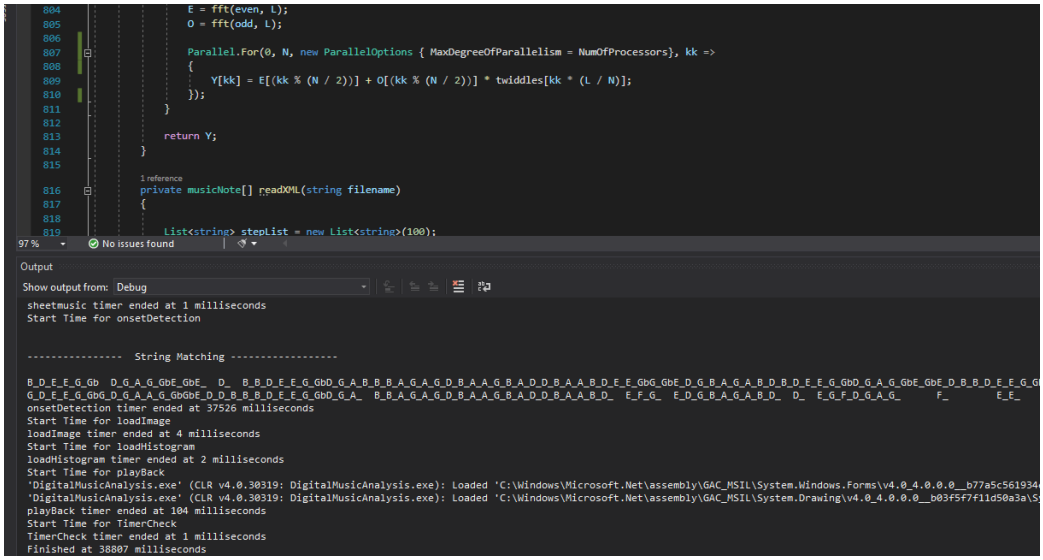
Appendix I



Appendix J



Appendix K



```
            E = fft(even, L);
            O = fft(odd, L);

            Parallel.For(0, N, new ParallelOptions { MaxDegreeOfParallelism = NumOfProcessors}, kk =>
            {
                Y[kk] = E[(kk % (N / 2))] + O[(kk % (N / 2))] * twiddles[kk * (L / N)];
            });
        }

        return Y;
    }

    1 reference
    private musicNote[] readXML(string filename)
    {

        List<string> stepList = new List<string>(100);
```

```
97 %   No issues found

Output
Show output from: Debug
sheetmusic timer ended at 1 milliseconds
Start Time for onsetDetection


---------------  String Matching -----------------

B_D_E_E_G_Gb  D_G_A_G_GbE_GbE_  D_  B_B_D_E_E_G_GbD_G_A_B_B_A_G_A_G_D_B_A_A_G_B_A_D_D_B_A_A_B_D_E_E_GbG_GbE_D_G_B_A_G_A_B_D_B_D_E_E_G_GbD_G_A_G_GbE_GbE_D_B_B_D_E_E_G_G
G_D_E_E_G_GbG_D_G_A_A_G_GbGbE_D_D_B_B_B_D_E_E_G_GbD_G_A_  B_B_A_G_A_G_D_B_A_A_G_B_A_A_D_D_B_A_A_B_D_  E_F_G_  E_D_G_B_A_G_A_B_D_  D_  E_G_F_D_G_A_G_      F_        E_E_
onsetDetection timer ended at 37526 milliseconds
Start Time for loadImage
loadImage timer ended at 4 milliseconds
Start Time for loadHistogram
loadHistogram timer ended at 2 milliseconds
Start Time for playBack
'DigitalMusicAnalysis.exe' (CLR v4.0.30319: DigitalMusicAnalysis.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Windows.Forms\v4.0_4.0.0.0__b77a5c561934
'DigitalMusicAnalysis.exe' (CLR v4.0.30319: DigitalMusicAnalysis.exe): Loaded 'C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Drawing\v4.0_4.0.0.0__b03f5f7f11d50a3a\S
playBack timer ended at 104 milliseconds
Start Time for TimerCheck
TimerCheck timer ended at 1 milliseconds
Finished at 38807 milliseconds
```